IziBuddy — The Complete Manual

IziBuddy — The Complete Manual

User, administrator and operator manual

InZeMobile

TABLE DES MATIÈRES

Partie 1 — Foundations

Chapter I — What IziBuddy is 11
Chapter II — The mental model 16
Chapter III — Installing the dashboard 20
Chapter IV — Installing the browser extension 24
Chapter V — The first five minutes 28

Partie 2 — Working with templates

Chapter VI — Anatomy of a prompt 35
Chapter VII — Creating, editing, duplicating, trashing, restoring, purging 39
Chapter VIII — Markers and lifecycle states 46
Chapter IX — Categories 50
Chapter X — Tags and the tag hierarchy 54
Chapter XI — Containers: folders vs collections 58
Chapter XII — Searching and filtering 63
Chapter XIII — Views: tiles, table, split pane, dates, density 68
Chapter XIV — Bulk operations 73
Chapter XV — Import and export 78

Partie 3 — Authoring with POO²L (manual-side surface)

Chapter XVI — The smart editor 85
Chapter XVII — Parameters from the UI 90
Chapter XVIII — Constants from the UI 94
Chapter XIX — Attachments 97
Chapter XX — Inheritance and includes from the UI 101
Chapter XXI — Notes and sections 105
Chapter XXII — Rendering and instantiation 112

Chapter XXIII — Reading map 116

Partie 4 — Collaboration

Chapter XXIV — Sharing vs duplicating 121
Chapter XXV — Groups 124
Chapter XXVI — Invitations 128
Chapter XXVII — Group templates and the instance Git repository 131
Chapter XXVIII — Versioning and diff 135
Chapter XXIX — Scoring 139

Partie 5 — The browser extension

Chapter XXX — Anatomy of the floating panel 145
Chapter XXXI — Logging in from the extension 150
Chapter XXXII — The Templates view 154
Chapter XXXIII — The Profiles screen 158
Chapter XXXIV — Per-platform behavior 162
Chapter XXXV — Stash, substitution, parameter sets at injection time 166

Partie 6 — Customization

Chapter XXXVI — Themes 173
Chapter XXXVII — UI presets 177
Chapter XXXVIII — The features panel 182
Chapter XXXIX — Contextual tips 187
Chapter XL — Language and keyboard shortcuts 192
Chapter XLI — System variables for templates 197

Partie 7 — Administration

Chapter XLII — The admin dashboard 205
Chapter XLIII — User management 209
Chapter XLIV — License management 212
Chapter XLV — Analytics and event log 215

Chapter XLVI — Audit and alerts 219
Chapter XLVII — The diagnostics page 224

Partie 8 — Operations

Chapter XLVIII — The `izibuddy.ini` configuration file 231
Chapter XLIX — Logging 240
Chapter L — Backups 245
Chapter LI — The REST API 250
Chapter LII — Deploying behind a reverse proxy..... 256
Chapter LIII — Upgrading IziBuddy 262

Partie 9 — Connectors

Chapter LIV — Connectors — overview..... 269
Chapter LV — Database connectors..... 274
Chapter LVI — Object storage: cloud backends and attachments 280
Chapter LVII — AI-platform connectors..... 284
Chapter LVIII — E-mail / SMTP connector 289
Chapter LIX — Git remote connector 294
Chapter LX — Webhook connector 301
Chapter LXI — Lemon Squeezy integration..... 305

Appendices

Appendix A — Appendix ..... A. — Glossary
Appendix B — Appendix ..... B. — Icon legend
Appendix C — Appendix .... C. — Keyboard shortcuts — reference card
Appendix D — Appendix ..... D. — Error catalog
Appendix E — Appendix ..... E. — FAQ
Appendix F — Appendix ..... F. — Troubleshooting decision tree
Resources..... 344

Lexique 346

Index thématique 355

Index alphabétique 360

Partie 1

Foundations

Chapter I — What IziBuddy is

IziBuddy is a TEMPLATE[1]-management platform for the five major AI chatbots — Claude, ChatGPT, Grok, Gemini, and DeepSeek. You write your templates once, in a central DASHBOARD, and the BROWSER EXTENSION injects them on demand into whichever chatbot's editor you happen to be in. It is an answer to the daily friction of copy-paste and re-typing : your templates live in one place, versioned, shared, and organized ; the extension does the rest.

1.1 — The two halves

IziBuddy is made of two pieces of software that work together but can also live apart.

The dashboard is a web application built on FastAPI (server side) and vanilla JavaScript (client side). This is where you create, edit, organize, and share templates. You install it locally, on a team server, or use it through a hosted instance. Source : `src/main.py`, `src/api/`, `src/static/`.

The browser extension is a Manifest V3 add-on available for Chrome, Edge, and Chromium-compatible browsers. It injects a FLOATING PANEL into the chatbot's page ; from that panel you search for a prompt, fill in its parameters, and push it into the chatbot's editor. Source : `src/extension/manifest.json`, `src/extension/background.js`, `src/extension/content/`.

The two halves talk to each other through the dashboard's REST API. You can use the dashboard without the extension (to write, VERSION, export) ; you can use the extension without your own dashboard if an administrator has paired it with a hosted instance (your templates then live on the server, and the extension is the consumer).

1. Les termes en petites capitales renvoient au lexique.

1.2 — The five target platforms

The extension recognizes five conversational-AI platforms :

- **Claude** (claude.ai) — Anthropic's chatbot. Supported specifics : sidebar, *chain-of-thought* blocks, *Thought for Xs* badges. CONNECTOR : `src/extension/content/content-platform-claude.js` .
- **ChatGPT** (chat.openai.com, chatgpt.com) — OpenAI's chatbot. The extension detects the `data-chat-theme` attribute and adapts to the light/dark THEME. Connector : `content-platform-others.js` .
- **Grok** (x.com/i/grok) — xAI's chatbot. A few quirks in the message-input area.
- **Gemini** (aistudio.google.com) — Google's AI Studio. INJECTION targets AI Studio only, not the consumer gemini.google.com surface.
- **DeepSeek** (chat.deepseek.com) — DeepSeek's chatbot. No readable *Server-Sent Events* channel.

Adding a new platform means writing a connector — a content script that detects the editor, injects the prompt, and (where the platform allows) reads the response back. The connector model is detailed in Chapter LVII.

1.3 — The connector model in one paragraph

IziBuddy is built around two kinds of connectors : a single **database** backend (where templates and their metadata are stored), selected through the `[database]` SECTION of `izibuddy.ini` , and — on the extension side — one **AI-platform** connector per supported chatbot. There is no separate object-storage layer : attachments are handled by their own attachment backend (base64, sqlite, or filesystem). The default database backend is the filesystem, which is enough to get started ; production installs switch to a server database such as PostgreSQL. The full catalog — databases, attachment backends, AI platforms, plus SMTP and Git — is the subject of Part IX.

1.4 — What IziBuddy is not

IziBuddy manages your templates ; it does not run them. A few clarifications that head off common misunderstandings :

- **It is not an AI.** There is no language model inside. Rendering a prompt — substituting parameters, applying inheritance, expanding includes — is deterministic text processing, not text generation.
- **It is not a chatbot.** No conversation happens inside IziBuddy ; the extension pushes the rendered prompt into the chatbot's editor that you have open, and the conversation happens there.
- **It is not an API gateway.** IziBuddy does not contact any AI provider. If you want to orchestrate API calls from your own scripts, you can of course export a rendered prompt through IziBuddy's REST API and send it wherever you want — but routing the request to an AI provider is not IziBuddy's job.

This boundary is deliberate. It simplifies the security model (IziBuddy stores no AI API keys), it simplifies compliance (no user data leaves IziBuddy for an AI provider), and it simplifies the code (one role, done well).

1.5 — The POO²L language in a nutshell

IziBuddy templates can be written in POO²L — *Prompt Object-Oriented² Language* — a declarative language designed for reuse, inheritance, and composition. POO²L is not mandatory : a plain-text prompt (level L0 in POO²L terminology) is a valid POO²L prompt. You climb the sophistication ladder when the need arises : sections (L1), includes (L2), structured data and validation (L3), inheritance (L4), collections (L5).

This manual covers POO²L through the lens of the interface : how to declare a parameter from the parameter panel, how to wire two templates together with inheritance from the UI, and so on. Part III is dedicated to that surface.

1.6 — Where to go next

This manual is deliberately complete — from the first account you create to multi-tenant production — but it does not expect you to read it end to end. Depending on your role :

- **You are an end user** : start with Chapter V (*The first five minutes*) for a guided tour, then read Parts II, V, and VI as you need them.
- **You are a Group or platform administrator** : jump to Part VII (*Administration*) after going through the mental model in Chapter II.
- **You are an operator or integrator** : Part VIII (*Operations*) and Part IX (*Connectors*) are written for you ; Chapter III (*Installing the dashboard*) is your practical entry point.
- **You want to understand templates in depth** : Part III (*Authoring with POO²L*).

The next chapter lays out the underlying mental model — the five domain objects and their relationships — which makes the rest easier to navigate.

Templates and document nature

A **template** is the common reusable entity. Its **Nature** is `prompt` , `skill` or a custom identifier such as `mod` or `acme.review` . Create **New prompt** or **New skill**, then edit Nature in Properties when needed. Changing nature preserves the UUID, references, parameters and history. The leaf template owns its type ; includes never change it. Mixed inheritance can be allowed with warnings or denied.

```
@@@template review_assistant
@@@type skill
@@@description Reusable review instructions
```

```
@@@section rules
Separate observations from assumptions.
```

Use `type:acme.review` to search a custom nature. The **Example** flag replaces the old Template flag; `flag:example` finds these starting examples. `flag:draft` continues to find private drafts. A prompt copies the rendered text, a skill exports SKILL.md and resources, and other types export composed text without claiming a specialized package format.

Chapter II — The mental model

Before opening the dashboard, it is worth nailing down the vocabulary. IziBuddy manipulates five domain objects — prompt, CONTAINER, TAG, group, version — and these five are enough to describe almost everything you will do afterward. The rest of the interface is just staging for these objects and their relationships.

2.1 — The five domain objects

The **prompt** is the unit of work: a name, a description, a body (plain text or POO²L), and a set of metadata (markers, score, author, creation date). It is what the extension pushes into the chatbot's editor, after rendering.

The **container** is the object that groups templates. Two variants exist: the **FOLDER** (`kind="folder"`), purely organizational, and the **COLLECTION** (`kind="collection"`), which is a *class* of templates in the POO²L L5 sense — it can be inherited from and it can host virtual templates. A prompt inside a collection can reach its inherited body through the `parent` reference (prompt axis) and reach members of a containing collection through qualified includes. Chapter XI covers the distinction and how to promote a folder to a collection.

The **tag** is a cross-cutting label. A prompt can carry several tags; a tag cuts across containers. The tag hierarchy (FEATURE `tagsHierarchy`) lets you organize tags as a tree — for example `client/acme/legal` — without breaking compatibility with flat tags. Chapter X is dedicated to tags.

The **group** is a multi-tenant *scope*. A prompt lives either in a user's personal scope or in a group's scope. A group has two membership levels — plain **members** and **group admins** — where

admins manage the group's membership and settings. SHARING between users and collaboration within a group are the subject of Part IV.

The **version** is a snapshot of the prompt at a given moment. Each save can produce a new version, visible in the prompt's timeline and restorable. Chapter XXVIII covers versioning and diff.

2.2 — The relationship diagram

A prompt belongs to **exactly one** container (never zero, never two). It carries **zero to N tags** independently of its container. It lives in **one and only one scope** — personal or group — and does not change scope on its own (moving between scopes exists but is an explicit action). It accumulates **zero to N versions** over time.

Containers form a tree (one parent, many children). Tags can also form a tree when the hierarchy is enabled. Groups can themselves contain sub-groups. Versions are chained in chronological order.

In POO²L, a collection can **inherit** from another collection, and a prompt can **include** other templates. These relationships add a graph on top of the container tree ; they are described in Chapter XX. Source-of-truth files : `src/core/tag_based_templates_manager.py`, `src/config/template-structure.yaml`.

2.3 — Sharing vs duplicating

Two verbs that the rest of this manual uses heavily, and that do not mean the same thing.

Sharing a prompt means granting read (and possibly use through the extension) access to another user or to a group, with no transfer of ownership. The prompt still lives in exactly one place ; the recipient sees it but does not edit it. If the author modifies the prompt, the recipient sees the modification.

Duplicating a prompt produces an independent copy in the recipient's scope. From that point on, the two copies live separate lives : editing one does not affect the other. Duplication is the natural tool when you want to « adopt » a shared prompt to customize it.

Chapter XXIV spells out the practical consequences (notifications, versioning, deletion) and supplies a decision matrix for when the choice is not obvious.

2.4 — The lifecycle of a prompt

A prompt typically goes through four states — *draft*, *published*, *in flight*, *deprecated* — that are not exclusive labels but a combination of markers and filters.

You start by creating a prompt as a *draft* : it usually carries the **WIP** MARKER (work in progress) and may be **HIDDEN** so it does not clutter other users' lists. When it is ready, you drop the WIP and optionally tag it **IMPORTANT** ⭐ to surface it.

Through its life, the prompt may be LOCKED (**Locked** 🔒) to prevent accidental edits, or tagged as a **TEMPLATE** — that is, usable as a base class by other templates in POO²L.

When it becomes obsolete, you mark it **DISABLED** : it disappears from the default lists but stays visible through the dedicated FILTER, allowing archiving without deletion. Trashing (SOFT DELETE) and purging (hard delete) are covered in Chapter VII.

The six markers and their interactions with filters are the subject of Chapter VIII.

2.5 — Why this anatomy

The five-object model is not a flourish. It separates three concerns that competing tools often blend : *organization* (containers and tags), *collaboration* (groups and sharing), and *evolution* (versions

and lifecycle markers). By keeping them apart, IziBuddy lets you reorganize without losing history, share without losing control, and mark the maturity of a prompt without having to move it.

The rest of the manual leans on this vocabulary continuously. When you read « a locked group prompt », you now know it is a prompt living in a group's scope, carrying the Locked marker, hence read-only until the lock is removed ; everything else falls out from that.

2.6 — Where to go next

If you are eager to use the application, skip ahead to Chapter V (*The first five minutes*). If you are an operator or integrator, Chapter III (*Installing the dashboard*) is the next step. If you want to understand what templates can really do, Part III (*Authoring with POO²L*) picks up after the basic manipulation in Part II.

Chapter III — Installing the dashboard

This chapter covers the first startup of the dashboard, from the « hosted or self-hosted » decision to the first logged-in user. The load-bearing choices (database, object storage, TLS hardening) are introduced here and pushed forward to Part IX and Part VIII for the details.

3.1 — Two paths : hosted or self-hosted

If someone — your administrator, your team, your software vendor — has given you a URL and an account, you are on a **hosted** instance. Skip straight to Chapter V : your dashboard is already installed, all that is left is to populate it. Installation, upgrades, and backups are handled elsewhere.

If you want (or have to) host IziBuddy yourself, you are on the **self-hosted** path. The rest of this chapter is for you. Depending on the size of your installation, it may be either a Python process on your own machine (single-user mode, a few minutes) or a service behind a TLS Reverse proxy with external database and object storage (team or production mode, see Chapter LII).

3.2 — Prerequisites

Whichever target you are aiming for, you will need :

- **Python** 3.10 or higher. Older versions are missing annotations used by the code.
- **Disk space** : count a few hundred megabytes for the code and dependencies, plus the size of your templates (very modest in practice — a prompt is a few kilobytes).
- **Network** : one free TCP port (8765 by default) for the web server. In production, that port is exposed through a reverse proxy ; locally, you only need to bind it on `localhost`.

For a team install, add : a TLS certificate (Let's Encrypt is fine), an SMTP service for invitations and password resets, and — if you expect more than a handful of users — a server database such as PostgreSQL. See below.

3.3 — Picking a database backend

At install time, IziBuddy uses a single **database backend** to store your templates and their metadata (name, description, tags, OWNER, content). The default backend is the filesystem : everything lives in `~/.izibuddy/`, which is the right choice to get started. Attachments have their own attachment backend — base64, sqlite, or filesystem — selected independently ; there is no separate object-storage layer.

For a production install, the usual switch is a server database such as PostgreSQL, for durability and indexes. The full connector list — the supported databases and the attachment backends — is laid out in Chapter LV and Chapter LVI. You can change your mind later through the migration utility described in Chapter LV.

The choice materializes in `izibuddy.ini` (Chapter XLVIII) under the `[database]` section, where the `active = ...` key names the backend in use. You edit the config and restart the server ; the next sections walk through the practical sequence.

3.4 — First install, step by step

A minimal self-hosted install comes down to four steps :

1. **Get the code** — `git clone` the IziBuddy repository, or unzip a release archive.
2. **Install dependencies** — `pip install -r requirements.txt` in a virtual environment.
3. **Start the server** — run the `./izibuddy` launcher script. On first run, IziBuddy creates its `~/.izibuddy/` data tree and an `izibuddy.ini` (in the working directory) with defaults.

4. **Open the dashboard** — point your browser at `http://localhost:8765/`. The home page offers to create an account.

Server-side orchestration goes through FastAPI startup handlers (`@app.on_event`), which start the managers, load the configuration, and register the routers. Reference source : `src/main.py` and `src/config/`.

3.5 — The first user

The first registered account is administrator. There is no « root » password to invent : the first user who goes through the signup form gets administrator rights automatically and gains access to the admin dashboard (Part VII). Choose a strong username and password — you will manage every other account from this first one.

Subsequent accounts, created through the same form, are ordinary users. The administrator can promote any of them to admin from the user management screen (Chapter XLIII), and can disable self-signup if the install targets a closed team.

3.6 — TLS and hardening

On your own machine, on `http://localhost`, you do not need TLS. As soon as the dashboard is exposed on a shared network — *a fortiori* on the Internet — it must be served over HTTPS. IziBuddy does not ship its own TLS terminator : the recommended practice is to put a reverse proxy (nginx, Caddy, Traefik) in front of the Python server and let the proxy handle the certificate.

The full list of headers to forward, the bootstrap of the first user in a hardened environment, and the production checklist live in Chapter LII. If you are heading for production, read that chapter **before** opening port 443.

3.7 — Desktop mode

IziBuddy can also run in desktop mode — a standalone binary with its own embedded single-user database. This mode is documented elsewhere (`docs/plans/ROADMAP-DESKTOP.md`) and uses the `single_user_database` connector. It has no first-user-to-promote : the system user is the only account.

3.8 — What to do next

With the dashboard in place, two directions :

- If you want to test the full pipeline right away, jump to Chapter IV (*Installing the browser extension*) and then Chapter V (*The first five minutes*).
- If you are installing for a team, read Chapter XLVIII (the `izi-buddy.ini` file), then Chapter LII (reverse proxy), then come back to Part VII for user creation and group configuration.

Chapter IV — Installing the browser extension

The extension is IziBuddy's operational arm : it is the piece that injects your templates into Claude, ChatGPT, Grok, Gemini, and DeepSeek. Dashboard and extension can live apart, but the pair is what makes the tool useful day to day. This chapter covers installation, permissions, and pairing with your dashboard instance.

4.1 — Where to download the extension

The extension is a **Manifest V3** add-on for Chromium-compatible browsers : Chrome, Edge, Brave, Opera, Vivaldi. Three install paths coexist.

The official path, coming soon, goes through the *Chrome Web Store*. Once available, it is the simplest : one click, automatic updates, nothing to do.

The **direct-download** path provides a `.zip` file to load as an unpacked extension. You extract it into a folder, enable *Developer mode* in `chrome://extensions`, then *Load unpacked* and point at the folder. The extension appears in the toolbar, ready to use.

The **build-from-sources** path is meant for developers and integrators : you check out `src/extension/`, run the provided build script, and obtain the same `.zip`. It is also the path used for test builds while adding a new AI-platform connector.

The extension source lives under `src/extension/`. The reference manifest is `src/extension/manifest.json` ; the background service worker is `src/extension/background.js`.

4.2 — Enabling the extension on each platform

Once installed, the extension stays dormant until you open one of the recognized chatbots. The supported hosts — declared in the manifest — are :

- `claude.ai`
- `chat.openai.com`, `chatgpt.com`
- `x.com/i/grok`
- `aistudio.google.com`
- `chat.deepseek.com`

No per-platform action is required : opening the URL is enough, the corresponding content script injects itself, and the floating panel appears, folded into a corner of the page. Clicking it opens it. Chapter XXX describes the panel's anatomy ; Chapter XXXIV covers per-platform specifics.

If you want to add a new AI platform, that is the job of a dedicated connector — see Chapter LVII.

4.3 — Permissions, explained

The manifest asks for a small set of permissions. It is worth understanding them, especially in an enterprise setting where security reviews each add-on one by one.

`activeTab` : lets the extension interact with the tab that is active at the moment the user clicks the icon. Targeted, not global.

`storage` : lets the extension store preferences locally, the *Stash*, the chosen theme, the pairing configuration. No data leaves the machine through this permission.

`scripting` : required to inject the content script into the chatbot's page. This is what allows the floating panel to exist.

`tabs` : lets the extension know which tabs are open (URL, title) to decide where to activate.

`notifications` : used for unobtrusive feedback (injection success, pairing error). A system notification, not an intrusive popup.

The manifest declares broad `host_permissions` (`http://*/*` and `https://*/*`) so the content script can attach on the supported chatbot pages ; the extension nonetheless only activates its connectors on the platforms listed above.

4.4 — Pairing the extension with your dashboard

On first launch, the extension asks where to find your templates. Two pieces of information are enough :

1. **The dashboard URL.** For a local install, `http://localhost:8765` . For a hosted instance, the URL your administrator gave you.
2. **Your credentials** — either username + password, or an access token if your administrator has configured token-based authentication.

Pairing happens through the **Config** tab of the floating panel. The form is minimal : URL, credentials, *Connect*. The extension stores a session token and then queries the dashboard's REST API to retrieve your prompt list. See `src/extension/tabs/tab-config/` .

The *Remember me* option (`remember_me`), when enabled, keeps the session across browser restarts. Fine on your personal machine, to be avoided on a shared workstation.

If your dashboard changes URL (move, production rollout), all you need is to go back to the **Config** tab and re-pair. No uninstall required.

4.5 — The link from the dashboard

On the dashboard side, the `chromeExtensionLink` feature adds a shortcut to the extension's installation in the UI. It is purely cosmetic (a link) but practical : a user who has just created their dash-

board account immediately sees how to continue the install on the extension side. The feature is included in every PRESET, from Simple upward.

4.6 — Quick troubleshooting

Three classic failures at first pairing.

The floating panel does not appear. Check that the extension is enabled for the platform in question (`chrome://extensions`, *Details*, *Site access*). On some ChatGPT layouts, a page refresh after installation is required for the content script to inject.

Pairing fails with a network error. Check the dashboard URL (with or without `https://`), that the server is running (`/api/status/` responds), and that no chatbot CSP is blocking outbound calls. The CSP on Claude and ChatGPT is restrictive ; the extension works around this by routing through its service worker.

Injection does not work but the panel shows up. This is typically a platform-connector issue — the chatbot's editor has changed structure. Check the extension version ; see Chapter XXXIV for per-platform diagnostics.

Chapter V — The first five minutes

This chapter is a guided walkthrough. You start with a freshly created account, you write your first prompt, you send it into ChatGPT through the extension. Five minutes, watch in hand. By the end, you will know how to do what nine users out of ten do 90% of the time — the rest of the manual is just elaboration on this foundation.

5.1 — Create an account and land on the dashboard

You open the dashboard URL — the one your administrator gave you, or `http://localhost:8765` if you followed Chapter III. The signup page shows up.

You fill in a username, a password, an email address (used for invitations and reminders). You submit. You are signed in.

The screen you land on is the **dashboard** : three main zones — the top navigation bar, the templates list in the middle, and the filters and side panel on the right. On first startup, the list is empty. That is our starting point.

The main page's code lives in `src/static/index.html` and its associated scripts.

5.2 — Write your first prompt

Click **New ▾** (the menu button in the toolbar) and pick *New prompt*. The editor opens in the right-hand panel. Three fields are asked for : a *name*, a short *description*, and the *content* of the prompt. To start, keep things minimal :

- **Name** : `Polite rewording`
- **Description** : `Rewords a message while staying courteous`
- **Content** :

`Reword the following message while staying polite and professional, without changing the substance:`

Click **Save**. The prompt appears in the list on the left. It is your first domain object. It lives in the *user* scope (your personal space), in the root container, with no tags, no markers, at version 0.1.

5.3 — Open ChatGPT and bring up the panel

You leave the dashboard open, and in a new tab you go to `chatgpt.com`. You sign in to your ChatGPT account if needed.

Once the page is loaded, look at the top-right corner : a small rectangular mini-panel has appeared. That is the **IziBuddy extension** injecting itself. Click it. The floating panel unfolds.

If nothing shows up, go back to Chapter IV — the extension is probably not paired yet. The panel's *Config* tab is waiting for your dashboard URL and credentials.

5.4 — Inject the prompt

In the panel's **Templates** tab, you see the list of your dashboard templates (at this stage : one, your *Polite rewording*). You click it.

The extension copies it into ChatGPT's editor — the input area at the bottom of the page fills up with the text of your prompt. You add the message you want reworded by hand, after it, and then you submit. ChatGPT replies.

You have just closed the loop end to end : dashboard → extension → chatbot. No copy-paste, no mental duplication of the prompt, no risk of forgetting the latest revision.

5.5 — What just happened

A few seconds to retrace the steps that happened without being named.

Authentication. The panel queried your dashboard's REST API with the pairing token, and received your prompt list in response. No traffic went to OpenAI. See Chapter XXXI for details.

Platform detection. The extension's content script recognized `chatgpt.com`, loaded the ChatGPT connector (`content-platform-others.js`), located the message editor in the DOM, and prepared the injection. Chapter XXXIV covers the mechanics.

Template rendering. Because our prompt has no variables, no parameters, no includes, the rendered content equals the source. For a richer POO²L prompt, rendering would have substituted parameters, expanded includes, applied inheritance — see Chapter XXII.

Versioning. When you clicked *Save*, version 0.1 was archived. Any later edit will create version 0.2 without erasing anything. See Chapter XXVIII.

5.6 — Quick tour of the interface

Now that the loop works, take a few seconds to spot the main pieces of the dashboard.

The **top bar** carries the scope selector (my space, my groups), the search field, and the user menu (preferences, UI preset, sign out).

The **left side panel** lists containers (folders, collections), tags, and quick filters (markers *Important*, *WIP*, *Locked*, and so on).

The **middle panel** shows the templates list. Two views — a tile grid and a detailed table — selectable in the panel's top-right corner. Chapter XIII covers them.

The **right panel**, when a prompt is selected, shows its content, its editor, its POO²L parameters, its versions, its attachments. When the SPLIT PANE (feature `viewSplitPane`) is on, a second prompt can be opened simultaneously next to it. Handy for comparing or for drag-and-drop.

5.7 — What's next

You now have the bare minimum. The rest of the manual goes deeper in each direction, depending on your role.

- To author richer templates — parameters, sections, inheritance, attachments — Part III is your destination.
- To organize, find, and find fast — Part II (Chapters IX to XIV) covers categories, tags, containers, search, views, bulk operations.
- To share with a team — Part IV (Chapters XXIV to XXIX) covers group scopes, invitations, Git repositories, collaborative versioning.
- To push the extension further — *Stash*, theme profiles, parameter sets at injection time — Part V (Chapters XXX to XXXV).

But first, get comfortable with the five domain objects : prompt, container, tag, group, version. Everything else builds on top of them.

Partie 2

Working with templates

Chapter VI — Anatomy of a prompt

A prompt, in IziBuddy, is more than a block of text. It is a structured object, with a visible part you edit and a hidden part the system maintains for you. Understanding both layers makes everything that follows — POO²L parameters, inheritance, versioning, sharing — much easier to handle.

6.1 — The visible fields

When you open a prompt's editor, you see four main things.

The **name** identifies the prompt in the list. It is also the basis for prefix search. There is no strict technical constraint beyond uniqueness within its container, but short, descriptive names (two to four words) make life easier as the library grows.

The **description** is one sentence that says what the prompt is for. It shows up in the *tiles* view under the name, in the *list* view beside it, and in the extension panel while searching. A well-crafted description — action verb plus complement — pays off the moment you, or a teammate, are hunting for the right prompt among hundreds.

The **tags** are thematic labels. A prompt can carry as many as you want ; the first tag in the list (the *primary tag*) plays a role in certain contexts — restoring from trash, for example. Tags drive filtering, sorting, hierarchical navigation when `tagsHierarchy` is on. See Chapter X.

The **container** is the prompt's canonical location — a folder or a collection. A prompt belongs to exactly one container ; to « file it elsewhere », you move it. See Chapter XI.

The **content**, finally, is the body of the prompt. It is text. If it contains POO²L constructs — `@@@=...` parameters, sections, includes, inheritance — they will be processed at render time. If it does not, the content is sent as-is. POO²L is strictly opt-in.

6.2 — The hidden state

Beyond those four fields, each prompt carries metadata maintained automatically. You do not edit it directly, but it exists and it matters.

The **markers** are the six lifecycle labels : *Important* ⭐, *Locked* 🔒, *WIP*, *Disabled*, *Template*, *Hidden*. They are toggled from the prompt's context menu. Chapter VIII covers each one in detail.

Whether a prompt sits in the active library or in the **trash** is a separate, binary piece of state. There is no derived « status » field — what you filter on are the individual markers plus trash presence, not a synthesized label.

The **version** is tracked as an `x.y` pair, saved on each meaningful change. Version history remains accessible under *Versions* in the right panel. See Chapter XXVIII.

The **author** and **group ownership** identify who created the prompt and which scope it lives in (personal user or group). Within a group, the distinction that drives permissions is membership : a plain member and a group admin, where admins manage the group itself. A *Locked* prompt is read-only for everyone until the lock is removed.

The **quality score**, when the `viewScoring` feature is on, is a value out of 10 **computed** by the analysis engine from the prompt's structure — it is not a user rating. A prompt can opt out of SCORING via the **Note** flag (stored as `no_scoring`); the `filterNoScoring` filter surfaces the templates carrying that flag. See Chapter XXIX.

The reference code for all this metadata is `src/core/tag_based_templates_manager.py`, which exposes the CRUD operations and maintains invariants. The API surfaces it in `src/api/templates.py`.

6.3 — Source vs rendered

A distinction that looks subtle and is not. A prompt has *two* textual representations : its **source** — what you edit — and its **rendered** form — what is sent to the chatbot.

For a plain-text prompt, source and rendered are identical. For a POO²L prompt, they can differ a lot. Rendering substitutes parameters, expands includes, applies inheritance, and replaces the system variables `@@@=DATE`, `@@@=USER`, and friends. The source does not change during this processing.

This duality is one of IziBuddy's core purposes : you maintain the source — DRY, factored, inherited — and the system produces the rendered text for you at injection time. History tracks the source, the editable version is the source, full-text search is on the source. The rendered form is just the final product.

Rendering is delegated to the POO²L engine (the Rust `pool-core`, via its bindings) through `src/core/pool_engine.py`. It is deterministic — same inputs, same output — which simplifies security (no surprising output) and compliance (no data leaks to an AI provider).

Chapter XXII goes into the details of rendering, the *preview* in the UI, and the system variables it substitutes.

6.4 — Visible on the API

The REST API exposes this whole structure. A call to `GET /api/templates/get/{id}` returns a JSON object with :

- `name`, `description`, `content`, `tags`, and the container the prompt lives in (the visible fields above),

- the marker flags (`important`, `locked`, `wip`, `disabled`, `is_example`, and so on), the version, the author, and the group it belongs to (the metadata),
- optionally the *attachments* and the POO²L parameters extracted by the parser.

You do not have to manipulate this JSON by hand to use IziBuddy — the UI handles it — but it is useful to know it is there when you want to write a script that bulk-ingests or exports templates (Chapter LI, Part VIII).

6.5 — A word on POO²L parameters and constants

When a prompt carries POO²L parameters — declared with the `@@@=...` syntax we will see again in Chapters XVII and XXII — they appear in the editor as a separate list, below the content : parameter name, type, constraint, default value. You can declare them from this UI without touching the source text, or you can write them directly into the content — both paths produce the same object.

Similarly, **constants** (Chapter XVIII) are stored apart from the content, in a dedicated area. They are never asked for at injection time : they are substituted at render time, period.

Chapter VII — Creating, editing, duplicating, trashing, restoring, purging

This chapter covers the routine operations on a prompt. Six verbs, all reachable from the UI, all mirrored by clearly named REST endpoints. None of them require POO²L knowledge — these are the basic gestures of document management, transposed to the prompt domain.

7.1 — Creating a prompt

The **New prompt** button opens the editor in the right-hand panel. Three minimum pieces of information are asked for : a name, a description, content. Tags and container can stay empty at creation time — the prompt lands in the root container of your current scope, with no tags.

Once saved, the prompt is immediately available in the list and in the extension panel. No explicit publication step : there is no *draft → published* transition in IziBuddy. If you want to signal that a prompt is still work in progress, put the *WIP* marker on it (Chapter VIII) ; the dedicated filter then lets your collaborators ignore it or keep an eye on it.

On the API side : `POST /api/templates/create`. On the UI side : `src/static/index-templates-crud.js`.

7.2 — The Smart editor

The editor — *smart editor* in the code — brings several aids when the content is rich : POO²L syntax highlighting, autocomplete on parameter and prompt names, inline validation (unresolved reference

in red, deprecated include in yellow). It also hosts the editor *Notes* (feature `editorNotes`) and the *attachments* (feature `editorAttachments`).

Chapter XVI dives into all its features. If you write plain text, the only thing to know is : the *Save* button (keyboard shortcut Ctrl+S) saves and creates a version if the content has changed.

7.3 — Editing, save semantics

Clicking a prompt in the list opens it in the right-hand panel. Any change — name, description, content, tags, container, markers — is tracked by the editor. A *modified* indicator appears as soon as you change anything ; clicking elsewhere without saving templates you to save.

Each save can create a new version. The rule is not a size threshold : **any** change to the content produces a new version, but saves are **deduplicated by checksum** — saving content identical to the current version creates no new version. See Chapter XXVIII for the exact rules.

Endpoint : `PUT /api/templates/update/{id}`. To change *multiple* templates in a single request (Bulk operations), see `PUT /api/templates/update/bulk` and Chapter XIV.

7.4 — Duplicating a prompt

Duplication creates an independent copy. You start from an existing prompt — yours, or one shared by a teammate — and you obtain a new object that you own, in your scope, with its own life cycle.

Duplication carries only the current content : the copy starts fresh, with no history. The two copies are independent from that point on.

The endpoint — `POST /api/templates/duplicate/{id}` — produces a copy in the caller's own space. It is the classic « I saw your prompt, I want mine starting from there » gesture.

Once duplicated, the new prompt has no link to the original. If the original author edits their version, yours does not move, and vice versa. This is the essential difference with *sharing*, where the prompt stays under its author's control (Chapter XXIV).

7.5 — Sending to trash (soft delete)

Deleting a prompt from the UI does not destroy it immediately. It goes to the **trash** — *trash* in the code — from where it can be restored as long as it has not been purged. This is the *soft* delete, as opposed to the *hard* delete covered below.

Trash preserves everything : content, metadata, versions, tags, attachments. It is invisible in the main list by default ; you reach it through the dedicated filter or the *Trash* view in the menu.

The endpoints :

- `GET /api/templates/trash/list` — list trashed templates in the current scope.
- `GET /api/templates/trash/count` — quick count for the UI badge.
- `DELETE /api/templates/delete/{id}` — send a prompt to trash.

Trash is per-user (or per-group for group templates) : another user's trash is none of your business.

7.6 — Restoring a prompt

Restoring brings the prompt back identical : same content, same versions, same metadata. Endpoint : `POST /api/templates/restore/{id}`.

One detail is worth a mention. If meanwhile you have deleted a tag that the prompt carried — or if you restore into another scope where that tag does not exist — you can provide a *new primary tag*

(`new_primary_tag` in the request) to replace the old one. The UI offers it automatically when needed ; nothing to do by hand in the common case.

Restoration and trash form an inseparable pair : restoration is what makes soft delete really soft.

7.7 — Purging (hard delete)

Purging deletes *everything* : prompt, versions, attachments, scores. No going back — the trash has no trash of its own. Endpoint : `DELETE /api/templates/purge/{id}` .

The UI confirms before every PURGE. It is an operation you do not trigger by accident.

Two variants exist for mass operations :

- `DELETE /api/templates/purge-selection` — purges a set of checked templates in a single request. See Chapter XIV.
- `DELETE /api/templates/purge-all` — empties the trash entirely, with a single end-of-cycle Git commit (instead of N commits) if a Git repository is configured. Useful for big-cleanup sweeps.

7.8 — Bulk delete, bulk restore

Selecting multiple templates in the list — Ctrl+click, Shift+click, mouse rubber band (feature `actionRubberBand`) — unlocks mass actions : delete, restore, move, tag several templates in one operation. Chapter XIV details the selection experience and the related shortcuts.

Bulk operations matter because they change the tool’s scale. Doing 200 micro-actions with the mouse is tedious ; a rectangular selection followed by *Delete* is instantaneous. The system maintains integrity — consistent trash, preserved versions — no matter the count.

Skills and local drafts

A prepared prompt defines a repeatable task. A skill provides reusable instructions for an agent. Both use the same POO²L composition : parameters, includes, inheritance, collections and attachments.

Create and convert

Choose **New skill** and enter a name, logical ID, description and instructions. The **Document type** property converts prompts to skills and back under the same UUID. References keep their target ; descendants are not converted. IziBuddy stores identity in properties ; exported POO²L declares `@@@template` + `@@@type skill`. Parameters are filled when generating the file, not automatically every time an agent uses it.

Composition and policy

A skill can inherit a prompt and vice versa. The requested document's declaration determines its nature ; includes never change it. Both can belong to the same collection. Administrators choose **Allow** (default, with warnings) or **Deny** mixed inheritance in document policy. Deny checks every reached inheritance edge, including collection-member overrides, but does not forbid a mixed include by itself. Existing data remain available for correction. The demo's setting belongs only to that browser.

Work locally

Choose **Local draft** in properties or the Save menu. The UUID and last published state stay intact. IndexedDB stores the draft and its attachments for your account, application address and browser, with no shared version, commit or push. The existing `wip` flag is a work-in-progress label ; it does not provide draft isolation.

Search `flag:draft`, optionally with `type:skill`, to find all your drafts, including unpublished new documents. **Use my local drafts** in Preview overlays your draft parents and includes. The result says **Local preview — unpublished**. Draft verification also runs locally; missing input values must still be supplied.

Drafts do not synchronize across devices or browsers. Download a JSON backup and use **Restore a draft backup** to recover the same identity. Back up before clearing browser data. **Discard** keeps the published version.

Publish

Publish checks rights, identity, policy, published dependencies and the starting revision. It never publishes dependency drafts automatically. A conflict shows local and published states. Accepting a new base keeps your content without merging; edit if needed, then explicitly publish again. Errors retain the draft; retrying synchronization does not duplicate an identical version.

Publishing a shared component may change its consumers' next instantiations; drafts do not. Historical direct save remains available. Once a draft exists, Save stays local until Publish.

Export

Instantiate the skill, fill requested inputs and choose **Export skill**. Save a standalone `SKILL.md` inside a folder named after the skill, or extract the ZIP, which already contains that folder and its resources. Names are derived from logical IDs, using lowercase letters and hyphens. A property description is required. Existing body YAML frontmatter, unsafe paths, resource collisions and missing bytes are rejected.

The extension opens skills in the web editor. API/CLI `target=skill` returns text and relative files encoded as base64. Export does not install or execute anything in an agent. When a source is published again, previously exported files must be regenerated and replaced manually in the target agent.

Chapter VIII — Markers and lifecycle states

A prompt does not have a linear life — it gets drafted, tested, published, highlighted, unpublished, archived. IziBuddy does not model each of these states as a formal transition ; it provides instead **six markers** that you combine to fit your way of working, and a panel of **filters** that respond in mirror. This duo covers all the usual life cycles without imposing one of its own.

8.1 — The six markers

Important ⭐. The pin. You apply it to the templates you use every day, or that you want pushed to the top of the list. No functional effect ; only visual emphasis and a dedicated filter (`filterImportant`).

Locked 🔒. Locked. As long as a prompt carries this marker, it is read-only : its content, name, tags, and container cannot be changed. To edit again, you remove the lock. It is the anti-accident mechanism for critical templates — typically templates shared in a team that nobody should touch without prior agreement. Filter : `filterLocked` .

WIP. *Work in progress*. You put it on templates still being drafted, where you do not want colleagues to treat them as final. No locking effect — full edit access is preserved — just a visual signal and a filter (`filterWip`).

Disabled. Disabled. It is the equivalent of « obsolete but kept around » — a visual signal that a prompt is out of favour, without deleting it. Unlike *Hidden*, a *Disabled* prompt still shows in the default list ; the `filterDisabled` filter lets you show only the disabled templates, or exclude them, for cleanup or reactivation work.

Template. Template. Indicates that a prompt is intended to serve as a *base* for other templates — typically a POO²L class that other templates inherit from. Filter : `filterExample`. Not directly tied to the notion of *collection* (Chapter XI), even though the two often overlap in practice.

Hidden. Hidden. Disappears from default views, even more deeply than *Disabled* — a *Hidden* prompt does not appear in full-text search unless you explicitly turn on the `filterHidden` filter. Useful for templates internal to a system, throwaway drafts, technical helpers.

8.2 — How they are toggled

Each marker is set or cleared from the prompt's context menu — right-click in the list, or the dedicated button in the edit panel's toolbar. Several markers can coexist on the same prompt without contradiction : *Locked* + *Important* is common (« my key prompt, don't touch ») ; *WIP* + *Template* is common too (« base class still being written »).

A few combinations are logically exclusive — *Disabled* and *Hidden* partly overlap in use, for instance — but the system enforces no strict exclusion rule. It is up to you to set your team's convention.

8.3 — The role of the mirror filters

Ordinary attribute filters cycle through neutral, show only, and exclude when exclusions are enabled. **Hidden is the exception** : normal excludes hidden items by default, green selects hidden items only, and a grey ghost shows both hidden and non-hidden items. Grey means the visibility restriction is off, not that the button is unavailable. Hidden always has three states.

The first ones match the markers above : `filterImportant`, `filterLocked`, `filterWip`, `filterNoScoring` (templates carrying the `no_scoring` flag, see Chapter XXIX), `filterHidden`, `filterDisabled`, `filterExample`.

The next three — `filterFolder`, `filterCollection`, `filterTemplate` — filter by *kind* of listed object : show only folders, only collections, only templates. Handy when your tree mixes everything.

`filterClearAll` is the broom button : one click resets all filters. Worth keeping accessible when you work with several filters active in parallel.

`filtersExclude`, finally, is the toggle that **unlocks the third tristate state**. With it off, each ordinary marker filter cycles between two states only (neutral and *show only*) ; with it on, the cycle gains the *exclude* state, letting you turn a filter into a negative one (*non-Important*, *non-WIP*, and so on). It is what makes exclusion possible without doubling the controls. See Chapter XII for typical uses.

8.4 — Useful combinations

A few patterns show up often in practice.

« **My must-haves** » : `filterImportant` on, everything else neutral. You get the short list of pinned templates. Combined with the *Simple* UI preset, this is the minimal interface for someone using ten to twenty templates.

« **Everything unfinished** » : `filterWip` in *show only* mode. Clear view of your draft queue. Combined with sort-by-modification-date, this is your daily working board.

« **Quarterly cleanup** » : `filterDisabled` in *show only*, possibly plus `filterNoScoring`. You see what nobody uses any more and what is marked as Notes — candidates for purge or review.

« **Step out of the locked pile** » : `filterLocked` in *exclude* mode. You focus on the templates still editable without any risk of touching the locked ones.

« My group's POO²L classes » : `filterExample` on, possibly with `filterCollection`. Catalog view of the inheritance bases available.

8.5 — Markers and trash presence

Two nearby notions not to confuse.

A **marker** is a discrete label set on purpose by a user. It can be removed at any time ; it does not affect the ability to edit (except *Locked*) or to delete.

There is **no derived « status » field** : IziBuddy does not synthesize a single active/draft/deprecated label. The overall state of a prompt is read directly off its markers plus whether it sits in the trash — which is exactly what the filters act on.

Trash presence is binary and orthogonal : a prompt is either in trash or not. Markers survive trashing (a *Locked* prompt that goes to trash comes back *Locked* on restore), but trash itself is not a marker.

8.6 — Practical notes

Avoid combining *Hidden* and *Disabled* without a reason : both do roughly the same thing and end up confusing the team. Pick one as your convention and stick with it.

Locked is powerful but heavy-handed when an admin sets it on a shared prompt : downstream editors see their edits refused. Talk to your team before locking a big batch.

Template is purely informational : it does not technically turn a prompt into a class in the POO²L sense. For a prompt to serve as an inheritance base, you declare it as such through the parent picker (Chapter XX) or by placing it in a *collection* (Chapter XI). *Template* is the visual hint that announces the intent.

Chapter IX — Categories

« Category » is the broad, primary classification of a prompt — the functional family it belongs to, above the finer thematic axes. In IziBuddy there is **no separate category object** : this classification role is realized through **tags**. What earlier versions called a category is, today, a tag — typically a top-level tag in the tag hierarchy. This chapter explains how tags fill the categorization role ; the full tag reference is Chapter X.

9.1 — The model

Since there is no dedicated category field, you get the « one broad family per prompt » effect by convention : reserve a small set of top-level tags — *Writing*, *Code*, *Analysis*, *Brainstorming*, whatever fits — and treat them as your categories. With the tag hierarchy on (`tagsHierarchy` , Chapter X), a top-level tag such as `code/` groups everything filed under it.

Unlike a rigid one-per-template category, a tag-based classification does not forbid a prompt from carrying two family tags — but the discipline of picking one primary family tag reproduces the old behaviour when you want it.

In practice, a family tag shows up in :

- the prompt’s thumbnail in the tile view, among its tags,
- the tag columns and chips in the table view,
- the detail panel on the right when a prompt is selected,
- the left side panel’s tag list, which lets you filter the list down to a single family.

The effect is organizational and navigational. Rendering, parameters, inheritance — none of it depends on tags. They are an organizational layer, not a semantic one.

9.2 — Create, rename, delete

Because a category is just a tag, all the operations are the tag operations of Chapter X. Creation is implicit : typing a family tag on a prompt creates it. Renaming a tag everywhere — so every prompt that carried the old family tag now shows the new one — is the *Rename tag everywhere* function, available when the `tagsAdmin` feature is on.

Deletion follows the tag rule, which is stricter than a category deletion used to be : **a tag can only be deleted once it is no longer used**. If templates still carry it, deletion is refused, and you must first remove the tag from those templates (or rename/merge it). There is no « reassign the orphans on delete » dialog — you clear the usage first, then delete.

9.3 — Family tag vs finer tags vs container

Two neighbouring axes remain, three once you keep the family/finer distinction in mind. The choice between them is the most common organizational question at the start. Here is a rule of thumb :

The **container** is the canonical location. You go there when you look for a prompt physically — like a folder in a file explorer. A prompt has exactly one container, no exception.

The **family tag** is the main classification — the broad functional family. *Writing*, *Code*, *Analysis* make good family tags. Keep them few and stable.

The **finer tags** are the multiple thematic axes. A prompt can carry several ; one can apply across families. *Urgent*, *EN*, *B2B*, *Target-SMB* are typical finer tags.

The ideal pattern : your container reflects a project organization (`Marketing/Newsletters` , `Dev/Unit-tests`), a family tag reflects a functional family (*Writing*, *Debug*), your other tags reflect orthogonal axes (language, criticality, target, tone).

9.4 — When not to add a family tag

Family tags are a lightweight convention ; they become heavy when they try to do the finer tags' job. If you catch yourself wanting several family tags on one prompt, what you really have is finer tags. If you create one family tag per project, what you need is a container (or a collection).

In practice : stay under ten family tags or so for a given scope. Beyond that, users no longer recognize them at a glance and the visual grouping dissolves. When your catalog crosses that threshold, arbitrate — either merge close family tags or push the granularity into finer tags.

9.5 — Family tags in a group scope

Tags are defined *per scope*. Your personal space has its tags ; each group has its own. When you switch scope (Chapter XXV), the tag list shows the matching set.

For a group, a group admin typically curates the family-tag catalogue — *Sales*, *Support*, *Product*, for example. Plain members tag and filter with it ; only a group admin can restructure the shared tags (rename everywhere, merge). This editorial discipline is what makes navigation predictable when joining a new shared space.

9.6 — A few proven patterns

For personal use, three to five family tags are usually enough : for example *Writing*, *Code*, *Analysis*, *Personal*, *Brainstorming*. Beyond that, the distinctions blur.

For a marketing team, family tags like *Briefs*, *Copywriting*, *Rewrites*, *Ideas*, *Audits* have worked well in practice.

For a development team, *Code*, *Tests*, *Doc*, *Debug*, *Refactor* cover most of the ground.

For a cross-cutting use, you can also skip the family-tag idea entirely and rely on flat tags alone. Nothing forces a primary classification — it is a convention you adopt only if it helps.

9.7 — The implementation side

In code, the old category machinery is now backed by tags. The front-end carries transition stubs (`index-tags.js` — the legacy `selectCategory` / `openCategoryEditModal` entry points redirect to the tag equivalents `selectTag` / `editTag`). There is no distinct `category` field on a prompt in `src/repositories/template_repository.py`, and no `/api/templates/categories` route: everything goes through the tag API (`src/api/tags.py`) and `PUT /api/templates/update/{id}` on the prompt's `tags`. The tag/container model and the trade-offs between the two are covered in Chapter XI.

Chapter X — Tags and the tag hierarchy

Tags are IziBuddy's most-used organizational tool. Flexible — a prompt can carry several, the same tag can apply to hundreds of templates — they are the main axis for search, filtering, and thematic navigation. This chapter covers both the flat and hierarchical flavors, and the admin operations on shared tags.

10.1 — Flat by default

At startup, tags are **flat**: a plain list of labels with no relationship between them. You create a tag by typing it into the prompt editor — either picking from the autocomplete dropdown or typing a new label, which is created on the fly.

A single prompt can carry as many tags as you want. There is no uniqueness constraint beyond the name: `B2B` and `B2C` can coexist without any relation. This is deliberate: flat tags leave the semantics entirely to the user.

For navigation, the dashboard's left side panel lists your tags in alphabetical order, with a usage counter on the right. Clicking a tag filters the center list to the templates that carry it — the basic gesture.

10.2 — Hierarchical, as an option

Beyond fifty tags or so, the flat list becomes uncomfortable. The `tagsHierarchy` feature lets you structure the list as a tree: `client/acme`, `client/beta`, `tech/python`, `tech/javascript`. The convention is slash as separator.

Turning on `tagsHierarchy` does not change the tags themselves — a tag is still a string. What changes is the display: the side panel folds into a tree, you expand `client/` to see its sub-tags. And cli-

cking a parent tag (`client/`) filters by *all* its sub-tags — handy when you want to see every prompt tied to a domain, regardless of the sub-segment.

The feature is part of the *Standard* preset in the registry — see `PRESETS` in `src/static/index-features-registry.js`. You can turn it on or off independently through the features panel (Chapter XXXVIII).

10.3 — Create, rename, delete

Creation is always implicit : typing a non-existent tag name in the editor creates it. No separate panel to navigate — creation is inlined in the tagging gesture itself.

Renaming is a bit more delicate : renaming `Client-Acme` to `client/acme` has to propagate to every prompt that carried the old name. That is the *Rename tag everywhere* function — available when the `tagsAdmin` feature is on, through the tag's context menu in the side panel.

Deletion is only allowed on an **unused** tag. A tag still carried by one or more templates cannot be deleted : the API refuses it (`DELETE /api/tags/{name}` returns an error when the usage count is above zero), and asks you to remove the tag from those templates first — or to rename or merge it into another tag. There is no « reassign the orphans on delete » step ; you clear the usage, then delete the now-empty tag.

The tags REST API lives in `src/api/tags.py` — about 700 lines of code covering CRUD, merge, and mass rename. The business manager is `src/core/tags_manager.py`. The front-end lives in `src/static/index-tags.js`.

10.4 — Click a tag to filter

The `clickTagToFilter` feature makes the tags displayed in thumbnails and the detail panel directly clickable : a click applies the tag as a filter in the center list. Ctrl+click *adds* the tag to the current selection (cumulative filter) ; Shift+click *excludes* the tag (negative filter).

It is a quality-of-life feature that saves a lot of time when navigating by tags. It is in the *Advanced* preset and above ; you can turn it off if you prefer to reserve tag clicks for editing.

10.5 — Merging two tags

A typical case : you have accidentally created `Client-Acme` and `client-acme` (one capitalized, one not). You want to keep only one. That is **merge** : every prompt carrying the old tag receives the new one, and the old one is deleted. It is atomic — all or nothing.

The operation is in the tag's context menu (`tagsAdmin` required). The UI asks for the target, confirms the operation, and applies. If some templates carried both tags simultaneously, the result is that they end up carrying only the merged tag (no DUPLICATE).

10.6 — Admin operations on tags

The `tagsAdmin` feature groups the operations whose effect goes beyond a single prompt : rename a tag *everywhere*, merge two tags, delete a tag with reassignment, export the catalog.

In a personal scope, you are necessarily the admin of your own tags ; the feature is still useful to decide whether you want those functions visible in your UI or whether you prefer the cleaner interface of a *Simple* preset.

In a group scope, `tagsAdmin` mirrors the membership level : only a group admin can rename or merge the group's shared tags. Plain members can create and tag, but not restructure. See Chapter XXV for the membership model.

10.7 — Common anti-patterns

Too many tags, not enough usage. When every prompt carries eight tags and half of them are never reused, tagging no longer serves filtering — it becomes cognitive overhead. Set yourself a discipline : a tag deserves to exist only if at least three templates will carry it.

Hierarchy that runs too deep. `marketing/campaigns/2024/Q3/B2B/SMB` is a six-level tag. Nobody will remember it. Stay at two or three levels ; beyond that, what you want is probably a container (Chapter XI).

Finer tags that should be a family tag. If a tag is carried by *every* prompt of a certain kind, it is a broad family tag (Chapter IX) or a container (Chapter XI), not a finer thematic tag.

Renames without merging. Renaming `client_acme` to `Client-Acme` creates a *new* tag — the old one does not disappear. Always use *Rename tag everywhere* (`tagsAdmin`) rather than a per-template search-and-replace.

10.8 — When tag filtering is not enough

Beyond a certain volume — several hundred templates, ten or fifteen thematic axes — tag filtering hits its limits. You will then want to combine filters (Chapter XII), introduce containers (Chapter XI) to structure the tree physically, or use the *split pane* (Chapter XIII) to manipulate two lists in parallel.

Tags remain the most flexible axis. But they are not the only tool ; the manual comes back to them regularly because they combine with everything else.

Chapter XI — Containers : folders vs collections

Containers are the tree structure of your prompt library. They look like folders in a file explorer — and that is exactly what they are, in the simplest case. But IziBuddy distinguishes two flavors, *folder* and *collection*, that share the same structure while playing very different roles. This chapter untangles the distinction.

11.1 — The `kind` field

Each container has a `kind` field. The two flavors this chapter contrasts are `"folder"` and `"collection"` ; two further kinds exist — `"library"` and `"project"` — which behave folder-like for addressing while a *project* also stops upward scope resolution for `@@@include`. Beyond the `kind`, everything else — name, parent, hierarchy, contents — is identical.

A **folder** (`kind = "folder"`) is for organizing. It is a binder. It groups templates by project, by client, by theme, by whatever makes sense to you. No semantics beyond the grouping. You manipulate it like a folder in a file explorer : create, rename, move, trash.

A **collection** (`kind = "collection"`) is more than a binder. It is a *class* in the POO²L L5 sense : it can be inherited from and it can expose **virtual templates** (templates resolved through inheritance rather than stored on their own). A member of a collection reaches its own inherited body through the `parent` reference (the prompt axis) and reaches members of a containing collection through qualified includes.

The `collectionOO2` feature surfaces this object-oriented dimension in the UI ; without it, a collection behaves functionally just like another folder.

11.2 — When to use a folder

The folder is good for 80% of organizational needs. It is the right choice whenever :

- you do not need inheritance between templates,
- the templates inside are independent of one another,
- you just want to group for faster retrieval.

A folder `Newsletters/Welcome` that holds four templates — *Welcome client*, *Welcome partner*, *Welcome internal*, *Welcome reminder* — is a typical case. No inheritance between them ; the folder just keeps them together.

The folder remains available at every POO²L level (L0 to L5). It does not prejudge the content : a folder can hold plain-text templates, POO²L L3 templates, anything.

11.3 — When to promote to a collection

The collection is the tool the moment you want to **factor out** — write once what should be common, leave it in a parent, *specialize* it in descendants. Four signs say it is time to promote a folder to a collection :

1. **You are duplicating the same instruction block across several templates.** That block should live in a parent and be inherited, not copied.
2. **Your templates share a section structure.** The *Context*, *Objective*, *Output format* sections recur identically in five templates of the folder — that is a class asking to exist.
3. **You want `parent` somewhere.** The `parent` reference — reaching a prompt's inherited body — only exists in a collection inheritance chain.
4. **You want virtual templates.** A virtual prompt is resolved on the fly by inheritance, without being physically stored as a separate prompt.

If none of these lights up, keep your folder. A collection brings power — and a bit of conceptual complexity — pay the cost only when you get something for it.

11.4 — A decision matrix in prose

Concretely, start with a folder. The second time you duplicate the same block across two templates in the folder, ask yourself the question. The third time, promote.

A CONTEXTUAL TIP (Chapter XXXIX) — `promote-collections-3-folders` — automatically suggests this promotion when the system detects three structurally similar folders.

The other way around : if a collection no longer serves inheritance, you can **demote** it back to a folder. Promotion and demotion are metadata operations — the `kind` changes, the templates stay. Feature : `containerPromoteDemote`.

11.5 — Promote, demote

Promotion opens the class declaration in the editor : the collection gets an identifier, can reference a parent, can expose virtual templates. Demotion is the reverse — it disconnects the class, inheritance links are broken, virtual templates become unreachable. The UI confirms before demotion when descendants are affected.

Collection-declaration extraction — `extract_collection()` in `src/core/inline_declarations_parser.py` — analyzes the POO²L declaration inside the container and feeds server-side inheritance resolution (Chapter XX for the UI details ; *Reference* §16–§19 for the semantics).

11.6 — Container-related features

The registry groups seven features in the *Containers* category :

- `containerKindUI` — shows the *kind* selector (folder vs collection) at creation and in the detail panel.

- `containerPromoteDemote` — exposes the promote and demote actions.
- `containerNewFolder` — *New folder* button in the toolbar.
- `containerNewCollection` — *New collection* button (separate from the previous : you can turn one on without the other).
- `containerPathDrop` — enables drag-and-drop of a prompt onto a container path segment (breadcrumb) for quick moves.
- `collectionOO2` — reveals the object-oriented surface : class declaration, parent, virtual templates, `parent`.
- `containerColorStripe` — shows a colored stripe on containers for quick visual grouping.

The *Simple* preset does not include `collectionOO2` : a beginner sees containers strictly through the folder lens. The *Standard* preset and above include it. It is a clear illustration of the principle « complexity climbs with the need ».

11.7 — The tree and movements

Containers form a tree. You create sub-containers without a depth limit (in practice, staying within three or four levels keeps things readable). Moving a prompt between containers happens through drag-and-drop in the left panel (feature `actionDragDrop` on the actions side), or by picking a new container in the detail panel.

Drag-and-drop onto the *path bar* at the top of the list — the breadcrumb — is driven by `containerPathDrop`. It is a subtle but effective bit of ergonomics : when you see `Marketing / Newsletters / Welcome` above your filtered list, you can drop a prompt directly onto any of those segments to move it into the corresponding container.

11.8 — Group containers vs personal containers

The tree is per scope. Your personal space has its tree ; each group you belong to has its own. When you switch scope, the left panel shows the matching tree.

A prompt always lives in *one* scope ; it cannot be in both your personal tree and a group’s tree at the same time. To « migrate » a prompt between scopes, you **duplicate** it (Chapter XXIV) — the copy goes to the other scope, the original stays.

Chapter XII — Searching and filtering

Once your library passes fifty templates or so, finding them quickly becomes the most frequent operation. IziBuddy exposes two tools — full-text search and a filter panel — that combine. You can use them independently, but it is their combination that makes the tool fast.

12.1 — Full-text search

The search field at the top of the dashboard queries live. It covers the text fields of a prompt :

- the prompt's **name**,
- its **description**,
- its **content** (the source — the raw string, not the rendered form),
- its **tags**.

Search is case-insensitive and uses substring matching by default : typing `news` finds `Newsletter`, `Newsletters`, `NewsRoom`. It matches on the literal characters, so it is **not** accent-folding — `etudiant` does not match `étudiant`. No regex on the search field, but combining filters covers most cases where regex might be tempting.

Crucially, search runs **entirely client-side**, over the prompt list already loaded into the page. There is no server search endpoint : `GET /api/templates/list` takes only an `includeDeleted` flag — it never receives a query string. Filtering and matching happen in the browser (`src/static/index-templates-render.js`), which is why results are instantaneous and why search keeps working without a round-trip.

12.2 — Filters : the side panel

The left side panel exposes filters organized in sections.

Markers : the six markers from Chapter VIII plus **Note** (stored as `no_scoring`). Ordinary filters offer neutral, show only and, when enabled, exclude. Hidden has the separate cycle below. Feature ids are `filterImportant` , `filterLocked` , `filterWip` , `filterNoScoring` , `filterHidden` , `filterDisabled` , `filterExample` .

Container kind : three filters — `filterFolder` , `filterCollection` , `filterTemplate` — to restrict the list to a single kind. Handy when the tree mixes everything.

Tags : the list of your tags, each clickable as a filter. In hierarchical mode (`tagsHierarchy`), parent tags display as expandable ; clicking a parent filters by all its sub-tags. This is also where your « category » family tags live (Chapter IX) : filtering by a top-level family tag narrows the list to that family.

Current container : implicit, defined by your navigation in the tree. Clicking a container in the left panel restricts the list to its content (recursive if you have configured it that way).

12.3 — Combining filters

A normal click replaces included criteria in the same family. Shift+click adds a criterion ; the AND/OR control determines how it joins the query. The field also accepts explicit AND, OR, NOT and parentheses. Ordinary attribute filters gain a red exclude state when `filtersExclude` is enabled ; otherwise they alternate between neutral and include.

Hidden always has three modes, independent of `filtersExclude` :

Appearance	Visibility	Query
Normal	Non-hidden only (default)	No Hidden term

Appearance	Visibility	Query
Green	Hidden only	`flag:hidden`
Grey ghost	Hidden and non-hidden	`show:hidden`

The grey ghost remains clickable. `show:hidden` lifts only the default Hidden restriction ; it does not bypass other criteria, including OR groups. Hidden containers do not expose their contents in outside flat searches. Enter a container to browse its contents ; the query and inherited Hidden context remain, and subfolders keep their hierarchy.

Folder, project and collection counts cover owned templates recursively, excluding trash. Without an active criterion, the badge shows the total. With a criterion, it shows **X of Y templates** : X matches in the subtree (using each prompt's own flags, not inherited flags), Y is the unchanged total. Zero matches does not mean an empty folder ; inherited/virtual copies are not additional owned templates. Library badges count owned collections.

`filtersExclude` is useful for cleanup at scale. Suppose you want to see everything that is *neither* Important *nor* Locked — set both of those filters to *exclude*. The list drops every prompt that matches either.

12.4 — The Clear all button

`filterClearAll` resets every filter in one click. When you work with four or five active filters and want to start from a clean view, this is the button to reach for.

It also clears the full-text search field, so a single click returns you to the default view : hidden items are excluded again, and the current navigation location is preserved.

12.5 — A few useful search patterns

« My recent Important templates » : Important marker on, sort by modification date descending. The « quick view » for your daily routine.

« All notes » : the Note filter (`filterNoScoring`) on, sort by creation date. Surfaces the templates carrying the `no_scoring` flag (Chapter XXIX).

« Project templates that are neither Locked nor Disabled » : navigate into the project container, set `filterLocked` and `filterDisabled` to *exclude*. You see the templates still editable without risk.

« Orphan templates » : if you have the habit of tagging systematically, filter the inverse — a dedicated filter shows the templates that carry *no* tag. That is the queue of the forgotten ones.

« A group's templates with a given tag but not another » : switch to the group scope, click the first tag positively, the second one negatively. A formidable combination for navigating a structured catalog.

12.6 — Search from the extension

The *Templates* tab in the extension exposes a similar search, but with a narrower scope : it queries the dashboard's API and applies its own lightweight filtering. Search there is keyboard-driven, with no filter panel : for complex selections, go back to the dashboard. See Chapter XXXII.

The *Stash* (Chapter XXXV) is not a separate local store : stashed templates are ordinary backend templates carrying the `STASH` tag. The extension surfaces them as a quick-access shelf, but they live on the server like everything else.

12.7 — When search hits its limits

On very large catalogs — thousands of templates — client-side substring search shows its limits. The escape route is to **rationalize** your organization : if you search too much, you may be short on tags or container hierarchy. Search is the tool of last resort ; the structural axes (family tags, finer tags, container) are faster when they are well thought out.

Chapter XIII — Views : tiles, table, split pane, dates, density

The same library of templates can be displayed in different ways. IziBuddy offers two layouts, a two-pane working mode, and several density settings. Picking the right view for what you are actually doing — browsing visually, or editing while reading a neighbour — changes the experience markedly.

13.1 — Two layouts

The layout selector sits at the top right of the main list, as a two-button switch between *grid* and *details*.

The **grid (tile) view** shows each prompt as a thumbnail with its name, short description, tags, and markers. It is the visual view : you scan a grid, you spot things by colour and icon. It works well when you do not know exactly what you are looking for and you are *grazing* through your library. It is always available.

The **details (table) view** shows templates as a spreadsheet : one column per field — name, description, tags, markers, version, date. It is the dense view, sortable by clicking a header, filterable through the side panel. It is the working view when you are editing at scale, or when you want to grasp your library as a whole. This view is gated by the `viewDetails` feature ; where the feature is off, the grid is the only layout and the toggle disappears.

Pick the layout by intent : grid to explore, details to work.

13.2 — Tile sizes

The `viewTileSizes` feature adds three sizes to the tile view — *small*, *medium*, *large*. Small tiles pack more thumbnails on screen (useful on very large libraries) ; large tiles give each card room to breathe with a more visible description.

Without the feature, the tile view uses a single size. The feature is part of the *Standard* preset and above ; in *Simple*, you stay on the default.

13.3 — Table density

`viewTableDensity` plays the same role for the table view : three densities — *comfortable*, *standard*, *compact*. *Compact* fits twice as many rows on screen ; *comfortable* spreads them out for long reading sessions.

Combined with `viewTileSizes`, you get two fine-tuning levers for your screen and your working mode. A MacBook Air does not carry the same payload as a 27-inch monitor ; the densities let you arbitrate.

13.4 — The details view toggle

`viewDetails` is what unlocks the *details* (table) layout described above. With the feature on, the layout selector offers the details table alongside the grid ; with it off, the pane stays on the grid and, if a pane was already showing details, it rolls back to the grid automatically. It is the switch between « browse visually » and « work in a spreadsheet ».

13.5 — Full or relative dates

By default, dates are shown relatively : *3 days ago*, *yesterday*, *just now*. Readable but imprecise.

Which of the two forms shows is decided by the TILE SIZE, not by a preference : large tiles have the room for *created → modified*, medium and small tiles for the modification alone. It is a question of space, so space decides — the renderer emits both forms and the stylesheet shows one. A toggle for it did exist and existed only for that ; it went away with the rules that now make the choice.

The toggle is per-user : your teammate can stay on relative dates while you work in absolute.

13.6 — The split pane

`viewSplitPane` is the feature that turns the interface into two panes side by side, each with its own list, its own search, its own selected prompt. It is the tool when you :

- compare two templates (side-by-side beats switching every time),
- edit a prompt while consulting another (the documentation, the inheritance parent, the collection neighbour),
- move by drag-and-drop between two containers (one in each pane).

The two panes are symmetric. Each can be in grid or details view independently. Each can have its own active filters.

Keyboard shortcuts make the split pane fluid. **Tab** switches the active pane. **F5** and **F6** act on the selection : F5 **copies** the active pane's selection into the other pane, F6 **moves** it (feature `action-CrossPaneF5F6` , *Actions* group of the registry). No modifier keys are involved ; the shortcuts are ignored while a text field holds focus. It is an expert move that pays off heavily in two-pane editing.

The split pane is in the *Expert* preset only. A contextual tip (Chapter XXXIX) suggests it past a certain volume — `cut-paste-burst-split` detects work patterns that benefit from it.

13.7 — Score visualization

`viewScoring` adds the score column in the details view, and a badge on tiles. The score is a quality value out of 10 computed by the analysis engine, not a user rating (Chapter XXIX).

When the feature is off, scores still exist in the background (the `filterNoScoring` filter keeps working), but they are not visible in lists. Switch it off for a clean uncluttered mode.

13.8 — Views and presets

Each UI preset enables a coherent subset of view features.

- **Simple** : grid view only, single size, relative dates, no scores, no split pane. Minimal interface.
- **Standard** : grid and details layouts, tile sizes, score display, relative dates by default ; no split pane. The balance point.
- **Advanced** : adds table density and full dates (and the `filtersExclude` exclude state).
- **Expert** : everything enabled, including the split pane.

You switch preset from the user menu ; you can also override feature-by-feature in the features panel (Chapter XXXVIII). See the exact feature list per preset in `src/static/index-features-registry.js` .

13.9 — Persisting your choices

Your view choices — layout, tile size, density, full dates or not — are stored in your web settings (`~/.izibuddy/users/{username}/settings/web.json`) and mirrored in the local cache `localStorage.izibuddy_features` . This avoids the *flicker* at the next start : the page appears immediately in the view you left.

The actual view rendering lives in `src/static/index-templates-render.js` on the front end ; the features registry is in `src/static/index-features-registry.js` .

13.10 — A few usage patterns

« Which of these ten templates should I use ? » Grid view, large size. You browse the descriptions, you spot by family tag. Thirty seconds.

« Monthly audit : what should I clean up ? » Details view, compact density, full dates, scores visible. Sort by last used and by score. One page is enough to see what has not been used for six months.

« I am writing a new prompt starting from a neighbour. » Split pane : the neighbour on the left (reference pane), the new prompt on the right. Tab switches the active pane ; F5/F6 copy or move a selection across. You read on the left, you type on the right.

« Move thirty templates from one folder to another. » Split pane, compact table in each pane. Multi-selection on the left, drag-and-drop to the list on the right. Chapter XIV details the mechanics.

Chapter XIV — Bulk operations

Scaling up changes the tool. When your library reaches several hundred templates, handling each one by hand becomes impractical. Bulk operations — multi-selection, drag-and-drop, cut-paste, keyboard shortcuts, undo and redo — are what turn IziBuddy from a prompt editor into a fully fledged document-management tool.

14.1 — Multi-selection

Three basic gestures select more than one prompt at a time.

Ctrl-click adds or removes a prompt from the current selection. You click the first one, you Ctrl-click the others ; all clicked templates end up selected at the same time. A Ctrl-click on an already-selected prompt removes it.

Shift-click selects a range. You click the first prompt, you Shift-click the last ; every prompt between them (in the current list order) is selected. The classic file-explorer gesture.

The rubber band — *rubber band* in the code — is the third gesture. The `actionRubberBand` feature lets you draw a selection rectangle with the mouse : click on empty space, drag, and every prompt inside the rectangle is selected. The fastest gesture when the templates to select are visually adjacent.

A selection can hold a handful of items or several hundred ; the operations that follow work at the same speed in either case (one network round-trip for the whole batch).

14.2 — Drag-and-drop

`actionDragDrop` enables drag-and-drop of one or several selected templates. Three common destinations :

- a **container** in the left panel — moves the templates into that container,
- a **tag** in the side panel — applies that tag to the templates,
- the **trash** — sends the templates to soft delete.

Drag-and-drop respects the selection : if three templates are selected and you drag one of them, the three follow. It is the universal file-manager idiom, transposed.

The `containerPathDrop` gesture (Chapter XI) extends this to the breadcrumb : you drop on a path segment to move into that container. Combined, you get very fluid navigation from the table to the containers.

14.3 — Cut-paste, copy-paste

`actionCutPaste` and `actionCopyPaste` add the classic keyboard operations :

- **Ctrl+X** cuts the selection (the templates are flagged as « to be moved » and stay visible in place until you paste).
- **Ctrl+C** copies the selection (the templates will be duplicated at the destination).
- **Ctrl+V** pastes at the current insertion point — typically the container selected in the left panel or in the split pane.

The difference from drag-and-drop : cut-paste works *across panes* in the split view, *across distant containers* of the tree, and lets you prepare an operation (cut, then navigate elsewhere, then paste) without having to hold the mouse button down through the navigation. The two gestures coexist ; pick whichever fits.

`actionCopyPaste` produces duplicates — the pasted templates are independent copies of the originals, just like the *Duplicate* button of Chapter VII.

14.4 — The F5 and F6 cross-pane shortcuts

`actionCrossPaneF5F6` is aimed at the split pane. When two panes are open (Chapter XIII), they act on the active pane's selection :

- **F5 copies** the selection into the other pane.
- **F6 moves** the selection into the other pane.

To change which pane is active, press **Tab**. So a full cross-pane move is : select in the active pane, **F6**, done — the templates land in the other pane's current container in one keystroke, no mouse, no copy/paste dance. Both operations are undoable (Chapter XIII). The shortcuts take no modifier keys and are ignored while a text field holds focus.

A contextual tip — `cut-paste-burst-split` — detects this pattern and suggests turning on the split pane if it is not already.

14.5 — Undo and redo

`actionUndoRedo` adds **Ctrl+Z** / **Ctrl+Y** on top of bulk operations. An accidental deletion, a botched move, a mis-applied tagging — one Ctrl+Z restores the previous state.

History is bounded to the current session : Ctrl+Z does not survive a page reload. For longer rollbacks, you rely on versioning (Chapter XXVIII), which survives everything. Undo-redo is the in-the-moment tool, not the archival one.

A tip — `first-deletion-undo` — fires on the user's first deletion and reminds them that Ctrl+Z exists. The kind of help that saves a beginner from an accidental delete.

14.6 — The bulk update endpoint

On the API side, all these bulk operations go through a single endpoint : `PUT /api/templates/update/bulk`. The request body is a list of `(id, patch)` pairs — a prompt id and the delta to apply. Each patch is applied **atomically per prompt** ; there is no cross-template transaction, so a partial failure leaves the earlier successes committed. The response reports a per-id status so the client can show a success/error count without follow-up calls.

This shape has two practical consequences.

On the one hand, **performance is constant** : moving 200 templates takes roughly as long as moving 5. One network round-trip. The UI stays responsive even on very large selections.

On the other hand, **Git commits are grouped** when a Git repository is configured (Chapter LIX). Instead of 200 *Move prompt X* commits, you get a single *Bulk move : 200 templates* commit. The history stays clean.

The endpoint accepts a fixed whitelist of fields : the marker flags (`important` , `locked` , `disabled` , `wip` , `no_scoring` , `is_example` , `virtual`), plus `tags` , `color` , `short_name` , `notes` , and `collection` . Name, description and content are **not** in the bulk set — editing those stays single, through `PUT /api/templates/update/{id}` .

14.7 — Bulk delete and restore

Destructive operations have their own dedicated endpoints to make their scope obvious :

- `DELETE /api/templates/purge-selection` — purges a checked set of templates in a single transaction. See also Chapter VII.
- `DELETE /api/templates/purge-all` — empties the trash entirely.

Bulk restore stays on `POST /api/templates/restore/{id}` called for each id ; on the UI side, you select the templates in the trash and press *Restore selection* — the UI iterates, but the user-facing result is instantaneous.

14.8 — A few typical patterns

« Tag thirty templates at once. » Multi-selection (rubber band or Ctrl-click), right-click, *Add a tag*. One request, thirty templates tagged.

« Migrate a folder into another container. » Navigate into the source folder, *Select all* from the context menu, Ctrl+X, navigate to the destination, Ctrl+V. Three seconds for a hundred templates.

« Quarterly cleanup. » `filterDisabled` filter on, *Select all*, *Purge selection*. One confirmation, the trash drains the disabled templates.

« Re-tag a batch. » Filter by the tag to change, *Select all*, drag-and-drop onto the new tag. The whole selection switches in one block.

14.9 — Combining with the split pane

The split pane (Chapter XIII) plus bulk operations is the most powerful combination for reorganizing. Two containers side-by-side, selection on the left, F6 to move it across, Tab back to the left pane, next selection, and so on. This is also the pattern the `cut-paste-burst-split` tip highlights.

Chapter XV — Import and export

A prompt library is an asset. It grows, it travels — from a personal laptop to a team instance, from a test environment to production, from one machine to its replacement. IziBuddy ships with a JSON import/export pipeline designed for exactly that : move a coherent set of templates, collections and libraries from one instance to another with predictable conflict resolution.

This chapter covers the user-side workflow and the admin path. The format is documented enough that you can also generate or read it from your own tools.

15.1 — Exporting

The **Export** action lives in the user menu of the dashboard and hits the endpoint `GET /api/templates/export`. Two scopes are available :

- **My templates** — the default ; exports every prompt you own.
- **Group templates** (group admins) — adds the templates of a group you administer.

Two flags refine the output : `include_trash` toggles trashed templates (off by default), `include_my` lets a group admin add their own personal templates on top of the group's templates.

The output is a single JSON file : a **POO²L package**, the normative interchange format of the standard (§7.4), identified by its `poo2l_package: "1.0"` envelope :

```
{
  "poo2l_package": "1.0",
  "exported_at": "2026-05-21T14:30:00Z",
  "templates": [
    {
```

```
      "storage_id": "3f2a9c1e-…",
      "content": "@@@template Dev.CODE-REVIEW\n@@@name Code review\n@
      "x_izibuddy": { "tags": ["dev"], "owner": "jacques" }
    }
  ],
  "collections": [ { "storage_id": "…", "content": "@@@collection Dev
  "libraries": [],
  "external_dependencies": [ { "kind": 'template', "reference": "Comm
  "x_izibuddy": { "exported_by": "jacques", "include_trash": false, "
}
```

Each prompt record is deliberately small : the POO²L `content` is the **single source of truth**. Title, identifier, collection, inheritance, parameters, constants, sections — everything the text expresses lives in the text, never in a parallel structured field. What the text cannot express travels in the record's `x_izibuddy` block : tags, color, lock, notes, owner, timestamps and the other application metadata (default values are omitted). Blocks written by other tools (`x_othertool`) are re-emitted verbatim. A record for a *Level-0* prompt — raw text with no directives — adds an `identity` object (`id` , `name` , `description`) since the content alone cannot carry them. Every `x_*` block is **strippable** : remove it and you still have a conforming package that renders identically.

Collections and libraries travel as first-class objects (`collections[]` / `libraries[]`), each with a synthesized `@@@collection` / `@@@library` block as its content and a deterministic `storage_id` — importers match containers by name/path, never by that id. Folders and projects are not packaged entities : they ride the envelope's `x_izibuddy.folders` . The templates list is flat — there is no per-user grouping ; ownership is carried per record by `x_izibuddy.owner` . One known limitation : attachment payloads (`record:` / `blob:` storage) are not packaged — only their declarations travel, inside the content. The legacy Markdown export of older versions of IziBuddy is gone ; the POO²L package is now the only format, and it is comprehensive enough to round-trip.

15.2 — Importing

Import is a **two-phase** operation, on purpose : it lets you see what will happen *before* it happens. The endpoint `POST /api/templates/import` accepts a `mode` parameter :

- `mode=check` — parses the file, validates the envelope (only `poo2l_package` 1.x is accepted), lists the templates found, and computes conflicts against the current library. Nothing is written. The response is a report.
- `mode=execute` — performs the actual import based on the conflict-resolution choice you make from the report.

The conflict policy is global to the operation : either `skip` (existing prompt wins, imported one is dropped) or `replace` (imported prompt overwrites the existing one). Finer per-template resolution is not in the API — if you need surgical control, run `check`, drop the problem entries from the JSON file by hand, and re-run.

Before any `execute`, the server takes a full BACKUP of your templates via the `BackupManager` (see Chapter L). If the import goes wrong, you can roll back to the timestamped `.xz` archive without touching anything Git-side.

15.3 — Ownership and the admin path

Ownership does not need a separate envelope section : every record carries its owner in `x_izibuddy.owner`. On import the routing follows from who you are. A regular user imports everything into their own account — foreign owners are ignored and group sharing is cleared, so a package can never escalate rights. A system administrator imports each record into the account named by its `x_izibuddy.owner`, which is the path for migrations between instances : export from the source, copy the file, import on the target as admin, verify, and you're done. Containers (collections, libraries, folders) are recreated **before** the templates they contain ; a container that already exists on the target is never overwritten.

15.4 — Format stability

The `poo2l_package` field at the top of the file is what the importer checks. Only `poo2l_package` 1.x files are accepted — there is no legacy reader, and the proprietary envelopes shipped by prior IziBuddy versions are rejected. If you have old archives, re-export them from an IziBuddy version that produces POO²L packages.

The format is the standard's interchange format (§7.4), versioned for a reason : it will evolve. Treat the JSON files as transport artifacts, not as a long-term storage format — keep your templates in IziBuddy, where the `BackupManager` and Git mode (Chapter L) handle persistence properly.

Partie 3

Authoring with POO²L
(manual-side surface)

Chapter XVI — The smart editor

IziBuddy's prompt editor is not a plain `<textarea>`. It is an editor that is aware of POO²L syntax : it colorises `@@@=...` blocks as you type, offers completions when you open a bracket, and flags unresolved references before you save. The goal is not showiness — it is to reduce the typos that, in a strict-syntax language, silently turn a parameter into a literal.

This chapter covers the editing surface only. The specialized dialogs — parameters, constants, attachments — are the subject of Chapters XVII through XXII.

16.1 — Syntax highlighting

The smart editor recognizes the `@@@` sigil that opens every POO²L construct — line directives (`@@@keyword …`), inflowing substitutions (`@@@=name`), inflowing includes (`@@@{include …}`) and comments (line `@@@-- …` , bounded `@@@(-- … )`). Each token is coloured according to the keyword that follows the sigil :

- **Red** : `include` (and its variants `include parent` , `include self.` , `include parent.`) — a call that must resolve to another prompt or to a parent section.
- **Orange** : `section` , `section?` , `section!` , `section virtual` , `section final` — a section marker. The `?` / `!` suffix marks a user-toggled section (the user decides include/exclude at instantiation, with `?` defaulting OFF and `!` defaulting ON) ; the `virtual` / `final` modifier — placed *after* the `section` keyword — marks the section *abstract* (descendants must implement) or *sealed* (descendants must not redeclare).
- **Yellow** : `param` , `param?` — a prompt parameter.
- **Green** : system-variable substitutions `@@@=DATE` , `@@@=USER` , etc.

- **Blue-grey** : `verbatim` and `end-verbatim` — uninterpreted block.
- **Grey** : the comment markers `--` (line and bounded `@@@(-- … )`) and the block `comment` / `end-comment` — text ignored at render time.

The palette code is centralized at the top of `src/static/index-smart-editor.js` (constant `AUTOCOMPLETE_OPTIONS`). Changing a colour happens in a single place — useful to know if your custom theme clashes visually with a hue.

16.2 — Autocomplete

As soon as you type the sigil `@@@`, a dropdown menu opens below the cursor and offers the available POO²L keywords. The menu narrows down as you keep typing : `@@@in` filters the list down to `include`, `include parent`, `include self.`, and so on.

Keyboard selection : up/down arrows to navigate, `Enter` or `Tab` to insert, `Esc` to close the menu. On insertion the editor places the cursor at the useful spot in the skeleton — typically inside the braces, so that you immediately type the name of the referenced element. For instance, picking `include` inserts `@@@{include }` and positions the cursor between the two spaces in the middle.

A second flavour of completion kicks in for identifiers that already exist : when the editor knows you are typing a parameter name, the menu lists the parameters already declared in the prompt. Same for constants and for prompt ids referenced by `include`. This contextual completion saves the reader from having to switch to the parameters panel to double-check the exact spelling.

16.3 — Inline validation

On every edit, a lightweight parser analyses the content and surfaces two kinds of visual signals :

- **Red squiggle** : unresolved reference. The most common case is an `include` pointing at a non-existent prompt id, or a `@@@param X` whose name `X` is not declared in the parameters panel.
- **Yellow squiggle** : non-blocking warning. Typically an `include` to a prompt marked *deprecated*, or a parameter that is declared but never used in the body.

These signals do not block saving. IziBuddy would rather let you save an intermediate state — you know what you are doing — but the warnings save you from discovering a problem at injection time.

One validation signal is different in kind : an `@@@assert` failure. Assertions are checked at *render* time, against the finished text, not at edit time — so they surface in the render preview rather than as an inline squiggle, and unlike the warnings above they are **blocking** : a prompt whose rendered content fails an assertion cannot be injected until the content (or the assertion) is fixed. Authoring assertions is covered by the POO²L reference ; here it is enough to know that a blocked render reading « assertion failed » points at a guarantee the prompt's author wrote on purpose.

16.4 — Notes and attachments in the editor

Two features live alongside the prompt body and are accessible from the editor when their feature flags are active :

- `editorNotes` opens a **Notes** side panel : a secondary editor for your editorial comments on the prompt. These notes never flow into the rendered output — the engine ignores them, so the chatbot never sees them. They are, however, stored *with* the prompt and travel with it : an export carries them in the record's `x_izibuddy.notes` block, so anyone who can see the prompt can read them. Chapter XXI details their use.

- `editorAttachments` enables the **attachments** manager : fixed files attached to the prompt, or files asked for at injection. The POO²L-side declaration is `@@@attachment ...` ; the UI side is covered in Chapter XIX.

When either feature is off, the corresponding pane disappears entirely from the editor — no greyed-out button, no empty panel. That is the general feature philosophy (Chapter XXXVIII).

16.5 — `contenteditable` or `textarea` editor

Under the hood, IziBuddy ships two editing engines. The historical engine is a classic `<textarea>` augmented by `src/static/index-textarea-utils.js` (indentation, selection, shortcuts). The newer engine — active when highlighting is requested — is a `contenteditable` driven by `src/static/index-contenteditable-editor.js`, which makes real-time colouring possible without re-injecting the DOM on every keystroke.

The switch between the two is transparent : you do not see it. If ever highlighting stops working on an exotic input (large paste, significant non-UTF-8 content), fall back to `textarea` mode by temporarily turning off the corresponding feature — the loss is purely visual.

16.6 — Useful shortcuts

The editor respects the usual conventions :

- `Ctrl+Z` / `Ctrl+Y` : undo / redo.
- `Ctrl+S` : save.
- `F5` / `F6` : in *split pane* mode, **copy** (F5) or **move** (F6) the current selection to the other pane — a file-manager-style gesture, not a focus switch ; `Tab` toggles which pane is active (feature `actionCrossPaneF5F6`, see Chapter XIV).

The undo history is local to the editing session : it does not replace versioning (Chapter XXVIII), which records saved states for the long run.

16.7 — Editor versus rendered output

What is visible in the editor is never what is sent to the AI. Rendering — substituting system variables, expanding includes, resolving inheritance, applying parameters filled in at injection — is delegated to the POO²L engine through the render endpoints of `src/api/templates.py`, covered in Chapter XXII. The editor shows the **source** ; injection manipulates the **rendered form**. This is the most fundamental distinction to keep in mind as you read on.

Chapter XVII — Parameters from the UI

A parameter is a piece of data that the prompt's author delegates to the end user at injection time. It is the mechanism that turns a prompt into a reusable template : you write the structure once, and each execution populates it differently. In POO²L, a parameter is declared with the `@@@param ...` directive ; from the dashboard, you generally do not have to type that directive yourself — a dialog does it for you, validates the constraints along the way, and inserts the correct syntax into the content.

This chapter covers the UI surface : how to open the dialog, which types are available, how to express constraints, how to distinguish required from optional, and what the dialog looks like on the end-user side.

17.1 — Opening the parameter dialog

In the smart editor, the **+ Parameter** button in the editor's side toolbar opens the parameter dialog. The declared parameter is inserted at the current cursor position.

Parameters and constants live in the **Definitions** panel, opened from the ⋮ menu of the editor. No feature toggle governs it : the editor's POO²L surfaces are not features. What governs them is the **editing-level ceiling** (`pool_max_level` , Options → Editor → Input) — one question, one control — and the panel is reachable whatever the preset. You may equally edit parameters by typing the POO²L directive directly, or switch to the Advanced preset from the Options page.

17.2 — Picking a type

The dialog offers a drop-down list of types. Each type drives the widget that the injection dialog will display to the end user, and each type accepts a distinct subset of constraints :

- **text** — single-line string. Widget : simple input field. Constraints : `minlen`, `maxlen`, `pattern`.
- **textarea** — multi-line string. Widget : resizable text area. Same constraints as `text`.
- **number** — numeric value. Widget : numeric field with steppers. Constraints : `min`, `max`.
- **date** — ISO-formatted date. Widget : the browser's native date picker. No additional constraint.
- **boolean** — true or false. Widget : checkbox or toggle. No additional constraint.
- **select** — choice from a finite list. Widget : drop-down menu. The choices are carried by the type itself, as `select("a", "b", "c")`.

The chosen type is recorded in the directive after the `:` type separator, as `@@@param name : <type>`. If you edit the directive by hand later, you can change the type ; the dialog will reopen in sync with the value it finds.

17.3 — Expressing constraints

The dialog's **Constraints** panel shows the relevant fields per type. Three families :

Length and bounds. For `text` and `textarea`, `minlen` and `maxlen` set the acceptable string length. For `number`, `min` and `max` set the numeric range. Constraints are written as bare `key=value` tokens on the directive (no surrounding spaces) ; the engine recognizes `minlen`, `maxlen`, `pattern`, `min` and `max`.

Shape. For `text` and `textarea`, `pattern` accepts an ECMAScript regular expression that the input value must satisfy. The dialog validates the regex as you type; if it is malformed, the field turns red and saving is blocked.

Finite choices. For `select`, the **Options** panel is an editable list of `value=label` entries. The value is what will be substituted at render time; the label is what appears in the drop-down menu on the end-user side.

An invalid constraint at input time produces a message under the relevant field and prevents the parameter from being saved. Valid constraints are serialized into the directive as `@@@param name : text minlen=3 maxlen=80`. A default value, when present, is introduced by a standalone `=` token and comes after the constraints: `@@@param name : text minlen=3 = "draft"`. An optional trailing label is a literal quoted string, not a translation key: `@@@param name : text = "draft", "Document title"`.

17.4 — Required or optional

The dialog offers an **Optional** toggle. When on, it appends the `?` marker to the keyword, turning `@@@param` into `@@@param?`. The distinction is semantic:

- **`param`** — the value must be supplied. If the user leaves the field empty in the injection dialog, the **Send** button stays disabled.
- **`param?`** — the value can be omitted. An optional parameter must carry a default (the `=` token); when the user leaves the field empty, that default is substituted. It is up to you to choose a default that reads gracefully when it stands in for a missing value.

The default value, introduced by the `=` token, is used by both variants: for `param`, it pre-fills the field the user may accept or change; for `param?`, it is what gets substituted when the user leaves the field empty.

17.5 — The user-side dialog

When the extension prepares the injection, it inspects the rendered prompt, locates the parameters still to be resolved, and opens an input dialog. That dialog is generated dynamically :

- One row per parameter, in the order they appear in the prompt.
- The widget native to the type (text field, checkbox, date picker, and so on).
- The label is the parameter identifier, unless a trailing quoted label string was declared on the directive.
- Constraints are replayed on the client : a field that violates `pattern` , `min` , or `maxlen` is marked red and the **Send** button stays disabled until every required parameter is valid.

If the prompt has no parameters at all, the extension skips the dialog and injects directly. If the prompt has only `param?` parameters all supplied through defaults, the dialog still appears — IziBuddy's philosophy being that the end user always sees what they are about to send before sending it.

17.6 — Parameter sets

When a prompt is regularly run with the same values (for example a recurring project name), the end user can save their entries as a **Parameter set**. On the next call to the same prompt, the extension offers the list of saved sets ; picking a set pre-fills every field in one click. Sets are stored **server-side**, per prompt and per user, under the user's presets directory in `templates.repo` ; they are served by `GET /api/templates/presets/{id}` , so the same sets follow the user across machines.

Creating and selecting a set happens from the injection dialog — see Chapter XXXII for the extension-side procedure and Chapter XXXV for the storage details.

Chapter XVIII — Constants from the UI

Constants look like parameters — they are named values substituted into the rendered prompt — but they differ on a fundamental point : the end user is never asked to supply them. They are fixed by the prompt's author, live with the prompt, and apply silently at render time. The injection dialog does not show them.

The usefulness of a constant becomes obvious as soon as a prompt holds a stable value that *could* change one day. Hard-coding your organization's name in the prompt content is legal but uncomfortable : the day you want to change that name across forty-two templates at once, you will regret having put `Acme Corporation` in every body instead of a reference to a single constant.

18.1 — The constants panel

The constants panel is reached from the smart editor through the **Constants** tab of the side panel. It lists the constants already declared in the current prompt, as a table :

- **Name** — the identifier declared with `@@@const name` and referenced in the body through the substitution `@@@=name` .
- **Value** — the string that will be substituted at render time.
- **Type** — text, textarea, number, date, or boolean. Used for display and validation, not for substitution (which is always textual in the end).
- **Description** — free text, displayed as a tooltip in the smart editor when the user hovers a reference to the constant.

The **+ Constant** button opens a dialog modelled on the parameters one, without the Optional toggle and without the Sets checkbox — by construction, a constant has no alternative value and is not saved on the user side.

Constants share the **Definitions** panel with parameters, opened from the ⋮ menu of the editor. No feature toggle governs it : the editor's POO²L surfaces follow the editing-level ceiling (`pool_max_level`, Options → Editor → Input), not the feature registry. An author who prefers to may use parameters with a default value for the same purpose, at the cost of a longer dialog for the end user.

18.2 — Why not a defaulted parameter

The distinction between a constant and a defaulted parameter is sometimes subtle. Three criteria settle the call :

Intent. A constant expresses « this value will *never* be supplied by the end user ». A defaulted parameter expresses « this value is usually this one, but the user may override ».

Injection dialog. A constant does not appear in the dialog. A defaulted parameter does appear, with its value pre-filled. If the prompt has only constants (and no parameters), the dialog is skipped entirely — injection is immediate.

Cognitive load. An injection dialog with ten parameters, eight of which have defaults nobody changes, becomes an intimidating wall of fields. Promoting the eight to constants leaves two visible fields : the experience is dramatically better.

18.3 — Typical uses

Three families of use recur across every deployment observed :

- **Organizational identity.** Company name, e-mail signature, legal warning line. One constant per value ; a rebranding updates the templates by editing the constant in each prompt — or, better, in a shared parent prompt through inheritance (Chapter XX).
- **Tone and persona.** Brand voice (« formal », « casual », « technical »), expected level of detail, response language. These rarely vary per prompt : they are decided at the template-family level.

- **Addresses and URLs.** Links to internal documentation, the wiki, a Git repo, a Slack channel. Hard-coding the URL in the content forces you to fix N templates when it changes ; one constant per domain solves the problem.

18.4 — Constants and inheritance

A constant declared in a parent prompt (or in a collection, the case that formalizes the « class » role) is inherited by the children. The child can override the constant by redeclaring it — same name, different value — or leave it as is. Resolution walks the inheritance chain from the bottom up, just like for parameters.

This mechanic is what makes constants useful at scale : declare `organization` in a root collection, and the forty templates that inherit from it automatically pick up the value. The day the organization is renamed, you edit the constant in the root collection, and the forty templates follow.

18.5 — Editing an existing constant

Clicking a row in the panel opens the dialog in edit mode :

- The value is editable, the type is locked (changing the type would change the semantics of references already in place).
- The name is editable, but this renames only the `@@@const` *declaration* — the metadata entry. It does **not** rewrite the `@@@=old` references sprinkled through the body ; those you must update yourself. Rename with care, and use the editor's inline validation (Chapter XVI) to catch references left dangling.
- Deletion asks for confirmation if references to the constant's name exist in the content : removing a referenced constant would leave unresolved calls behind.

Chapter XIX — Attachments

A prompt is not always pure text. Sometimes you want to attach an image, a PDF, a code fragment for analysis — something the chatbot will receive alongside the conversation. POO²L handles this through the `@@@attachment ...` directive, and the dashboard drives it from a dedicated picker.

This chapter covers the UI surface : how to add an attachment, the difference between a **fixed** attachment and a **to-be-supplied** attachment, the optional variant, and inline attachments encoded as data-URIs.

19.1 — The attachments picker

The **Attachments** tab of the editor's side panel lists the attachments declared in the prompt. Each row shows :

- The logical name (the identifier used in the directive).
- The MIME type, detected or declared.
- The size (for stored files).
- The origin : *fixed* (the file is attached to the prompt and stored on the server), *to be supplied* (the end user provides it at injection time), or *inline* (data-URI encoded in the content).

The **+ Attachment** button opens a dialog where a radio control picks between the two binding times — a *fixed* attachment or a *to-be-supplied* one (an inline attachment is just a *fixed* one whose source is a data URI). The manager is driven by the `editorAttachments` feature, active in the Standard, Advanced and Expert presets.

19.2 — Fixed attachments

A fixed attachment is attached to the prompt at creation time. The file is uploaded to the server, stored through the configured object-storage connector (see Part IX), and referenced in the directive by its logical identifier :

```
@@@attachment briefing : application/pdf fixed file "briefing.pdf"
```

The `fixed` keyword is the binding time (bound at definition) ; the `file` keyword is the source (where the content lives — here, a server-stored path ; the other sources are `blob` for an inline data URI and `record` for an external store reference).

At injection, the extension fetches the file from the API and attaches it to the request sent to the chatbot, through the attachment mechanism specific to each platform (simulated drag-drop, enriched clipboard, the platform's **+** button).

Typical uses : a reference PDF common to all uses of the prompt, a grading rubric, a logo, a diagram. The file does not change from one call to the next — the prompt and the attachment live together.

19.3 — To-be-supplied attachments

A to-be-supplied attachment is declared by the author but not provided at creation time. At injection, the injection dialog adds a file field (drag-drop or **Browse** button) ; the end user picks the file from their workstation and it travels with the request.

The directive uses the `to-include` binding time, which carries no source — the content is provided at injection — followed by a comma and a user-facing description :

```
@@@attachment image : image/* to-include, "Image to analyse"
```

The type can be a precise MIME (`image/png`), a generic MIME (`image/*`), or a comma-separated list (`image/*, application/pdf`). The widget in the injection dialog filters the accepted files accordingly.

Typical uses : a prompt that analyses an image, summarizes a PDF, critiques a code fragment. The prompt body is stable ; what you feed it changes every call.

19.4 — The optional variant

As with parameters, the `attachment?` directive makes a `to-include` attachment optional (the `?` is only meaningful with `to-include`, since a `fixed` attachment is always present). At injection :

- If the user supplies a file, it is attached normally.
- If the user does not, the request goes out with no attachment.

It is up to the prompt's author to write content that handles both cases. A common phrasing : « If an image is attached, analyse and describe it. Otherwise, ask the user to clarify their question. » The chatbot receives the instruction and branches based on what it sees as an attachment.

19.5 — Inline attachments

For small fragments — an SVG diagram, a few-kilobyte PNG logo, a CSV excerpt — you can encode the attachment directly in the directive as a data-URI :

```
@@@attachment logo : image/png fixed blob "
```

This is a `fixed` attachment whose source is `blob` — the bytes are embedded inline as a data URI rather than stored as a separate file.

The upside : no server-stored file, no extra HTTP request at injection. The downside : size bloats the prompt content (base64 adds about 33 % over the binary size) and editor readability suffers.

Because the data URI can run long, keep inline attachments to the small fragments described above ; anything larger belongs in a *fixed file* attachment stored on the server. Saving and rendering treat the inline data URI like any other attachment source.

19.6 — The user-side dialog

At injection, if the prompt declares one or more *to-be-supplied* attachments, the injection dialog surfaces them in a dedicated section, below the parameters. Each attachment is a drag-drop zone with a **Browse** button ; a preview appears as soon as a file is selected.

Fixed attachments and inline attachments are not shown in the dialog — they are silently prepared in the background and attached to the request at send time.

Chapter XX — Inheritance and includes from the UI

Inheritance is the operation that sets POO²L apart from a plain template language. A child prompt *is* a parent prompt — it inherits its parameters, constants, sections, and can override any of them at will. An `include` is the complementary operation : it pulls the content of another prompt and inserts it verbatim at a precise spot, with no identity relationship.

The dashboard drives both operations from the UI : a parent picker in the prompt metadata, a context menu on include references, an inheritance-chain viewer, and cycle detection at save time.

20.1 — Picking a parent

The **Parent** field in the editor's metadata panel is a prompt picker. Click it, a list of templates opens, and you pick one. On the POO²L side this sets the header's `inherits` clause : `@@@template child inherits parent`. There is no separate `@@@inherits` directive — inheritance is declared inline on the prompt header.

The **Template** marker is a useful convention here : it has no technical role on its own but signals an author's intent — « this prompt was meant to be extended, not called directly ».

Once the parent is picked, the editor recomputes the effective list of parameters, constants, and sections by walking the chain. Inherited elements appear in italic grey in the side-panel listings — you see them, you can override them, you cannot delete them from the child.

This mechanic is not behind a feature toggle. Like every POO²L surface of the editor it answers to the **editing-level ceiling** (`pool_max_level`, Options → Editor → Input) : inheritance is L4, so the ceiling has to reach it. One question — how far into POO²L ? — and one control.

20.2 — The inheritance-chain viewer

The **View chain** button next to the parent picker opens a panel that shows the full chain, from bottom to top :

```
My prompt (current)
  └─ Writing brief (direct parent)
       └─ Marketing persona (grandparent)
            └─ Root persona (template)
```

Each line is clickable : a click opens the corresponding prompt in a new editor tab (or in the secondary pane if split mode is active — see Chapter XIII). The panel is backed by the `GET /api/templates/get/{id}/inheritance-chain` endpoint, which returns the resolved chain as a plain list of `{id, name}` nodes, root first and the current prompt last. The response is deliberately minimal — it carries no per-node breakdown of declared parameters, constants or sections ; the effective inventory is computed by the editor as it walks the chain.

20.3 — Overriding an inherited element

An override is declared like any ordinary element — same directive, same identifier as the parent — but the editor flags it visually : the row in the Parameters (or Constants, or Sections) panel carries a small chevron `↻` indicating that the name already exists in the chain. Hovering the chevron shows where the inherited definition comes from and how yours differs.

Overrides need not be exhaustive : you can change a single constraint (for example bump `maxlen=80` up to `maxlen=200`) while letting everything else stay inherited. General rule : the nearest version in the chain wins, attribute by attribute.

20.4 — Includes

The `@@@{include other_template}` directive inserts the rendered content of another prompt at the exact spot where it appears. Three variants :

- **`include name`** — includes the prompt with id `name` (or exact name).
- **`include self.section`** — includes the content of section `section` of the current prompt (useful to reorder or recompose without duplicating).
- **`include parent.section`** — includes the same-named section of the direct parent. This is the equivalent of `super()` in object-oriented languages : you insert the parent's version at a precise spot, then enrich it.

From the UI, the smart editor's autocomplete helps with includes : typing the sigil `@@@` and picking `include` inserts the `@@@{include }` skeleton and positions the cursor between the braces, ready for you to type — or pick from the contextual completion list — the id of the prompt or section to reference (Chapter XVI).

20.5 — Cycle detection

The inheritance chain is checked for cycles **at save time** : if you try to save a prompt whose parent eventually loops back to the current prompt, the save is refused with a message pointing to the offending link (the server's `inheritance_validator` raises a circular-inheritance error).

Include cycles are a different matter. A prompt that transitively includes itself is caught by the **engine at render time**, not at save — the renderer walks the include graph as it expands and reports the loop as a blocking diagnostic. So a broken include does not necessarily block a save ; it surfaces when the prompt is rendered.

On the client side, the smart editor underlines any unresolved reference in red as you type, giving you the earliest warning.

Chapter XXI — Notes and sections

Two neighbouring editor mechanics deserve a chapter of their own : **notes** and **sections**. Both are auxiliary constructions — they either add no text to the rendered prompt (notes) or structure the content without changing its semantics (sections). Mixing them up is a classic mistake among new authors ; this chapter clarifies their roles and their UI surface.

21.1 — The Notes pane

The **Notes** pane is a secondary editor reached through the **Notes** tab of the smart editor's side panel. You write whatever you want in it : the why of the prompt, its known limitations, an informal history of changes, links to the Slack conversation where the need was raised, bibliographical references.

Three properties to remember :

- **Notes never reach the rendered output.** They are stored with the prompt, but the render engine systematically ignores them. The chatbot will never see them.
- **Notes are not inherited.** Each prompt has its own notes ; a child does not see the parent's notes and vice versa. They are strictly attached to the instance.
- **Notes are visible to anyone who can see the prompt.** If you share the prompt (Chapter XXIV) or publish it in a group (Chapter XXVII), the recipients see the notes. Think of them as source-file comments, not as a private scratchpad.

The pane is driven by the `editorNotes` feature, active in every preset except Simple. The syntax is Markdown : headings, lists, links, inline code, and code blocks render in the preview, as in any Markdown editor.

21.2 — When to use a note

A note is welcome every time you feel the urge to write a sentence to yourself or a colleague inside the prompt content, starting with `<!-- ... -->` or `// ...`. Those comments eventually leak into the rendered output, by accident or fatigue. The note removes the temptation.

Typical uses observed in deployments :

- **Provenance.** « This prompt comes from a discussion with X on March 12 ; the original need was to cover case Y. »
- **Known limitations.** « On ChatGPT-3.5 this prompt sometimes drifts into free-form output despite the instruction ; on GPT-4 and above, reliable. »
- **Links to internal documentation.** « Full procedure : see wiki page Onboarding Security §3.2. »
- **TODOs and refinement ideas.** « Try whether a ‘respond in strict JSON’ instruction usefully replaces the current grammar. »

Avoid using notes to store prompt content « to be enabled later ». For that, the prompt itself has the **WIP** marker, or you can duplicate-and-disable. Mixing prompt prose and notes in the same field makes maintenance painful.

21.3 — POO²L sections

A section is a named fragment of the content, opened by a `@@@section <name>` declaration. Everything that follows until the next section declaration (or the end of the prompt) is the section’s content — no closing marker is required, though an explicit `@@@end-section` exists for authors who want to bound a section unambiguously. At render time the declaration line becomes a **markdown heading** : the section’s name titles its own content, so the rendered prompt still says which part is which.

Their value only appears with inheritance. A section can be **inherited**, **overridden**, or **inserted by reference** through `include parent.section`.

A section declaration is built from independent axes — each optional, freely combinable (with one exception : `virtual` and `final` cannot coexist) :

1. **Presence toggle** (optional) : `?` or `!` — the SECTION TOGGLE — glued to the keyword `section`, never to the name.
2. **Inheritance modifier** (optional) : `virtual` (abstract — descendants must implement) or `final` (a FINAL SECTION — sealed, descendants must not redeclare), placed **after** the `section` keyword.
3. **Placement** (optional) : one of `first`, `last`, `before X`, `after X` after the name, to position the section in the rendered order relative to its siblings.

The general shape is therefore — sigil first, toggle glued to the keyword, modifier after it, no numbering index :

```
@@@section[?|!] [virtual|final] <name> [placement] [: <description>]
```

The three toggle variants of the keyword itself :

- **`section <name>`** — ordinary section, always rendered. No dialog. Children that redeclare the same name override it.
- **`section? <name> : <description>`** — user-toggled section, default **OFF**. At instantiation the renderer shows a dialog ; the user must check the box to include the section. The description is the dialog text.
- **`section! <name> : <description>`** — user-toggled section, default **ON**. Same dialog ; the user must uncheck to exclude it.

The description is mandatory for `?` and `!` because it is the text the user reads in the include/exclude dialog. `?` and `!` are mutually exclusive. The presence toggle is **purely about user choice at instantiation** — it does not affect scoring, weight, or any other application-level concern.

The two inheritance modifiers extend a section with the *abstract* and *sealed* contracts familiar from class-based OO — the modifier comes right after the `section` keyword:

- **`section virtual <name>`** — declares the slot but provides no body; some descendant must fill it. A `virtual` section's source must be empty between its marker and the next section (the engine rejects a VIRTUAL SECTION that carries a body).
- **`section final <name>`** — declares and seals the section; no descendant may redeclare it (no body change, no toggle change, no description change).

The same *abstract* / *sealed* contract exists at the whole-template level, but there it is expressed as a modifier on the prompt header — `@@@template virtual <id>` and `@@@template final <id>` — not as a standalone `@@@virtual` / `@@@final` line.

The UI surface lives in the **Structure** tab of the editor, which appears from ceiling level L1 onwards (`pool_max_level`, Options → Editor → Input) — sections are L1, so that is where the tab starts. No feature toggle governs it. The panel lists the sections declared in the current prompt and those inherited from the chain; you can create, rename, delete, and reorder. Section declarations that carry `virtual` or `final` are flagged in the panel so you can tell at a glance which slots must be implemented and which are sealed.

21.4 — What the reader of a rendered prompt sees

The structure is not lost on the way out. A rendered prompt opens with its own title — its name, then its identifier in parentheses, so a re-read of a session says which prompt produced the text — and each section carries its name as a heading :

In the prompt	In the rendered text
`@@@section objectif`	`## Objectif`
`@@@section objectif : But de l'analyse`	`## Objectif` — the description is not rendered
`@@@section steps.lecture : Lecture`	`### 1. Lecture`

Three points to keep in mind while writing :

- The heading text is the section's **name**, first letter capitalised, the rest untouched (`objectif` renders as `Objectif` ; for a subsection, the child segment : `steps.lecture` renders as `Lecture`). The **description** after the colon never reaches the rendered text : it is the prose the include/exclude dialog shows for a `?` / `!` section — mandatory there — and a note for the author anywhere else. A readable heading is therefore a matter of choosing the name.
- Sub-sections are **numbered** — `### 1.` , `### 2.` — in the order they render. An item switched off leaves no gap in the count, and a numbered list written *inside* an item keeps its own plain `1.` , `2.` without any confusion between the two.
- The levels `#` , `##` and `###` belong to the renderer inside a section. Once a prompt is cut into sections, typing one of them in a section's prose is a syntax error, caught at save time — the fix is usually to make that line a section of its own, whose name becomes the heading it was imitating. A prompt that is **not** cut into sections keeps its Markdown as written : naming a prompt

does not structure it, so an ordinary Markdown document remains a valid prompt and renders exactly as it stands. Deeper levels are free : `####` and beyond are ordinary text and sit naturally under the emitted `###`. A literal `#` that is not a heading — `C# rules`, `issue #12`, `#tag` — is untouched, and `@@@verbatim` blocks are left exactly as written when a literal `# heading` really is the point.

21.5 — Sections and ordering

When a parent declares several sections in a row, the order in which they appear in the rendered output is the order of their declaration. A child can override the order by redeclaring the relevant sections in the desired order ; sections that are not redeclared keep their relative position.

For finer control the declaration accepts a **placement** keyword after the name — `first`, `last`, `before <name>` or `after <name>` — which pins a section relative to its siblings regardless of where it is declared. That is the language mechanic for ordering ; there is no numeric index in the grammar. Prefixing names with digits (`01_introduction`, `02_context`) is a pure naming convention — the engine does not parse or reorder on those numbers — but it keeps the Sections panel readable.

21.6 — Notes vs sections vs comments

To recap :

- A **note** is editorial metadata about the prompt, never rendered.
- A **section** is a fragment of prompt prose, rendered but manipulable through inheritance.
- A POO²L **comment** is text ignored at render time. It comes in several forms : a line comment `@@@-- ...`, a bounded inflowing comment `@@@(-- ... )`, and a block `@@@comment ... @@@end-comment`. Comments annotate a complex directive ; they are not meant to carry structured editorial text the way notes are.

Picking the right mechanic for the intent makes templates much more readable over time.

Chapter XXII — Rendering and instantiation

Rendering is the operation that turns the source content of a prompt — prose intermixed with POO²L directives — into plain text ready to be sent to a chatbot. *Instantiation* is the precise moment at which that rendering happens, with the parameters and constants finally resolved. Telling the two apart is useful because rendering can occur in two places (the dashboard for preview, the extension for injection) and the pipeline has several stages that are worth pulling apart.

22.1 — The render preview

The **Live preview** entry in the editor opens a panel that shows the rendered prompt *as it would be sent to the chatbot if the injection happened now with the current values*. (Do not confuse it with `Ctrl+R`, which is the editor's find-and-**replace** shortcut, not a preview toggle.) The panel is read-only ; its purpose is to visually check the result before validating a change or sharing the prompt.

The preview applies every stage of the rendering pipeline *except* the parameter dialog — since no end user is around to type, parameters that carry a default take that default value. A parameter with no default cannot be resolved : the render reports it as unresolved (`UNDEFINED_PARAMETER`) and the panel falls back to showing the raw source, with the `@@@=` reference left visible. What you see is therefore an honest approximation, not an actual output.

22.2 — The rendering pipeline

Rendering is delegated in full to the POO²L engine (the Rust `pool-core`, reached through its bindings). IziBuddy does not re-implement any of the grammar ; it hands the engine the prompt source, a registry of the templates it may reference, and a context holding

the system-variable values and the user-supplied parameters. The engine then produces the rendered text. Its work covers, conceptually :

- **Inheritance and include resolution.** The engine resolves the `inherits` chain and expands every `@@@{include ...}`, pulling in referenced templates and sections and accounting for overrides.
- **Substitution.** Every `@@@=name` reference — whether it names a parameter, a constant or a SYSTEM VARIABLE — is replaced with its value in a **single pass**. There is no separate ordered stage per kind ; the engine resolves each reference once, from the values it was given.
- **Section handling and cleanup.** Toggled sections are kept or dropped, section markers and comments are stripped, and the result is assembled as plain text.

The result is a string, returned together with any warnings and the list of effective attachments. That string is what the extension injects into the chatbot's editor, or what the REST API returns when a client calls the render endpoint. A reference `@@@=X` that resolves to nothing is **not** silently ignored : the engine raises a blocking `UNDEFINED_REFERENCE` diagnostic and the API refuses to expose the best-effort output.

22.3 — System variables

System variables are referenced with the substitution form `@@@=NAME` . There is only one spelling — the double-brace `{{NAME}}` form some templating systems use does **not** exist in POO²L and is never substituted. Chapter XLI catalogues the available variables.

The values are computed by the backend at render time and passed to the engine as the `context.system` map :

Variable	Description	Example
`@@@=DATE`	ISO-formatted date	`2026-05-17`

Variable	Description	Example
`@@@=TIME`	Time	`14:32:00`
`@@@=DATETIME`	Date and time	`2026-05-17 14:32:00`
`@@@=YEAR`	Year	`2026`
`@@@=MONTH`	Month	`05`
`@@@=DAY`	Day	`17`
`@@@=WEEKDAY`	Day of the week	`Sunday`
`@@@=USER`	User name	`jean.dupont`
`@@@=LANG`	User language	`fr`
`@@@=GROUPS`	Groups the user belongs to	`dev, qa`
`@@@=GROUPS_ADMIN`	Groups the user administers	`dev`

`@@@=WEEKDAY` is sensitive to the user's language — you get `Dimanche` or `Sunday` depending on the current user's active language.

22.4 — One substitution form

There is nothing to choose between : `@@@=NAME` is the single, canonical way to reference a system variable, a constant or a parameter in the body. It is part of the grammar, so the same form works everywhere the engine reads — including as an argument inside a wider directive. The engine substitutes each reference once, in a single pass ; a value that itself contains an `@@@=` reference is **not** re-scanned.

22.5 — The render endpoint

For programmatic clients that want to fetch a rendered prompt without going through the extension, the dashboard exposes dedicated endpoints :

- `GET /api/templates/render/{id}` — renders the prompt with its defaults.
- `POST /api/templates/render/{id}` — accepts a JSON body holding the parameters and to-be-supplied attachments, and returns the rendered prompt as plain text, plus the list of effective attachments.

The extension and the API render through the **same engine**, so there is no behaviour drift between the two channels : both hand the same source and context to `pool-core` and get the same text back.

22.6 — Adding a system variable

Adding a new system variable is an operator task, not a user task. Because the engine owns the grammar, the reserved name must first be added to `pool-core` (the system-variable list in `parse.rs`) and the bindings rebuilt ; the backend then computes the value and adds it to the `variables` dictionary passed to the render endpoints of `src/api/templates.py`, where the adapter routes reserved names into `context.system`. The new variable is available in every prompt once the engine is rebuilt and the server restarts.

The procedure is detailed in Chapter XLI (manual side) and in the `CLAUDE.md` file at the root of the project.

Chapter XXIII — Reading map

You have just read seven chapters (XVI through XXII) that describe how the dashboard exposes POO²L : opening the parameters dialog, wiring two templates together through inheritance, watching the render preview update. The remaining chapters of this manual extend that knowledge in different directions — sharing, extension behaviour, administration, deployment. This short chapter is a within-book index : for each common question, jump to the chapter that answers it.

23.1 — Map of common questions

On variables and substitutions :

- « What is the exact syntax of a system variable — the substitution `@@@=NAME` ? » → Chapter XLI.
- « In what order are substitutions applied at render time ? » → Chapter XXII.

On parameters :

- « How do I declare a parameter from the UI ? » → Chapter XVII.
- « How are parameter values entered at injection time, and what is a parameter set ? » → Chapter XXXV.

On constants :

- « How do I declare a constant, and how is it rendered ? » → Chapter XVIII.
- « How do I refactor a constant’s name without breaking my templates ? » → Chapter XVIII.

On sections and notes :

- « What is the difference between a note and a section, and how do they appear in the render ? » → Chapter XXI.

- « How do sections behave under inheritance — are they overridden, inherited, included ? » → Chapter XXI.

On attachments :

- « How do I attach a file to a prompt, and how is it transmitted to the chatbot ? » → Chapters XIX and XXXIV.

On inheritance and includes :

- « How do I set a prompt to inherit from another ? » → Chapter XX.
- « How does the editor warn me about cycles or broken references ? » → Chapters XVI and XX.

On rendering :

- « What exactly happens between the moment I click *Inject* and the moment the chatbot receives text ? » → Chapter XXII.

On containers :

- « What is the difference between a folder and a collection ? » → Chapter XI.

On sharing and groups :

- « Should I share a prompt or duplicate it ? What changes when I move a prompt to a group ? » → Chapters XXIV and XXV.

23.2 — A practical rule

If your question starts with *« how do I… »* and is about a button, a dialog, or a workflow, the answer is in one of the chapters listed above. If your question starts with *« what happens if… »* and is about an edge case of the engine, look first at Chapter XXII (rendering) or at the chapter that introduced the feature you are stretching.

The chapters do not need to be read in order. Chapters XVI–XXII form a coherent block on the smart editor and POO²L surface ; the rest can be approached on demand, when the corresponding need arises.

Partie 4

Collaboration

Chapter XXIV — Sharing vs duplicating

You have written a prompt that works well and you want colleagues to benefit from it. Two options are available, which look similar but are fundamentally different : *share* the prompt with a group, or let a colleague *duplicate* it into their own space. This chapter explains what each option does, what it implies down the line, and how to choose.

24.1 — Sharing : exposing a prompt to a group

Sharing in IziBuddy is done **towards groups**, not towards individual users. There is no « share with Alice » and no « Shared with me » inbox — the unit of sharing is the group (Chapter XXV). You keep ownership of the prompt ; the members of the groups you share it with gain read access to it.

Concretely, sharing sets the list of groups a prompt is exposed to. The members of those groups then :

- See the prompt in their dashboard and use it from the extension exactly as if it were their own — search, injection dialog, personal parameter sets stored server-side per user.

They do not become owners : they cannot edit the content, parameters, constants or sections. When you modify the prompt after sharing, they see the change — there is a single source of truth. When you remove it from a group, it disappears from those members' view.

Two rules constrain who can share what :

- **Only the author can change a prompt's sharing.** A member who merely sees a shared prompt cannot re-share it.
- **You can only share into groups you administer.** The picker offers the groups where you are a group admin, obtained from `GET /api/groups/me/admin`.

The REST endpoint is `POST /api/templates/share-groups/{id}`, with the target group ids in the body (`{group_ids: [...]}`). It *replaces* the full set of groups the prompt is shared with. To remove a single group, `POST /api/templates/unshare-group/{id}` with that group's id.

24.2 — Duplicating : an independent personal copy

Duplicating turns a prompt you can see — typically one shared with you through a group — into an *independent copy* in **your own** space. You become the full owner of the copy. You can :

- Edit the content, parameters, notes — everything.
- Rename it, move it to your own containers, apply your own markers.
- Delete it, or share it (into groups you administer) in turn.

Once duplicated, the two instances lead separate lives. A change on the original no longer appears on your copy, and vice versa. Duplication is, by nature, a *fork*.

The matching endpoint is `POST /api/templates/duplicate/{id}`. It takes no recipient — it always copies into the caller's own space. The copy starts fresh : it does not carry over the original's version history or its group sharing.

24.3 — Choosing : the matrix

Three questions settle the choice between sharing and duplicating :

Must colleagues be able to modify the prompt ? If yes, they duplicate it. Group sharing is read access only ; there is no « share in edit mode » with an individual user.

Must the prompt evolve in coordinated fashion for everyone ? If yes, share it with the group. A change on the owner's side immediately benefits every member. If everyone duplicates, you create N divergent copies that have to be kept in sync by hand.

Is the prompt a template to be customized ? If yes, duplication is the natural move : each colleague duplicates the prompt and adapts their copy. Flagging the original **Template** makes that intent explicit.

24.4 — The governance nuance

Beyond the technical choice, one governance point matters. Group sharing leaves an explicit, revocable trail : the prompt's sharing record lists exactly which groups can see it, and you can pull a group out at any time. Duplication leaves no such trail — the copy becomes the duplicator's property and you no longer have any rights over it. That is by design, but it is worth a moment of thought before duplicating a prompt that holds sensitive information.

Note that the quality **score** shown on a prompt (Chapter XXIX) is computed from the prompt's own content, not accrued from usage — so nothing « bubbles up » from a duplicated copy to the original.

24.5 — Where to see and change sharing

The **Sharing** panel of the *Template properties* dialog lists the groups a prompt is currently shared with. From there you add or remove groups (again, limited to those you administer). Removing a group revokes access for its members ; the prompt itself is untouched. The same information is available programmatically through `GET /api/templates/sharing/{id}`, which returns the prompt's author and the list of groups it is shared with.

Chapter XXV — Groups

A group is a **named set of users**. It is the unit IziBuddy uses to distribute access to templates : you share a prompt *with a group* (Chapter XXIV), and the group's members gain read access to it. A group is not a separate storage space that owns its own templates, tags or Git repository — templates always belong to the user who created them ; a group is simply the audience a prompt can be exposed to.

This chapter covers what a group is made of, the two levels of membership, how groups nest, and how the **My groups** source works. Invitations — the onboarding path for new users — are the subject of Chapter XXVI.

25.1 — What a group is

A group has :

- An **id**, a **name** and a **description** ;
- A **list of members** ;
- One or more **group administrators** — a subset of the members who can manage membership ;
- Optionally, **sub-groups** : a group can contain other groups, and membership is transitive along that tree (an « effective » member of a group is anyone directly in it or reachable through a sub-group).

The group id is used throughout the REST API as `/api/groups/{group_id}/...`. The UI shows the name ; the id is stable behind it.

There is no notion of a group « scope » you switch the dashboard into, no per-group banner, and no per-group tag or container tree. Those are properties of your personal space ; a group only governs *who can see* the templates you share with it.

25.2 — The two levels of membership

A user's relationship to a group is one of exactly two levels :

- **Member** — belongs to the group. Sees and uses the templates shared with the group.
- **Group admin** — a member who, in addition, can manage the group's membership : add and remove members, and promote or demote group admins. A group admin is also who is allowed to *share a prompt into that group* (only groups you administer appear in the sharing picker).

There is no « editor » or « viewer » role, and no per-member permission matrix. The distinction is binary : member, or member who also administers.

Two safety rules apply to group admins : a group admin cannot remove **themselves** from the group, and cannot demote themselves from the admin role. Only a system administrator can do either — this prevents a group from being left with no admin by accident.

25.3 — Creating a group

Group creation is **reserved to system administrators**. The endpoint `POST /api/groups` is guarded by the admin dependency ; a regu-
lar user cannot create a group. The request carries the group's `id`, `name` and `description` — there is no visibility setting (no « private » / « public » directory), because a group is an access list, not a discoverable space.

Once a group exists, a system administrator or an existing group admin populates it (Chapter XXVI covers inviting brand-new users ; existing users are added directly through `POST /api/groups/{group_id}/members/{username}`).

25.4 — Managing membership

From the group's member view, a system administrator or a group admin can :

- **Add a member** — `POST /api/groups/{id}/members/{username}`.
- **Remove a member** — `DELETE /api/groups/{id}/members/{username}` (not yourself, if you are a group admin).
- **Promote to group admin** — `POST /api/groups/{id}/admins/{username}` (the user must already be a member).
- **Demote a group admin** — `DELETE /api/groups/{id}/admins/{username}` (not yourself, if you are a group admin).

The member list returned by the API sorts group admins first and marks each entry with an `is_group_admin` flag, so the UI can distinguish the two levels at a glance.

25.5 — The « My groups » source

The **Sources** menu of the dashboard, under **My groups**, lists the groups you belong to and lets you bring their shared templates into view alongside your own. It is a *source filter*, not a scope switch : your personal templates remain visible, and creating a prompt still creates it in your own space.

The `sourcesGroups` feature (*Organization* category) controls whether the **My groups** section is shown. It is active in the Advanced and Expert presets ; turning it off hides the section without changing any membership — it is purely a presentation choice.

25.6 — Nested groups

A group can contain sub-groups, forming a tree (`GET /api/groups/{id}/subgroups` for the direct children, `GET /api/groups/{id}/tree` for the full recursive tree). Membership is transitive : asking for a group's members with `scope=effective` returns everyone reachable through the sub-group tree, LDAP-style, while

`scope=direct` returns only the users added to that group itself. This lets an organization model a department that contains several teams without re-listing every person at each level.

25.7 — Deleting a group

Deleting a group is a system-administrator action (`DELETE /api/groups/{id}`), and it is refused while the group still has members: you empty the group first, then delete it. Because a group owns no templates of its own, deleting it does not destroy anyone's content — it only removes the access list. Templates that were shared with the group simply lose that group from their sharing record.

Chapter XXVI — Invitations

An INVITATION is IziBuddy's mechanism for **onboarding a new user onto the instance**. It is not a per-group membership request : an invitation is created at the instance level by a **system administrator**, it targets an e-mail address that does not yet have an account, and it optionally pre-assigns the future account to one or more groups. This chapter describes how invitations are created, validated, consumed and revoked.

26.1 — Who can invite, and what an invitation carries

Creating invitations is reserved to system administrators — the endpoint `POST /api/invitations` is guarded by the admin dependency. An invitation carries exactly three things :

- The recipient's **e-mail** address.
- An optional list of **groups** the new account will join once the invitation is used.
- An **expiry**, in days (`expires_days` , default **7**).

There is no per-invitation role and no free-form message field — the model is deliberately minimal. An invitation exists to let a specific e-mail address create an account and land in the right groups.

Because invitations onboard *new* users, creation is refused if a user with that e-mail already exists : the manager returns « A user with this email already exists ». For someone who already has an account, use group membership directly (Chapter XXV) rather than an invitation.

26.2 — The token

Creating an invitation generates a URL-safe random token (`secrets.token_urlsafe(32)`) and stores the pending invitation server-side, keyed by that token, with its e-mail, its groups and its expiry date. The token is the sole access path : knowing the recipient's e-mail is not enough to consume an invitation.

Creating an invitation does **not** by itself send an e-mail. The single-invitation endpoint returns the token to the administrator, who conveys the registration link out of band. Automated e-mail sending exists only in the batch path described below.

26.3 — Accepting an invitation

The recipient completes registration in one step. The client first validates the token — `GET /api/invitations/validate/{token}` returns the invitation's e-mail, its groups and its expiry if the token is valid — then posts the new account :

```
POST /api/invitations/register
{ "token": "...", "username": "...", "password": "..." }
```

On success the account is created with the invitation's e-mail, the user is added to the pre-assigned groups, the invitation is marked used (its token can no longer be reused), and an access token is returned so the new user is signed straight in.

26.4 — Expiration

An invitation is valid for **seven days** by default ; the value is set per invitation through `expires_days`. Past the expiry the token no longer validates and the registration attempt is refused. There is no self-service « extend » action — to give someone more time, an administrator issues a fresh invitation.

26.5 — Revoking

The only lifecycle action an administrator has over a pending invitation, besides letting it expire, is to **revoke** it : `DELETE /api/invitations/{token}` invalidates the token immediately. There is no Accept/Decline exchange on the admin side, no « Resend » and no « Extend » — the surface is Create, list, revoke, and (housekeeping) cleanup.

Administrators can review invitations through `GET /api/invitations` (all, including expired) and `GET /api/invitations/pending` (valid only), and purge stale entries with `POST /api/invitations/cleanup`.

26.6 — Bulk onboarding

To bring in many users at once, an administrator imports a CSV through `POST /api/invitations/batch-import`. The file has one recipient per row :

- Column 1 : **e-mail** (required).
- Column 2 : **groups** (optional, comma-separated).

Each row produces an independent invitation with its own token. The batch path is the one place where IziBuddy can send the invitation e-mails for you : the `send_emails` flag, when set, routes each invitation through the deployment's SMTP connector (Chapter LVIII), and a per-address failure does not stop the others. When `send_emails` is off, the import just creates the invitations and the administrator distributes the links. A summary reports how many were imported out of the total, along with any parse errors.

Chapter XXVII — Group templates and the instance Git repository

Two topics that often get conflated deserve to be kept apart : how templates reach a group, and how an instance can mirror its content to a Git repository. They are independent — group sharing governs *who sees* a prompt ; Git governs *how the file backend is backed up*. This chapter covers both, and where Git applies (which backends are GIT APPLICABLE).

27.1 — How templates reach a group

A prompt always belongs to the user who created it. It becomes visible to a group when its author **shares it with that group** (Chapter XXIV). There is no separate « group scope » in which templates are created and owned by the group ; sharing is an access grant on a personally-owned prompt.

The practical consequences follow directly :

- Every member of a group the prompt is shared with sees it in their dashboard and can use it from the extension, read-only. Group admins are the members allowed to *share templates into* the group in the first place.
- To take a personal copy of a shared prompt, a member **duplicates** it (Chapter XXIV) ; the copy lands in their own space and is theirs to edit.
- The prompt’s version history (Chapter XXVIII) is visible to everyone who can see the prompt. The history records the sequence of saved states ; it does not attribute each version to a named author — there is no per-version author field.

27.2 — When Git is applicable

Git in IziBuddy assumes the **file-based** storage backend — the default when no `[database]` section is set in `izibuddy.ini` (Chapter XLVIII §database). On installations backed by a real database (PostgreSQL, MySQL, MongoDB, etc.), the database is itself the shared source of truth across backends, and the entire Git layer is hidden from the UI : the *Git Settings* entry of the header menu disappears, and the `/api/git/*` endpoints return HTTP 409 *Git not applicable for this storage backend*.

27.3 — The instance Git repository

Git backup in IziBuddy operates on a **single, instance-level repository** — one repo per install, at `~/.izibuddy/templates.repo`, initialized at startup. It is not scoped per group or per user : there is no separate mirror for each group and no per-user repo. The one repository holds the instance's file-backend content in clear-text JSON, which gives you :

- **Durable, off-site backup.** The content lives in a Git repo you administer — on GitHub, GitLab, an internal forge, or a self-hosted server. An IziBuddy outage does not erase it.
- **Reviewable history in Git terms.** Any tool that speaks Git can consume the repo — CI, audit scripts, external diff tools.

Configuration lives in `izibuddy.ini`, not in a per-group UI dialog : there is no « Git tab » on a group, no encrypted-credentials form, no per-group branch selector, no *Test connection* / *Sync now* buttons and no batch queue. The relevant keys — `pull_strategy` (`ff-only`), `sync_on_startup` , `sync_on_read` — are documented in Chapter XLVIII §git, and the connector itself (authentication, repo layout, file format) in Chapter LIX.

27.4 — Recovering from the repository

Because the repository serializes the content with stable identifiers, it doubles as a recovery and migration path : point a fresh file-backed install at the same remote and let it synchronize, and the content rebuilds itself with inheritance links and includes intact. This is also how you clone an instance's content onto another machine — for example, moving from a test install to production.

27.5 — Multi-backend topologies

A single IziBuddy install rarely lives alone — teams end up running two or more backends (a dev machine, a laptop, a production host, a personal mirror). Three topologies are common, and the relevance of Git depends on which one you are in.

- **Single backend.** One process, one storage path. Git here is purely an *off-site backup* mechanism : nothing reads from it, only the sync writes to it. `sync_on_startup` and `sync_on_read` (Chapter XLVIII §git) add no value — leave them off.
- **Multi-backend sharing one database.** Two or more backends configured with the same `[database]` section. The database itself synchronises everything across instances ; Git is *not applicable* and the UI is hidden. This is the recommended topology for any team beyond a single user.
- **Multi-backend sharing one Git remote (file backend only).** Two file-backed backends pointing at the same remote — the canonical example is one user on two machines, or a personal laptop mirrored to a small cloud VM. Auto-commit and Auto-push cover the write path. For the *read* path, see the `sync_on_startup` and `sync_on_read` keys in Chapter XLVIII §git : without them, the second backend can read stale content until the next manual *Pull*.

Across all three topologies the ownership conventions hold : one user is the single owner of their templates, templates shared with a group are read-only on the members' side, and concurrent edits of the same prompt from two backends remain the writer's responsibility — IziBuddy will not merge competing bodies.

Chapter XXVIII — Versioning and diff

Every prompt lives across time. You write it in five minutes, you rework it two weeks later, you come back to it three months on after changing your mind. IziBuddy records each of these stages as a *version* — a snapshot of the prompt at a given moment, accessible, comparable, restorable.

This chapter describes the versioning system : when a version is created, how to see a prompt's history, how to compare two versions, how to roll back. The pivot feature is `versioningDiff`, active in the Advanced and Expert presets.

28.1 — When a version is created

A version is created when you **save** the prompt — the **Save** button or `Ctrl+S` — and the content has actually changed since the last version. The server deduplicates by checksum : saving with no modification does not add a version, so the history stays free of redundant entries. There is no background auto-save and no « percent-changed » threshold that snapshots behind your back ; saving is what records a version.

The general rule : a version is simply the history's unit ; you use it to navigate, not to trigger.

28.2 — The version dropdown

In the smart editor, versioning is surfaced through a **dropdown** in the editor toolbar — a 📌 trigger showing the current version number, which opens a searchable list of the prompt's versions, newest at the top. Each entry shows :

- The **version number** (an `x.y` value ; a `.0` marks a major version, a `0.y` a work-in-progress one).
- The **date** the version was saved.

- Its optional **note** — a short free-text label the author can attach to a version (used, for instance, to mark a milestone).

A search box at the top of the dropdown filters the list by version number or note. If unsaved edits exist, a **working version** entry sits at the top of the list. Selecting a version loads it for viewing ; the dropdown is the single control for navigating and restoring history — there is no separate tab, no per-version author column, and no character-delta indicator.

28.3 — The diff

The diff is the central tool for understanding what changed between two versions. The **Diff** viewer is a **side-by-side** display — two parallel columns, the old version on the left, the new on the right, differences aligned on the same line. There is no single- column « unified » mode.

What varies is the **granularity**, toggled between **Line** and **Word** by two buttons in the viewer's header (the preference is remembered locally). Line mode highlights whole changed lines ; Word mode refines the highlight down to the changed words within a line.

Server-side, the diff is computed by `POST /api/diff/compare` on the **POO²L source** of the two versions — not on the rendered output — so you compare what you actually edit. See the code in `src/api/diff.py` for the details.

28.4 — Comparing two arbitrary versions

By default, the diff compares a selected version to the current one. `POST /api/diff/compare` takes the two texts to compare — `text_old` and `text_new` , which the client fills from each version's stored source — so you can also diff two *historical* versions against each other : pick the older one as the reference (`text_old`) and the newer one as the target (`text_new`).

This answers questions like « what did I change between Tuesday and Thursday ? » — compare Tuesday evening's version against Thursday evening's and the diff shows you.

28.5 — Rollback

Restore from the version dropdown brings the prompt's content back to the state of an earlier version. Three important nuances :

- Restoring *does not delete* intermediate versions. They remain in the history. Restoring creates *a new version identical to the restored one*, which becomes the current version. You can therefore « undo the restore » by restoring the previous state.
- Content, parameters, constants, sections, and notes are restored. Attachments stored server-side are not brought back automatically : if you restore a version that referenced an attachment since deleted, the prompt shows a warning.
- Sharing (the groups a prompt is exposed to) is not affected by a restore — it is not part of the versioned content.

There is no dedicated « rollback » route : restoring is done by `POST /api/versions/{template_id}/copy`, which copies the chosen version's content forward as the new current version and records the operation in the history.

28.6 — Labeling a version

For versions that mark a milestone — the officially published version, the version validated by review — you can attach a short **note** to the version (« v1.0 », « before May 2026 redesign »).

Versions with a note display it in the dropdown, and the dropdown's search box matches on notes as well as version numbers, so a well-noted history stays navigable.

28.7 — Versioning and shared templates

When a prompt is shared with a group, its version history is visible to every member who can see the prompt. The history is a sequence of saved states with their dates and notes; it does **not** stamp each version with the name of the user who saved it — there is no per-version author field, so treat the history as a record of *what* changed and *when*, not *who*.

28.8 — What `versioningDiff` controls

The `versioningDiff` feature (Advanced and Expert presets) governs the **UI** only: with it off, the version dropdown and the diff viewer disappear from the editor. It does **not** turn off server-side versioning — versions continue to be recorded on save regardless of the feature. Re-enabling `versioningDiff` therefore brings the full history back into view; nothing was lost while the UI was hidden.

Chapter XXIX — Scoring

Scoring in IziBuddy is a **computed quality score** for a prompt — not a user rating. There are no stars to click and no votes to tally : the score is calculated from the prompt's own content by an analysis engine, on a scale of **0 to 10**. It tells you, over time, which templates are well-structured and which were written one day and never firmed up. Scoring is *optional* — many small deployments live without it — but as a library grows it becomes a precious signal for periodic housekeeping.

29.1 — Showing the score

Scoring is surfaced by the feature `viewScoring` (category *Display*), active in the Standard, Advanced and Expert presets. When it is on, the computed score appears next to templates — a **Score** column in the table view and a badge on the tiles — and the prompt's properties expose the detail of the calculation. Turning `viewScoring` off simply hides those indicators ; it changes nothing about the templates themselves.

29.2 — What the score measures

The score is produced by IziBuddy's scoring engine (`src/scoring/engine.py`), reached through `POST /api/scoring/score` . It is a **deterministic** function of the prompt's structure — the same content always yields the same score. Two ingredients drive it :

- **Section coverage.** The engine detects the canonical sections a prompt contains (role, context, objective, task, steps, constraints, rules, examples, output format, tone, audience, and so on) and checks them against the sections a given *goal* expects. A prompt missing a critical section for its goal scores poorly.

- **Coherence.** Beyond mere presence, the engine runs coherence checks — weighted dependencies and synergies between sections (for example, a stated task should be paired with an output format, and with constraints). Satisfying these lifts the score ; breaking a strict dependency pulls it down.

There is no stored history of individual ratings, no per-user vote, and no aggregation of votes — the score is recomputed from the content, not collected from people.

29.3 — Goals and standards

The engine scores a prompt *relative to an intent*. A **goal** (`BUT`) — such as `PRODUIRE` (produce) or `ANALYSER` (analyse) — declares which sections matter and which coherence checks apply. The available goals come from `GET /api/scoring/goals`. Because a prompt meant to *produce* has different structural needs than one meant to *analyse*, the same content can score differently under different goals.

A **standard** can further restrict the set of sections considered active. Together, goal and standard define the yardstick the content is measured against ; the score is the fraction of that yardstick the prompt meets, blended with its coherence fraction and mapped onto the 0–10 scale.

29.4 — The badge scale

The numeric score is summarized by a badge, so the list stays readable at a glance. The bands are fixed :

Badge	Score	Meaning
⚫	below 1	Not usable — minimal structure absent.
🔴	1 to 3	Critical element(s) missing for the goal.
🟠	3 to 6	Partly works — important sections or coherences missing.

Badge	Score	Meaning
🟡	6 to 7	Usable but not robust.
🟢	7 to 8	Good structure, reliable in most cases.
💚	8 to 9	Reliable and reusable.
💚✨	9 and up	Production-grade — structure and coherence at their peak.

A prompt that is missing a section flagged *required* for its goal is forced to 🔴 regardless of the rest, because the gap is disqualifying for that intent.

29.5 — The Note flag and filter

The filter `filterNoScoring` (category *Filters*) is **not** a « never rated » filter — there are no ratings to lack. It selects templates that carry the manual **Note** flag : templates an author has explicitly marked as *not meant to be scored* (a scratch prompt, a fragment, a container-like helper). The flag lets those templates opt out of the quality signal so they do not skew a housekeeping pass.

Note uses a coloured document icon, not a grey or crossed-out microscope. Use `flag:note` to find notes and `NOT flag:note` to exclude them. The stored field remains `no_scoring`, and older searches still work. A Note is not a new container type.

When you *do* run a housekeeping pass, sort or filter by the computed score itself to surface the weak templates, then decide each one's fate : firm up its structure, mark it **Disabled**, or send it to the trash.

29.6 — Detecting sections

The same machinery that feeds the score is available on its own through `POST /api/scoring/sections`, which returns the sections the engine detected in a piece of content, whether each is optional,

whether it is included by default, and a short preview. This is what lets the editor show *why* a score is what it is — which sections count as present and which are missing — rather than a bare number.

29.7 — Where scoring is configured

Scoring is a content-analysis feature, computed on demand; there is no server-side collection of votes to switch on or off, and no per-group breakdown of individual ratings, because none of that exists. What an operator controls is whether the `viewScoring` indicator is part of a preset (Chapter XXXVII) and whether the feature is available to users at all.

Partie 5

The browser extension

Chapter XXX — Anatomy of the floating panel

IziBuddy's browser extension manifests as a floating panel, injected into the chatbot page you are using. This panel is your gateway to your templates from Claude, ChatGPT, Grok, Gemini (Google AI Studio) or DeepSeek : no need to return to the dashboard, no need to copy-paste. You deploy it, you pick a prompt, you send it into the chatbot's editor. This chapter describes every piece of the panel and the logic that articulates them.

30.1 — The compact launcher

In its collapsed state, the panel shows only a small launcher — a discreet **rectangular mini-panel**, anchored by default in the **top right** of the chatbot page. It is intentional : the extension must not interfere with your conversation with the AI. You click the launcher to deploy the panel ; you click it again, or use the keyboard shortcut, to collapse it.

The launcher's position is remembered in `localStorage` (`inziderPanelPosition`, default `{ top: 20, rightOffset: 20 }` — top right). You can drag it to reposition it anywhere on the page. Its status indicator reflects the connection state to the dashboard : green if you are authenticated, gray otherwise — a discreet visual signal that saves you from digging into the configuration to check the auth state.

A tip many people miss : the extension exposes a global keyboard shortcut `Ctrl+Shift+I` (`Command+Shift+I` on Mac) that toggles the panel without touching the mouse. The shortcut is declared in the extension's manifest (`src/extension/manifest.json`, section `commands._execute_action`).

30.2 — The deployed panel

Once open, the panel occupies a width of a few hundred pixels, anchored near the top-right corner of the page (by default). It is **isolated from the host page's DOM** through a Shadow DOM : the chatbot page's styles do not contaminate those of the panel, and vice versa. This is what allows IziBuddy to have the same look whether opened on Claude, on ChatGPT or on Gemini. The orchestration of this isolation lives in `src/extension/content/content-panel-core.js` (see `loadShadowCSS`, `iziGetById`, `iziQuery`).

The panel is structured vertically into three zones :

- A **title bar** at the top, with the IziBuddy name, the authentication indicator and the quick buttons (collapse, expand).
- Below it, the **content of the current screen** — most of the time the Templates view, which fills the rest of the height.
- Contextual **navigation** to the secondary screens (Configuration, Profiles, About), reached through buttons rather than a permanent tab strip.

This anatomy is deliberately flat. No cascading menus, no contextual toolbar ; a single screen is visible at a time.

30.3 — Navigation between screens

Navigation is **contextual**, not a fixed tab strip. The **Templates** view is the default screen and shows no navigation banner at all — the search field and the mini-config button sit directly in the content. From there :

The **Templates** view is the heart of the extension : it lists the templates available from your account, lets you search, fill their parameters and inject them into the chatbot's editor. The whole of Chapter XXXII is devoted to it.

The **Configuration** screen holds the behavior preferences (auto-collapse, dark mode, number of quick buttons) and the login state. It is reached from the Templates view ; a back arrow returns you to the templates. Chapter XXXI explains the connection.

The **Profiles** screen selects a display profile — a visual theme, an accent color, a quick-buttons layout and a local Stash. It is **not** a top-level routable tab : it is opened from the Configuration screen. Detailed in Chapter XXXIII.

The **About / Login** screen displays the installed version, a link to help, and the login form when no account is paired.

The current screen is remembered between two openings of the panel (`sessionStorage` , key `inziderPanelState` , field `activeTab`). If you close the panel on the Templates view, it reopens on Templates.

30.4 — The title bar and its indicators

The title bar is more loaded than it appears. From left to right : the IziBuddy logo, the name of the active profile, a **status indicator** green or gray (auth OK / disconnected), a Claude usage indicator (consumption `Current session` / `All models` / `Opus only`) when you are on claude.ai, and finally the system buttons (collapse, close). The Claude usage indicator is retrieved by scraping `claude.ai/settings/usage` — see `src/extension/content/content-claude-usage.js` — and is only computed when you are actually on Claude. On the other platforms, that slot stays empty.

The connection status is refreshed periodically by light *polling* (a `GET /api/status` call to the configured server, see `tab-config-backend.js`). If the server becomes unreachable — network outage, restart — the indicator turns gray and a toast warns you ; but the panel remains usable as long as the templates are in local cache.

30.5 — Theme and position

The panel adapts to the host chatbot's appearance through the `theme.token_map` declared in each provider's YAML descriptor (`src/extension/config/ai_providers/*.yaml`). That map pairs the host platform's CSS custom properties with IziBuddy's own `--in-zinder-*` tokens; `ProfilesThemesUtils.applyTheme` compiles it into a `:root` override so the panel and the host page stay visually consistent. The panel's own light/dark rendering is driven by the `darkMode` setting (`dark` / `light` / `auto`, where `auto` follows `pre-fers-color-scheme`); you can also pick a profile theme from the Profiles screen (Chapter XXXIII).

The default position is the top right. You can drag the panel anywhere; its position is stored as a `top` offset plus a `rightOffset` from the window edge and restored on the next opening. The panel's height adjusts to the available resolution.

The panel's CSS is concentrated in `src/extension/floating.css` (~700 lines), loaded into the Shadow DOM by `content-panel-core.js`. Colors and tokens are expressed as CSS variables to allow overrides by custom themes.

30.6 — The footer

At the bottom of the panel, a small fixed zone houses contextual buttons: for example, on the Templates view, you find the quick buttons « Stash » (server-side templates tagged STASH) and « Recents » (the most recently injected templates). The number of buttons is adjustable from the configuration (`buttonCount`, `recentButtonCount`, `stashButtonCount` in `ConfigState`, see `src/extension/tabs/tab-config/tab-config-core.js`).

30.7 — Auto-collapse

To avoid interfering with your conversation, the panel can collapse on its own after an inactivity delay. This option (`autoCollapseEnabled`, 20 seconds by default) can be disabled. If you work in bursts

— you inject a prompt, the AI replies at length, you come back later — the auto-collapse saves you from having to collapse the panel by hand. If you work in quick back-and-forth (several templates in a row), disable it.

Chapter XXXI — Logging in from the extension

The extension has no use until it is tied to an IziBuddy dashboard : that’s where your templates, your collections, your groups live. Signing in is the first operation you do after installing the extension, and the last one you will come back to in case of an instance change. This chapter describes the initial sign-in, the *Remember me* option, and the moves to know when the server address changes.

31.1 — Initial sign-in

On install, the panel opens on the *About / Login* screen (Chapter XXX) and displays the login form. Two credentials are asked, exactly as on the web login page :

- **Username** : your username on the dashboard.
- **Password** : your dashboard password.

The backend the extension talks to is configured by **host** and **port** in the Configuration screen — `localhost` and `8765` by default (`window.inziderServerHost` / `window.inziderServerPort` , see `content-helpers.js`). The extension builds the base URL from those two values and appends `/api/...` on its own ; there is no free URL field and no token field.

When you validate, the extension calls `POST /api/auth/login` with a body `{ username, password, remember_me }` , retrieves a **JWT** and stores it in `chrome.storage.local` under the key `izibuddy_auth_token` (the username lands in `izibuddy_auth_user`). The title-bar status indicator turns green. You can go back to the Templates view : it is now usable.

The login code lives in `src/extension/tabs/tab-about-login.js` and `src/extension/tabs/tab-config/tab-config-core.js` (object `ConfigState`, which persists the token).

31.2 — A single JWT, one backend

There is no « personal token », no per-backend credential form, no multi-backend juggling : authentication is a **single JWT** obtained by username/password login against the one dashboard the extension is configured to reach. Everything the extension sends afterwards carries that JWT in the `Authorization: Bearer` header.

Profiles (Chapter XXXIII) are a **display** mechanism — theme, accent color, quick-buttons layout, local Stash — not a set of separate accounts or backends. Switching profiles does not re-authenticate against a different server.

31.3 — The *Remember me* option

The login form contains a **Remember me** checkbox (`remember_me`), passed to `POST /api/auth/login`. It asks the server for a longer-lived token so you are not signed out between browser restarts. In every case, the token is stored in `chrome.storage.local` — persistent browser storage keyed to your profile.

At every startup, the extension reads the stored token from `chrome.storage.local`. If no token is found, it reverts to disconnected mode and presents the login screen.

31.4 — Testing the connection

The Configuration screen checks reachability with a `GET /api/status` call to the configured host and port (see `tab-config-backend.js`). Two outcomes matter in practice :

- **Reachable** — the server answers `/api/status` ; the status indicator is green and the panel is usable.
- **Unreachable** — neither the host nor the port responds ; a toast reports that the server is not available on the configured port, and the indicator turns gray. Check network connectivity, the host/port values, and that the dashboard is running.

The status is also polled periodically in the background, so a server that goes down mid-session is reflected in the indicator without any manual action.

31.5 — Changing the server address

If your dashboard moves — a team migration, a switch between `localhost` and a staging server — update the **host** and **port** in the Configuration screen. The extension rebuilds its base URL from the new values and re-authenticates on the next login. There is no notion of several stored backends : one host/port at a time.

31.6 — What to do on a silent disconnect

The JWT expires — typically after a period set by the server. When expiration is detected (HTTP `401` response on an API call), the extension does **not** try to refresh a token silently. Instead it :

1. clears the stored auth (`clearAuth()` plus removal of `izibuddy_auth_token` and `izibuddy_auth_user` from `chrome.storage.local`);
2. dispatches an `izibuddy-auth-required` event ;
3. turns the status indicator gray and returns you to the login form.

The mechanism is implemented in the `TemplatesAPI` layer (`src/extension/tabs/tab-templates/tab-templates-api.js`, function `handle401`), which intercepts `401` responses and routes them back to sign-in. You log in again to obtain a fresh JWT.

31.7 — Security : what is stored, where

For the curious and for administrators who care about compliance, here is what lives in the browser after sign-in :

- In `chrome.storage.local` (persistent) : the JWT (`izibuddy_auth_token`), the username (`izibuddy_auth_user`), the active profile, and the cached templates. Chrome encrypts local storage at rest.
- In the panel's `Shadow DOM` : no secret, only the UI.

No data is sent to a third party, neither to Anthropic, nor to OpenAI, nor to Google. The extension speaks exclusively to the dashboard you configured.

Chapter XXXII — The Templates view

The Templates view is the operational heart of the extension. It's what turns the panel on your Claude or ChatGPT page into an input accelerator. You search for a prompt, you fill in its parameters, you send it into the chatbot's editor. This chapter details each component : the search field, the result dropdown, the Stash, the parameter dialog before injection, and the choice of a parameter set at the moment of sending.

32.1 — Overview

The Templates view is the panel's default screen. At the top, a **search field** with a searchable dropdown of results. Below, the **quick buttons** at the foot : Stash and Recents. There is no navigation banner on this screen — the search and the mini-config button sit directly in the content.

The list of templates your account can see is loaded once when the panel connects, then filtered **entirely on the client side**. There is no server-side pagination or infinite scroll : the searchable dropdown surfaces matches as you type.

The orchestration lives in `src/extension/tabs/tab-templates/tab-templates-core.js`, rendering in `tab-templates-ui.js`, the search dropdown in `tab-templates-search.js`, API calls in `tab-templates-api.js`.

32.2 — The search field

Type a few characters : the dropdown narrows live. The search targets the **name**, **description** and **tags** of the templates. A search for `cv` finds both *CV designer* and *curriculum vitae*. No regex, no boolean operators — the goal is speed, not expressive power.

The search works entirely on the client side : on connection, the extension loads the list of visible templates and filters it locally. Advantage : zero latency, and the last cached list is still searchable even when the connection to the dashboard is flaky. You navigate the dropdown with `↑` / `↓`, validate with `Enter`, and dismiss with `Esc`.

32.3 — The prompt entry

Each entry in the dropdown shows the prompt's name and, when space permits, its short description and tags. Selecting a prompt loads it : if it declares parameters, the parameter dialog opens (below) ; otherwise the prompt is ready to inject directly.

32.4 — The Stash : pinned templates

The **Stash** is a row of buttons at the foot of the panel (5 buttons by default, configurable from 2 to 8 in the Configuration screen via `buttonCount`). Clicking a button loads the corresponding prompt for one-click injection.

The Stash is not local browser storage : each Stash entry is a **real prompt on the server side**, tagged `STASH` (its primary tag) and identified by `stash_<profile>_<index>`. The Stash is loaded read-only by the extension (`filter(p => p.tags[0] === 'STASH')`), so it follows you across machines and is filtered out of your main view. See `src/extension/tabs/tab-templates/tab-templates-stash-core.js`.

The Stash is tied to your extension **profile** (see Chapter XXXIII) : the identifier prefix `stash_<profile>_<index>` keeps each profile's Stash separate.

32.5 — The parameter dialog

When a prompt declares POO²L parameters (Chapter XVII), an **input dialog** opens before injection. For each parameter, a field adapted to the declared type : `text` → text field, `textarea` → multi-line area, `number` → numeric field, `boolean` → checkbox, `select` → drop-down menu.

Mandatory parameters (`param`) are expected ; optional ones (`param?`) can stay empty — their default value is used. The values you enter are substituted by the same POO²L engine the dashboard uses (see below), which reports any problem at render time.

The dialog only shows up if the prompt actually declares parameters. A plain-text prompt goes straight to injection. Code : `src/extension/tabs/tab-templates/tab-templates-params.js` and the shared `src/extension/common/templates-params-dialog.js`.

32.6 — Choosing a parameter set

A complex prompt can declare **several predefined parameter sets** on the dashboard side (Chapter XVII) : for example, an e-mail-writing prompt may have a *Short formal reply* set, a *Long friendly reply* set, an *Informal draft* set. When such sets exist, the dialog offers a *Parameter set* selector at the top, which pre-fills the fields.

You can still modify the values after pre-fill — the set serves as a starter, not a straitjacket. The sets are loaded via `GET /api/templates/presets/{id}` at dialog opening (see `common/components/preset-manager.js`).

This mechanic is one of IziBuddy's most underestimated : it lets you turn a template-class into a *menu* of variants, without duplicating the content.

32.7 — The injection itself

Once parameters are filled, injection runs the pipeline :

1. the extension fetches the prompt's rendered source from the server via `GET /api/templates/render/{id}` ; the server resolves POO²L inheritance, expands `include` s and substitutes system variables ;
2. the extension performs **parameter substitution client-side** with the embedded POO²L engine (`TemplatesParams.substitute` , backed by the same `pool-wasm` engine as the backend) ;
3. it writes the resulting text into the chatbot's editor, dispatching the right DOM events (`input` , `change`) so that the chatbot's *Send* button activates ;
4. if attachments are present, they are handed to the platform-specific attachment router (see Chapter XXXIV).

At this stage, the panel collapses on its own (unless you disabled `autoCollapseEnabled`). All you have left to do is press *Send* in the chatbot — or edit the rendered prompt if you want to touch it up before sending.

32.8 — Offline mode

When the server is unreachable, the extension works from the **last cached** list of templates. Rendering does not degrade to a lesser engine : the extension carries the **same `pool-wasm` engine** as the dashboard (`src/extension/common/pool-engine-ext.js`), so parameter substitution — and the full POO²L grammar, including inheritance — is computed locally with the same result.

The one thing that needs the server is the set of **predefined parameter sets** (they live server-side) ; offline, the dialog falls back to free input. You recover them as soon as the dashboard becomes reachable again — no manual action needed.

Chapter XXXIII — The Profiles screen

An extension **profile** bundles a visual theme, an accent color, a quick-buttons layout and a local Stash. It is a **display** mechanism : a way to keep several presentation contexts on the same browser without reconfiguring the panel each time. This chapter describes the Profiles screen — selection, creation, visual customization — and presents the model for custom themes.

33.1 — What a profile is (and is not)

A profile is **not** a separate account, a separate backend, or a stored credential. Authentication is a single JWT against the one dashboard the extension is configured to reach (Chapter XXXI) ; switching profiles does not sign you into a different server. A profile only carries how the panel *looks* and which quick buttons it shows :

- a **name** ;
- a **theme** (a gray-shade base plus an accent color) ;
- a **quick-buttons layout** (`buttonCount` , and how the slots split between Stash and Recents) ;
- a **Stash**, addressed by the profile-prefixed identifiers `stash_<profile>_<index>` (Chapter XXXII).

Profiles are stored in `chrome.storage.local` (`templateProfiles`), with the active profile in `currentProfile` . See `src/extension/tabs/tab-config/tab-config-profiles.js` .

33.2 — Why profiles

A browser extension lives in a single browser, but a working day strings together several hats. In the morning you draft a client memo and want a sober dark theme and your review macros in the

Stash ; in the afternoon you switch to personal projects and prefer a light theme and a different Stash. A profile bundles those presentation choices so a single switch swaps them all.

33.3 — Selecting and managing a profile

The Profiles screen presents a **dropdown** (`profile-select`) that lists existing profiles ; picking one activates it. Next to the dropdown, three buttons act on the current profile : **edit** (✏️), **duplicate** (📋) and **delete** (🗑️). There is no card grid and no separate « New profile » wizard — duplication is the usual way to spin up a variant of an existing profile.

Switching a profile applies its theme, accent color, buttons layout and Stash. The panel updates in place ; the title bar shows the active profile's name. The switch is a presentation change, not a re-authentication.

33.4 — The per-profile Stash

As mentioned in Chapter XXXII, each profile has its own Stash. Concretely, the Stash slots are server-side templates tagged `STASH` whose identifier carries the profile prefix (`stash_<profile>_<index>`). Switching from one profile to another simply loads a different row of identifiers.

This has a practical consequence : to clean a profile's Stash, delete the corresponding templates from the dashboard — they are ordinary templates on the server.

33.5 — Profile themes

Each profile carries its own theme. The theme is built from a **gray-shade base** (the panel's background family) and an **accent color**, chosen from a color grid in the theme editor. The extension does not offer a fixed Auto/Light/Dark trio : `_deriveBaseTheme` classifies the chosen gray into a base (`verylight` , `light` , `medium` , `dark`)

and the accent hue is applied on top. The panel's own light/dark rendering is governed separately by the global `darkMode` setting (`dark` / `light` / `auto`, `auto` following `prefers-color-scheme`).

33.6 — Custom themes

The theme editor is a **color grid**: the first row picks the gray shade (the background base), the remaining rows pick the accent color. A preview pastille combines the accent's hue and saturation with the gray's luminance so you see the resulting tone before applying it (`tab-profiles-themes.js`, `_computePreview`).

The result is applied by injecting a style block identified by the class `inzinder-custom-theme` into the panel's Shadow DOM, which sets the internal CSS variables (`--inzinder-bg`, `--inzinder-text`, `--inzinder-accent`, …). The mapping between those internal tokens and each host platform's tokens comes from the provider's `theme.token_map` (Chapter XXX) and is compiled by `src/extension/tabs/tab-profiles/tab-profiles-themes-utils.js` (object `ProfilesThemesUtils`). There is no free border-radius slider and no compact/normal/airy density control.

33.7 — Quick-button layout

The profile also remembers the **quick buttons** at the foot of the panel:

- `buttonCount` — total number of slots (2 to 8, default 5),
- `recentButtonCount` — how many are reserved for *Recents*,
- `stashButtonCount` — how many are reserved for the Stash.

A profile geared toward heavy reuse raises `stashButtonCount`; a lighter profile keeps a couple of Stash slots and a couple of Recents. These keys live in `tab-config-options.js`.

33.8 — A display layer, not a security boundary

Because a profile is purely presentation, there is nothing secret to export or protect in it : no URL, no username, no JWT, no password. Cleaning up or duplicating a profile only rearranges how the panel looks and which Stash it points at — your credentials and your templates live on the dashboard, unaffected by the profile you happen to have selected.

Chapter XXXIV — Per-platform behavior

The IziBuddy extension serves the same gesture — *find a prompt and inject it* — across several different platforms. Each has its own message editor, its own DOM structure, its own quirks of file attachment. This chapter describes, platform by platform, the adaptations the extension applies under the hood. You never have to trigger them manually, but knowing them helps understand the rare cases where a platform evolves faster than the connector.

34.1 — Platform detection

On panel opening, the extension determines which platform it is on via `window.detectAIPlatform()`, which delegates to `AIProviders.detect(hostname, pathname)`. The provider descriptors live in `src/extension/config/ai_providers/*.yaml`; each declares the hostnames it matches. The recognized identifiers are `claude`, `claude-code`, `chatgpt`, `grok`, `gemini` and `deepseek`.

Text injection is **generic**: there are no per-platform `injectIntoClaude` / `injectIntoChatGPT` functions. The extension writes the rendered prompt into whatever composer the page exposes (`contenteditable` or `textarea`) and dispatches the DOM events that make the chatbot's *Send* button activate. What *is* platform- specific is **file attachment**: the central router `content-platform-router.js` dispatches to a per-platform attachment function (`attachFileToClaude`, `attachFileToChatGPT`, `attachFileToGrok`, `attachFileToGemini`, `attachFileToDeepSeek`). This split stays deliberately flat — debugging an attachment case stays local to a single file.

34.2 — Claude (claude.ai)

Claude uses a rich ProseMirror editor, with a `<div contenteditable="true">` as the input zone. The extension places the rendered text in the composer and dispatches a synthetic `InputEvent` so that ProseMirror registers the change and activates the *Send* button.

File attachment is handled by `attachFileToClaude` (`src/extension/content/content-platform-claude.js`) with three methods in cascade : direct `input[type="file"]`, aria-labelled *attach* button, then simulated drag-and-drop over the `contenteditable` area. The cascade allows survival of Claude UI revamps that change one method but leave at least one functional.

The Claude usage indicator (Chapter XXX) reads `claude.ai/settings/usage` via `fetch` with `credentials: 'include'`, extracts the three percentages (*Current session*, *All models*, *Opus only*) and displays them in the title bar. Code : `src/extension/content/content-claude-usage.js`.

34.3 — ChatGPT (chatgpt.com, chat.openai.com)

ChatGPT, in its latest versions, abandoned the `<textarea>` in favor of a `<div contenteditable="true">` flanked by a legacy `#template-textarea`. The *Others* connector (`content-platform-others.js`) tests both when attaching a file :

1. first looks for `textarea[data-id]` ;
2. falls back to `#template-textarea` ;
3. as a last resort, targets the first `div[contenteditable="true"]` under the `<form>` element.

File attachment (`attachFileToChatGPT`) starts by looking for a direct `input[type="file"]`, then falls back to an aria-label *Attach* button that needs to be clicked to reveal the input, then as a last re-

sort to drag-drop. ChatGPT's attachment button migrates regularly between `[data-testid*="attach"]` and a SVG without a stable attribute — the cascade survives most revamps.

The panel's appearance on ChatGPT is aligned through the provider's `theme.token_map` (Chapter XXX), not by observing a `data-chat-theme` attribute.

34.4 — Grok (x.com/i/grok)

Grok inherits the Twitter/X architecture : the input zone is a `[data-testid*="tweetTextarea"]` or a contenteditable `[role="textbox"]`. File attachment goes through aria-label *Media* or *Add* buttons, similar to those of the Twitter UI.

Grok is the platform most exposed to internal UI changes — the X interface evolves very fast. The connector therefore keeps extra robustness : if none of the attachment heuristics work, an error toast invites the user to report the case.

34.5 — Gemini (Google AI Studio)

For Gemini, the extension targets **Google AI Studio** (`aistudio.google.com`) — that is the only Gemini host it matches (`config/ai_providers/gemini.yaml`); the consumer `gemini.google.com` surface is not injected.

`attachFileToGemini` looks for an `input[type="file"]`, and where the attach control is a Material `mat-icon` reading `attach_file`, it clicks it to reveal the input. The panel's appearance is aligned through Gemini's `theme.token_map`, which maps the Material Design `--mdc-theme-*` variables onto IziBuddy's `--inzinder-*` tokens.

34.6 — DeepSeek (chat.deepseek.com)

Deep-
Seek is the simplest of the platforms : a classic composer and a *Send* button. `attachFileToDeepSeek` looks for the file input, falling back to an aria-labelled *attach* button. Text injection follows the same generic path as the others.

34.7 — The Server-Sent Events channel

There is one SSE channel in the extension, and it points the other way from what one might assume : the extension is an **SSE client to the IziBuddy backend**, not a reader of the chatbot's streaming. The code lives in `src/extension/content/content-sse.js` ; it opens an `EventSource` on `GET /api/events/stream` (with the connected user as a query parameter) to receive server-side notifications — for instance a shutdown warning from the dashboard. It does **not** listen to Claude's, ChatGPT's or Gemini's response stream.

The client only connects when the server is known to be reachable (`serverAvailable`), throttles reconnection attempts, and falls back to `anonymous` (broadcast-only) when no username is resolved.

34.8 — When a platform evolves

A platform changes its DOM, injection or attachment stops working. Diagnostic : open the developer console, consult `src/extension/content/content-platform-<platform>.js` or `content-platform-others.js` to identify the CSS selectors used, check in the inspector whether they still exist, then add an entry to the cascade or open an *issue* on the repository. Adding a brand-new platform (a new provider YAML) is covered in Chapter LVII.

Chapter XXXV — Stash, substitution, parameter sets at injection time

Chapter XXXII introduced the visible parts of the Templates view ; this one opens the hood. Three mechanisms work when you inject a prompt : the **Stash** is a set of server-side templates, the **substitution** turns the POO²L source into rendered text through the POO²L engine, and the **parameter sets** pre-load the input dialog. Understanding these three gears helps diagnose the rare cases where the injected result does not match what you expected.

35.1 — Where the Stash lives

The Stash is a row of slots pinned at the foot of the floating panel. Contrary to what its name suggests, this is not local browser storage : each Stash slot is a **real prompt on the server side**, tagged `STASH` (its primary tag, so it is filtered out of the main view) and identified as `stash_<profile>_<index>`.

The choice to store on the server has two motivations. On the one hand, your Stash follows you when you sign the extension in on a new machine — no need to rebuild everything. On the other hand, Stashes remain versioned just like your other templates : if you overwrite a temporary Stash during an intense work session, you can recover it from the version history (Chapter XXVIII).

The flip side : writing a Stash needs the server. In offline mode, the Stash remains viewable from the cache but cannot be modified. This is rarely a problem in practice — you don’t edit a Stash when you have no network, you use one.

35.2 — The per-profile index

The identifier prefix `stash_<profile>_<index>` ensures isolation between profiles. The *Work* profile and the *Personal* profile can both have a Stash at slot 3 without overwriting each other : they are `stash_work_3` and `stash_personal_3`, two distinct templates server-side.

The index runs up to `buttonCount - 1`. When you increase `buttonCount` in the configuration, the new slots are initially empty — the extension does not create blank `stash_<profile>_<n>` entries, it does so only on the first content save.

35.3 — The substitution engine

Rendering a prompt is delegated to the **POO²L engine** — the same `pool-core` engine (via its `pool-wasm` build) used by the dashboard and the backend. There is no bespoke extension pipeline. The engine performs, in a single render pass :

1. **Inheritance resolution** — if the prompt declares a parent, the parent and child bodies are combined according to the grammar's override rules.
2. **`include` expansion** — each `@@@{include X}` is resolved recursively ; cycles are detected and reported as a blocking diagnostic.
3. **Parameter substitution** — each declared `@@@param` is replaced by the value from the dialog (or its default when optional and left empty).
4. **System-variable substitution** — `@@@=DATE`, `@@@=USER` and the other reserved variables are substituted from the host-provided context (see Chapter XLI for the exhaustive list).

Concretely, at injection the extension fetches the server-rendered source via `GET /api/templates/render/{id}` (which resolves inheritance, includes and system variables), and applies the dialog's parameter values locally through the embedded engine

(`TemplatesParams.substitute`). System variables such as `@@@=DATE` therefore reflect the moment of injection, and stay in ISO form.

35.4 — Constants do not appear in the dialog

A useful particularity to know : POO²L **constants** (`@@@const X : text = ...`, Chapter XVIII) are substituted at render time **without appearing in the input dialog**. This is deliberate : a constant is a value the prompt's author set once and for all, not a variable the user has to re-fill on each injection.

Typical example : a code-review prompt may contain `@@@const STYLE_GUIDE : text = "PEP 8 strict"`. On every injection, the rendered text will mention *PEP 8 strict* without asking the user for confirmation. If the organization decides to switch to another style guide, the prompt's author edits the constant, and all users see the update at their next injection.

35.5 — Predefined parameter sets

A prompt can declare several **parameter sets** on the dashboard side (Chapter XVII). A set is a name + a list of *parameter → value* pairs. When such sets exist, the user picks one from the selector at the top of the input dialog ; the fields pre-fill.

Classic use case : an e-mail-writing prompt can offer the sets *Short formal*, *Long friendly*, *Informal draft*. Rather than having three almost-identical templates, the author keeps a single one with three sets.

Parameter sets are **stored server-side, per user**, not on the extension. They are read via `GET /api/templates/presets/{id}` when the dialog opens. A set only pre-fills values ; it does not change the prompt's parameter declarations or constraints.

You can of course modify the values after pre-fill. The set serves as a starter, not a straitjacket. If you systematically modify the same parameters of a given set, perhaps you deserve a new set.

35.6 — Saving a new set from the extension

When you fill the dialog with particularly useful values, you can save them as a named set. It is created server-side via `POST /api/templates/presets/{id}` and becomes available for your future injections of that prompt. Parameter sets are per-user : your sets are yours, they are not silently pushed onto other users.

This is one of the most efficient capitalization paths : you discover through usage a repetitive case, and you crystallize it into a named set without having to write a new prompt.

35.7 — The offline behavior

When the server is unreachable (Chapter XXXII), rendering does **not** fall back to a lesser engine. The extension carries the **same `pool-wasm` engine** as the dashboard (`src/extension/common/pool-engine-ext.js`), so the full POO²L grammar — parameter substitution, system variables, inheritance and `include`s — is computed locally with the same result.

What genuinely needs the server is the set of **predefined parameter sets** (they live server-side, per user) ; offline, the dialog falls back to free input. Everything else renders identically.

35.8 — Diagnosing an unexpected render

If the injected text does not match what you expected, the reflex is to open the prompt in the dashboard and use **Live preview** (Chapter XXII) with the same parameters : the preview uses exactly the same POO²L engine and produces the same result, which confirms the problem comes from the source content and not from a stale cache. Three frequent sources of surprise : a parent that changed without you knowing ; an `include` toward a prompt recently renamed ; a system variable in an unexpected form (`@@@=DATE` is always `YYYY-MM-DD`, see Chapter XLI).

Partie 6

Customization

Chapter XXXVI — Themes

IziBuddy operates in two visual contexts — the dashboard (a full web application) and the browser extension (a floating panel injected into a chatbot page) — and both support light, dark and custom themes. This chapter collects the transversal information on appearance ; extension theme details are covered in Chapter XXXIII, fine customization in Chapter XXXVIII.

36.1 — Light, dark, custom

The dashboard offers a **theme toggle** in the top right of the header. It does not read the system preference ; it **cycles three positions** :

- **Light** (☀️) — light backgrounds, dark text.
- **Dark** (🌙) — dark backgrounds, light text.
- **Options theme** (🎨) — the custom theme currently selected on the *Options* page (see below).

The `base` position (light or dark) drives the `data-theme` attribute on `<html>`, which triggers the component dark-mode CSS rules. The toggle's position is cached in `localStorage.izibuddy_theme_toggle` to avoid the *flash* at page load (the attribute is set at boot, before the server responds). The switching code lives in `src/static/index-themes.js`.

36.2 — The token system

All of IziBuddy's visual work goes through **CSS variables** — the *design tokens* — prefixed `--izi-`. A theme change does not swap an entire stylesheet, it changes the value of the tokens. This makes theme contributions simple : to create a new theme, you redefine a few dozen tokens, not hundreds of rules.

The main tokens :

- `--izi-bg` , `--izi-bg-secondary` , `--izi-bg-tertiary` — backgrounds.
- `--izi-text` , `--izi-text-secondary` — text.
- `--izi-primary` , `--izi-primary-dark` , `--izi-primary-light` — the accent color (base, hover/pressed, light fill).
- `--izi-success` , `--izi-error` , `--izi-warning` , `--izi-info` — semantic colors.
- `--izi-diff-added` , `--izi-diff-removed` , `--izi-diff-modified` , `--izi-diff-restored` — the comparator colors.

The default values are defined in `src/static/styles/base.css` — the single source of truth for the tokens (there is no separate `tokens.css`). The same file also defines the PPI scaling factor (see *Typeface and sizes* below).

36.3 — Per-platform appearance in the extension

Where the dashboard changes its own tokens, the extension has extra work : it must sit visually inside the host platform. It does not detect the host theme by reading a `dark` class or a `data-chat-theme` attribute, and it runs no luminance heuristic or `MutationObserver` on the host DOM. Instead, each provider descriptor declares a **`theme.token_map`** (`src/extension/config/ai_providers/*.yaml`) that pairs the host platform's CSS custom properties with IziBuddy's internal `--inzinder-*` tokens. `ProfilesThemesUtils.applyTheme` compiles that map into a `:root` override so the panel and the host page stay consistent. The panel's own light/dark rendering is a separate `darkMode` setting (`dark` / `light` / `auto` , where `auto` follows `prefers-color-scheme`).

36.4 — Custom dashboard themes

On the dashboard side, the *Options* page lets you pick from a set of built-in themes (Solarized, Nord, Dracula, Sépia, Océan, high-contrast variants…) and to define your own by editing the token va-

lues through a color editor with a real-time preview. There is **no free-form custom-CSS field** : customization goes through the token values, not arbitrary CSS.

Themes are stored **server-side** in your web preferences under the keys `themes` (your named token sets) and `active_theme` (the selected one), persisted through `PUT /api/users/me/web` .

36.5 — Custom extension themes

The extension has an analogous mechanism, described in detail in Chapter XXXIII : an `inzinder-custom-theme` style block is injected into the panel's Shadow DOM, setting the internal tokens (`--inzinder-bg` , `--inzinder-text` , `--inzinder-accent` , …). The correspondence between those internal tokens and the host platform's tokens comes from each provider's `theme.token_map` , handled by `src/extension/tabs/tab-profiles/tab-profiles-themes-utils.js` .

The practical idea : dashboard and extension can share a visual identity. If you settle on an accent hue in the dashboard, you can apply a matching accent in the extension via the corresponding profile.

36.6 — Contrast

Several of the built-in themes are **high-contrast** variants (for instance the *high contrast — dark* and *neutral gray* presets in `index-themes.js`), which raise the separation between text and background for readability. These are ordinary selectable themes ; IziBuddy does not auto-detect `prefers-contrast` and does not reject a theme whose token values you have made pale — you remain in control of the values you pick.

36.7 — Typeface and sizes

The default typographic family is the system stack (`system-ui, -apple-system, ...`): no web fonts to load, no FOUT. Sizing is not a three-step Compact/Normal/Airy control but a continuous **PPI scaling factor**: `--izi-scale-factor` (default `1`) is adjusted dynamically by JavaScript from the device's `devicePixelRatio`, and the derived tokens (`--izi-font-size-base`, `--izi-spacing-base`, tile and header sizes) are all computed as `calc(... * var(--izi-scale-factor))` in `src/static/styles/base.css`. High-density screens therefore get a proportionally scaled UI without a manual density switch.

36.8 — Compatibility with script blockers

A few users of aggressive blocking extensions (NoScript, uBlock in strict mode) can encounter broken rendering when stylesheets load out of order — the symptom is a flat, token-less background. The remedy: add the dashboard domain to the blocker's allowlist. IziBuddy loads no third-party script and sends no data to an external domain — you can allowlist it without risk.

Chapter XXXVII — UI presets

The dashboard interface unfolds or folds back according to your usage profile. A beginner doesn't need to see the *Hidden* filter nor the *Promote to collection* button ; an expert can't stand having them hidden. To avoid drowning newcomers while freeing veterans, IziBuddy offers four **UI presets** : Simple, Standard, Advanced, Expert. This chapter explains what each preset enables, how to switch from one to another, and how the legacy UI LEVEL (`ui_level`) setting is migrated silently.

37.1 — The preset logic

Under the hood, each preset is simply **a list of enabled features**. There is no hidden hierarchy, no different default behavior ; a preset = a *Set*. The reference code reads clearly in `src/static/index-features-registry.js` , constant `PRESETS` .

The practical consequence is important : moving from *Simple* to *Standard* is strictly equivalent to manually enabling the few features *Standard* adds. From *Standard* to *Advanced* likewise. You can start in *Simple*, step up to *Standard* after a week, step back down to *Simple* the day you work in a client presentation, without losing anything — your templates do not depend on the preset, only their visibility in the UI.

37.2 — Simple

The **Simple** preset enables six features, and only six :

- `filterClearAll` — a *Clear all* button to wipe current filters.
- `containerNewCollection` — create a new collection.
- `actionDragDrop` — move a prompt by drag-and-drop.
- `actionUndoRedo` — undo/redo an action.
- `chromeExtensionLink` — a link to the extension's installation.

- `tipsEnabled` — the contextual tips (Chapter XXXIX) remain active ; they are what will suggest stepping up the preset at the right moment.

Everything else is hidden. No marker filters, no tags, no versioning panel, no explicit POO²L parameters. The user sees their templates, edits them as plain text, injects them. That's all. And it's exactly the right starting point for someone who just wants to move past copy-paste.

37.3 — Standard

The **Standard** preset enables twenty-five features. It adds, over *Simple* :

- the main marker filters : `filterImportant` , `filterLocked` , `filterWip` , `filterNoScoring` ;
- the container-type filters : `filterFolder` , `filterCollection` , `filterTemplate` ;
- the display options useful day-to-day : `viewDetails` , `viewTileSizes` , `viewScoring` ;
- folder creation : `containerNewFolder` ;
- the L5 option on collections : `collectionOO2` , and the container color stripe : `containerColorStripe` ;
- notes, attachments and glyph replacement in the editor : `editorNotes` , `editorAttachments` , `editorGlyphReplace` ;
- bulk actions : `actionCutPaste` , `actionRubberBand` (on top of `actionDragDrop` and `actionUndoRedo` already active) ;
- tag hierarchy : `tagsHierarchy` .

It's the default preset for a user who has built a library of a few dozen templates and wants to start organizing them. Version management stays hidden — it arrives in *Advanced*.

37.4 — Advanced

The **Advanced** preset enables forty-two features. It adds, over *Standard* :

- the rare filters : `filterHidden` , `filterDisabled` , `filterExample` ;
- filter inversion : `filtersExclude` ;
- the complexity-level filter : `filterLevel` ;
- table density : `viewTableDensity` ;
- the full container UI : `containerKindUI` , `containerPromoteDemote` , `containerPathDrop` ;
- cross-pane copy-paste : `actionCopyPaste` ;
- the admin operations on tags : `tagsAdmin` ;
- access to groups : `sourcesGroups` ;
- click-to-filter on tag : `clickTagToFilter` ;
- the **versioning and diff** panel : `versioningDiff` .

It’s the target preset for power users : they need everything that touches organization and versioning, but not yet the very technical features (Git, lab tests).

37.5 — Expert

The **Expert** preset enables **all** forty-four features — it is declared as the whole registry, not as a list, so it can never fall behind it. Beyond the *Advanced* set, it adds :

- `viewSplitPane` — the side-by-side two-pane view ;
- `editorSmartQuotes` — smart-quote substitution in the editor ;
- `actionCrossPaneF5F6` — the cross-pane copy/move keyboard shortcuts ;
- `gitSettings` — Git integration for prompt synchronization ;
- `labTests` — the template-tests panel.

The editor's POO²L surfaces are not among them, in any preset : they follow the **editing-level ceiling** (`pool_max_level` , Options → Editor → Input), which is a separate question with a separate control.

It's the mode of advanced contributors and integrators. If you write your own scripts against the IziBuddy API, you will most likely be in Expert.

37.6 — Switching from one preset to another

The preset selector — a dropdown — lives at the top of the **features panel** (Chapter XXXVIII), in the *Options* page. Choosing *Simple*, *Standard*, *Advanced* or *Expert* immediately applies the preset : the UI reconfigures in place, without reload. Hidden elements disappear, new ones appear.

The switch is persisted server-side via `PUT /api/users/me/web` with the field `active_preset` . On the next dashboard load, you find your preset.

An important note : the switch **overwrites** your manual customizations. If you were in *Standard* with a few *Advanced* features enabled by hand, and you go back to *Standard*, those customizations are lost. The features panel (Chapter XXXVIII) lets you re-adjust finely after the switch.

37.7 — Auto-migration from `ui_level`

Before the redesign that introduced presets, IziBuddy used a three-level `ui_level` setting : *simple*, *standard*, *advanced*. Existing users have a `ui_level` recorded in their `web.json` .

On the first load after update, the application performs a **silent auto-migration** : the table `UI_LEVEL_TO_PRESET` of the registry (`simple` → `simple` , `standard` → `standard` , `advanced` →

`advanced`) is used to pick the equivalent preset, and that preset is persisted server-side. `ui_level` is still read but no longer drives anything.

Consequence : no user loses their preferences at the update. Former `advanced` users end up in *Advanced*, can step up to *Expert* if they want everything, and most don't even notice the migration.

37.8 — When to recommend a preset change

Three signals invite a preset step-up : more than 50 templates and *Simple*'s basic filtering becomes painful (move to *Standard*) ; repeated duplications to produce variants (move to *Advanced* for explicit POO²L parameters) ; more than 200 templates or application-level integration (*Expert*). The contextual tips system (Chapter XXXIX) detects these thresholds automatically and suggests the switch at the right moment ; but nothing prevents anticipating, the switch is free and reversible.

A frequent question : *does my dashboard preset reflect in the extension ?* No. Presets drive the dashboard UI ; the extension has its own configuration (Chapter XXXI). You can be *Expert* on the dashboard side for authoring, and use a deliberately minimal extension to inject.

Chapter XXXVIII — The features panel

The previous chapter introduced the four presets — Simple, Standard, Advanced, Expert. This chapter goes one notch deeper : it describes the **forty-nine individual features** that IziBuddy exposes, the UI that lets you toggle them one by one, and the persistence mechanism that keeps your configuration in sync across your sessions.

38.1 — Why granular features

Presets cover 80% of cases, but not 100%. A user on *Standard* may want to occasionally enable `viewSplitPane` because they are editing several templates in parallel, or hide `actionDragDrop` because they keep dropping templates by mistake on a sensitive trackpad. The features panel lets them make these adjustments without being forced to move up to a full preset.

Each feature has a stable **identifier** (for instance `viewSplitPane`), a **category** (among eight), an i18n **label**, and an i18n **description**. The code lives in `src/static/index-features-registry.js` ; the UI rendering in `src/static/index-features-ui.js` .

38.2 — The nine categories

The forty-four features are organised into nine categories, grouped into four areas (profile, main list, editor, advanced). The UI presents each category as a **tab** ; selecting a tab shows that category’s features.

Two of the nine — **Language** and **Credentials** — carry no feature checkbox of their own : they are fed entirely by the sections relocated into them, so a category tab can be useful while its feature count is zero.

Filters (13)

`filterImportant`, `filterLocked`, `filterWip` — marker filters (star, padlock, WIP). `filterNoScoring` — the **Note** flag, not templates that have never been scored. `filterHidden`, `filterDisabled`, `filterExample` — advanced markers. `filterFolder`, `filterCollection`, `filterTemplate` — by element type. `filterClearAll` — *Clear all* button. `filtersExclude` — enables exclusion for ordinary attribute filters. Hidden instead has normal (non-hidden only), green (hidden only), grey (all) states. `filterLevel` — filter by POO²L complexity level.

Display (5)

`viewDetails` — *Details* column. `viewTileSizes` — tile size selector (S, M, L). `viewTableDensity` — table density. `viewSplitPane` — side-by-side two-pane view. `viewScoring` — score display.

There is no toggle for the date format. Which of the two forms shows is decided by the TILE SIZE — large tiles have room for *created → modified*, medium and small for the modification alone — because it is a question of room, not of preference. The renderer emits both forms and the stylesheet shows one.

Containers (7)

`containerKindUI` — folder ↔ collection switch at creation time. `containerPromoteDemote` — *Promote* and *Demote* buttons. `containerNewFolder`, `containerNewCollection` — type-specific creation. `containerPathDrop` — path entry for hierarchical creation. `collectionOO2` — L5 semantics (collection as a POO²L class). `containerColorStripe` — the colored stripe on containers.

Organisation (4)

`tagsHierarchy` — hierarchical tag display. `tagsAdmin` — admin operations (merge, bulk rename). `sourcesGroups` — group selector in the filter bar. `clickTagToFilter` — clicking on a tag drops it as a filter.

Actions (6)

`actionDragDrop` — drag-and-drop. `actionCutPaste`, `actionCopyPaste` — cut/copy-paste in the list. `actionUndoRedo` — `Ctrl+Z` / `Ctrl+Y`. `actionCrossPaneF5F6` — F5/F6 copy/move between the two panes in *split pane*. `actionRubberBand` — lasso selection.

The editor's POO²L surfaces are NOT features

There is no POO²L category, and there are no `pool2l*` toggles. The editor's POO²L surfaces — the parameters and constants dialogs, the parent selector and chain viewer, section management — follow the **editing-level ceiling** (`pool_max_level`, Options → Editor → Input), which is the only axis that governs them: one question, one control.

Four such toggles did exist. Two drove nothing at all from the day they were added; the other two CONTRADICTED the ceiling — the Standard preset hid the Structure tab while the ceiling at L5 allowed `@@@section`. Two answers to one question, disagreeing. They were removed rather than reconciled.

Input (4)

`editorNotes` — private *Notes* pane attached to the editor. `editorAttachments` — attachments manager. `editorGlyphReplace` — glyph replacement in the editor. `editorSmartQuotes` — smart-quote substitution.

Advanced (5)

`versioningDiff` — versioning panel with diff. `gitSettings` — Git configuration. `labTests` — prompt test panel. `chromeExtensionLink` — extension install link. `tipsEnabled` — global tips switch (Chapter XXXIX).

38.3 — The category tabs and the tristate indicator

The categories are presented as **tabs**, one visible at a time — not a clickable accordion. Each tab carries a **tristate indicator** (`data-tristate`) that summarises the category at a glance :

- **on** — every feature in the category is active ;
- **partial** — some are active, others are not ;
- **off** — none are active.

The indicator is a **visual signal** ; clicking a tab switches the active category to reveal its features (it does not cycle a state). You toggle individual features from their own checkboxes inside the tab.

38.4 — Feature search

Above the tabs, a search field filters features by **label** (in the current language). Type `split` to surface `viewSplitPane`. When you search, the panel shows a flat cross-category view of the matches ; picking a tab afterwards clears the search and returns to that tab’s content.

It is the fastest entry point when you are reading this manual and want to enable the feature that was just mentioned.

38.5 — Named configurations

Instead of a single *Reset*, the panel offers **named configurations** : a selector to load a saved configuration, plus buttons to **save** the current feature set under a name (💾), **rename** it (✏️) and **delete** it (🗑️) — template-driven, like the custom themes. This lets you keep several personal setups (say, a lean « presentation » set and a full « authoring » set) and switch between them, beyond the four built-in presets.

38.6 — The extras relocator

A few features bring along UI components that do not simply appear/disappear : they migrate from one place to another (an option that lives in the main bar when alone, but moves into a sub-menu when several options of the same kind coexist). The **relocator** orchestrates these moves, invoked after every preset application or feature toggle.

38.7 — Persistence

Your preferences live in two places :

- **Server side** : in `~/.izibuddy/users/{username}/settings/web.json`, under the keys `active_preset` (string) and `features` (Dict[str, bool]). It is the source of truth, read on every page load.
- **Browser side** : in `localStorage.izibuddy_features`. It is a cache that lets your features apply at boot, before the server responds — eliminating the *flash* of a UI in *Standard* mode that suddenly reconfigures itself into *Expert*.

The synchroniser lives in `src/static/index-features.js`. When you toggle a feature, the API `PUT /api/users/me/web` is called with the new state. When the server responds, the local cache is updated. On server failure (network cut), your change stays persisted locally and is resynchronised on the next connection.

Chapter XXXIX — Contextual tips

IziBuddy has a wide functional surface — 44 features, 4 presets, dozens of configurable behaviors. Reading the manual cover to cover is unrealistic : most users never discover half of what would be useful to them. To bridge this gap between *what exists* and *what you actually use*, IziBuddy ships a system of **contextual tips** — discreet *toasts* that pop up at the right moment and offer to enable the feature that would make the task you are doing right now more efficient.

39.1 — The principle

A tip is not a system notification, not a blocking banner, not a scripted onboarding. It is a non-blocking toast in the bottom-right of the screen, showing a title, a short explanation, and one or more **action buttons** that apply the suggestion in a single click — typically via `IziFeatures.applyPreset` or `IziFeatures.setFeature`.

You can ignore it, read it and dismiss it, or click the proposed button. In every case, the tip disappears after a few seconds. It will not show again in the same session, and the quota mechanism guarantees that it cannot pester you if you are not interested.

The engine lives in `src/static/index-tips.js` ; the tips themselves are declared in JSON in `src/static/data/tips.json`. The toast style sits in `src/static/styles/tips-toast.css`.

39.2 — The eleven built-in tips

At the time of writing, IziBuddy ships **eleven tips**. Each one is attached to a *trigger* (an application event), to *conditions* (a feature state, a preset) and to zero or more *actions* (offered buttons).

`poolevel-max-reminder` — on **application boot** (trigger `app-Boot`); a reminder about the maximum editable POO²L complexity level. Button : *Open settings* (`openSettings`).

`first-deletion-undo` — on your **very first deletion** of a prompt; explains that deletion is *soft* (trash) and that you can restore or press `Ctrl+Z`. No action button.

`promote-folders-50` — at **50 templates** without `filterFolder`. Suggests moving up to *Standard* to discover folder-based organization. Button : *Apply Standard*.

`simple-preset-many-templates` — at **80 templates** in *Simple*. Polite insistence : your library exceeds what *Simple* handles comfortably. Button : *Apply Standard*.

`duplicate-burst-pool2l` — **3 duplications in 5 minutes** on the *Simple* or *Standard* preset. Implicit hypothesis : you are producing variants by duplication, which POO²L parameters avoid. Button : *Apply Advanced*.

`promote-collections-3-folders` — at least **3 folders and no collection**. Suggests enabling `containerPromoteDemote` to promote a folder into a collection. Button : *Enable Promote/Demote*.

`cut-paste-burst-split` — **5 cut-pastes in 10 minutes** without `viewSplitPane`. Hypothesis : you are editing several templates in parallel. Button : *Enable split pane*.

`first-version-saved` — at your **second version save**, if `versioningDiff` is inactive. Button : *Enable version diff*.

`no-tags-30-templates` — **30 templates with no tag at all**. Suggests enabling the tag hierarchy. Button : *Enable tag hierarchy*.

`standard-using-collections` — in *Standard*, at your second collection. A good moment for inheritance, POO²L parameters and *split pane*. Buttons : *Apply Advanced* + *Enable split pane*.

`expert-discovery` — at **200 templates** in *Standard* or *Advanced*. You are a power user, *Expert* unlocks everything. Button : *Apply Expert*.

39.3 — Triggers and conditions

Each tip declares a **trigger** (an event name plus thresholds) and **conditions** (the application state required to display the tip). The engine listens to events emitted by `IziTips.notify(eventName, payload)` throughout the application.

Available triggers are :

- `appBoot` — fired once when the application boots (no threshold).
- `templateCount` — total of visible templates, with a `min` threshold.
- `containerCount` — number of folders and collections, with `minFolders`, `maxCollections`, `minCollections`.
- `duplicateBurst` — N duplications inside a sliding window, with `count` and `windowMs`.
- `cutPasteBurst` — N cut-paste operations inside a window.
- `templateDeleted`, `templateVersionSaved` — occurrences with `occurrence: N` (Nth time the event happens).

Available conditions are :

- `anyFeatureOff: [...]` — at least one of the listed features is inactive.
- `preset: "simple"` — exact preset.
- `presetIn: [...]` — preset from a list.
- `tagsUsedAtMost: N` — at most N distinct tags in use.

Evaluation is immediate at notification time : if the trigger fires and all conditions are met, the tip is queued. If it has not been shown recently (see quotas below), it is displayed.

39.4 — Anti-spam quotas

Three hard-coded rules in the engine ensure that no tip invades your field of view :

1. **Five minutes minimum between two tips.** Even if ten triggers fire in cascade, you will see at most one every five minutes.
2. **Maximum three tips per local calendar day.** Beyond that, the next ones are silently ignored and will return the following day.
3. **Never the same tip twice in the same session.** If you close and reopen the tab, a new session starts — but the daily counter persists across sessions.

State (date of last toast, daily counter, tips already shown) lives in `localStorage.izibuddy_tips_state`. You can clear it from the browser *Developer tools* if you want to restart the tip cycle — but it is rarely useful, the system forgets on its own after a few days.

39.5 — *Dismiss* and global toggle

On the right of each toast, a × closes it immediately ; to the left of the ×, a discreet *Don’t show again* button permanently disables the tip for your account. The choice is stored in `localStorage.izibuddy_tips_state`, key `dismissed`. Useful for users who do not like being given suggestions — they can reject each tip one by one.

If even this homeopathic dose bothers you, the `tipsEnabled` feature (category *Advanced*, active in all presets) is the master switch : disable it from the features panel (Chapter XXXVIII) and no tip will ever appear. The engine reads the value at every evaluation, the toggle takes effect immediately.

39.6 — Tips and learning

A philosophical question : *what is the point of a tip system in a manual that already explains everything ?* The answer is a matter of **timing**. The manual answers a reference need ; the tip answers an attention need : *should I care about this right now ?* The two complement each other — the tip introduces a feature at the right moment, the manual chapter gives the detail if you want to dig in. The *Learning* book (third volume) revisits the tip sequence and devotes a dedicated chapter to it.

For developers who want to add a tip associated with a new feature, the procedure fits in three steps : trigger the event via `IziTips.notify('eventName', payload)`, add an entry in `tips.json` (trigger, conditions, i18n keys for title/body, actions), and add the FR + EN translations.

Chapter XL — Language and keyboard shortcuts

Two cross-cutting topics that affect your daily comfort : the display **language** (seven languages, switchable at any moment) and the **keyboard shortcuts** that speed up handling the prompt list. This chapter recaps both, and explains why IziBuddy keeps two separate internationalization systems — one frontend, one backend.

40.1 — Switching language

IziBuddy ships **seven interface languages** — English, French, Dutch, Spanish, German, Italian and Portuguese — plus a **System** option that follows the browser/OS preference. Only languages that have a translation file (`strings_XX.json`) are offered (`src/static/index-i18n.js` , `languages` map). The switch happens from the profile icon in the top-right of the dashboard : a *Language* menu. The change is applied immediately, with no page reload : every visible label is re-translated live.

The chosen language is persisted in two places — `web.json` on the server side (to follow you across browsers) and `localStorage` on the client side (to apply the language at boot, before the server responds). The switching code lives in `src/static/index-i18n.js` .

The extension has its own language preference, independent of the dashboard. You can use the dashboard in one language and the extension in another. This is not a gimmick : some users prefer their authoring UI in one language and a minimalist injection UI in another — the dichotomy is honored.

40.2 — Two i18n systems, why

The dashboard is a composite web system : JavaScript on the browser side, Python on the server side. Each has its own strings to translate : button labels live in JavaScript, error messages returned by the API live in Python. IziBuddy therefore maintains **two separate sets** of translation files.

System	Files	Usage
Frontend (JS)	`src/static/lang/strings_fr.json`, `strings_en.json`	`I18n.t('key')` in JavaScript
Backend (Python)	`src/lang/strings_fr.json`, `strings_en.json`	`t('key')` via `from lang import t`

Both systems share the same syntax : keys are dotted paths (`templates.saved`, `categories.created`, `features.viewSplitPane.label`), placeholders are simple braces (`"templates.imported": "Imported {count} templates"`).

Why keep two separate sets rather than one ? Three operational reasons :

- **Reduce the JavaScript bundle size** — the server has technical error messages that the UI does not need to load.
- **Allow server-side messages even when the client is not instrumented** — an API client that does not load IziBuddy's JS still receives error messages in its language.
- **Keep API message translations consistent with what is logged server-side** — which simplifies support.

The golden rule restated in `CLAUDE.md` : **no hard-coded literals ever**, always an i18n key, and that key must exist in **both languages** simultaneously. A string added to `strings_fr.json` without its `strings_en.json` counterpart is a bug.

40.3 — General keyboard shortcuts

The usual conventions apply. The list below only records shortcuts specific to IziBuddy or useful to have at hand.

Shortcut	Action
`Ctrl+S`	Save the prompt being edited.
`Ctrl+Z` / `Ctrl+Y`	Undo / redo (`actionUndoRedo` feature); `Ctrl+Shift+Z` also redoes.
`Ctrl+C` / `Ctrl+V`	Copy-paste (in the list: `actionCopyPaste`).
`Ctrl+X` / `Ctrl+V`	Cut-paste (in the list: `actionCutPaste`).
`Ctrl+F`	Open the contextual Find / Replace bar.
`Del`	Send selected items to the trash (in advanced selection mode).

There is no `Ctrl+A` « select all » shortcut — *Select all* is offered through the context menu. `Esc` was intentionally removed from the prompt editor modal (to avoid losing edits); it still closes the category modal and the color picker.

Shortcuts that depend on the corresponding features disappear when you disable the feature. Disabling `actionUndoRedo` makes `Ctrl+Z` inert in the list — the text editor keeps its own local undo.

40.4 — *Split pane* shortcuts

The `viewSplitPane` feature opens the list as two side-by-side panes. Each one has its own selection, its own filter, its own open prompt. The specific shortcuts (file-manager convention):

Shortcut	Action
`Tab`	Switch the active pane (no modifier).
`F5`	**Copy** the active pane's selection into the other pane.
`F6`	**Move** the active pane's selection into the other pane.

`F5` and `F6` take no modifier — a modifier cancels them. Both operate on the active pane's current selection and target the other pane's current collection, and both are undoable through the undo manager. These shortcuts depend on the `actionCrossPaneF5F6` feature (*Actions* category, present in *Expert* only by default). If you regularly work in two panes, enable it from the features panel (Chapter XXXVIII).

40.5 — Multi-selection shortcuts

Multi-selection is covered in Chapter XIV. The keyboard shortcuts that relate to it :

Shortcut	Action
`Ctrl+click`	Add/remove an item to the selection.
`Shift+click`	Select a range of items.

Lasso selection (click in empty space, drag to surround, optionally extended with `Ctrl`) is controlled by the `actionRubberBand` feature. There is no `Ctrl+A` / `Ctrl+Shift+A` keyboard select-all in the list.

40.6 — Drag-and-drop

Drag-and-drop **moves** the dragged item into the target container. There are no modifier variants (`Ctrl` to copy, `Alt` to alias, `Shift` to force) — a drop is always a move. To copy a prompt, use the copy/paste actions or the F5 cross-pane copy in split view.

40.7 — The editor and autocompletion

The prompt editor (Chapter XVI) does not add a family of bespoke key chords (there is no `Ctrl+/` comment, `Ctrl+D` line duplication or `Tab` indentation binding). `Ctrl+S` saves; the general clipboard and undo shortcuts apply. **Autocompletion** for POO²L constructs is not opened with a chord — it triggers as you type the `@@@` sigil (`src/static/index-macro-autocomplete.js`), and `↑` / `↓` / `Enter` / `Esc` navigate the suggestions.

40.8 — Extension shortcuts and date format

The extension exposes a global shortcut defined in the manifest: `Ctrl+Shift+I` (`Command+Shift+I` on Mac) toggles the panel's visibility. You can redefine it from `chrome://extensions/shortcuts` or its Edge/Brave equivalent — useful if the developer-tools inspector conflicts with it.

The language also controls **date and number formats**: `13/05/2026` and `1 234,56` in French, `5/13/2026` and `1,234.56` in English. POO²L system variables (`@@@=DATE` and friends), in contrast, always stay in ISO format regardless of the language — an intentional choice to guarantee that templates remain reproducible across users (Chapter XLI).

Chapter XLI — System variables for templates

Your templates can contain chunks of text that change every time they are used — today's date, the name of the injecting user, the list of groups they belong to. Hard-coding these values would be painful and error-prone ; capturing them as parameters would be verbose and redundant. IziBuddy therefore exposes a set of **system variables** that are automatically substituted at render time. This chapter lists the available variables, states the single syntactic form, and explains how an operator can add a new one.

41.1 — The list of variables

Eleven variables are reserved by the POO²L engine and substituted on every prompt render. They are referenced with the substitution form `@@@=NAME` :

Variable	Description	Example
`@@@=DATE`	Date in ISO format (`YYYY-MM-DD`)	`2026-05-17`
`@@@=TIME`	Time in `HH:MM:SS` format	`14:32:00`
`@@@=DATETIME`	Date and time (`YYYY-MM-DD HH:MM:SS`)	`2026-05-17 14:32:00`
`@@@=YEAR`	Four-digit year	`2026`
`@@@=MONTH`	Month in `01` – `12` format	`05`
`@@@=DAY`	Day in `01` – `31` format	`17`
`@@@=WEEKDAY`	Day of the week, in the user's language	`Sunday`

Variable	Description	Example
`@@@=USER`	Username (login)	`jean.dupont`
`@@@=LANG`	ISO language code	`fr`
`@@@=GROUPS`	Comma-separated list of the user's groups	`dev, qa`
`@@@=GROUPS_ADMIN`	List of groups the user administers	`dev`

`USERNAME` exists as a **host-provided alias** of `USER`: the backend adds it to the values it hands the engine, but it is *not* one of the engine's reserved names. The eleven names above are the reserved set the engine (`pool-core`) recognizes; the values are computed by the backend in the render endpoints of `src/api/templates.py` and passed to the engine as its `context.system` map.

41.2 — One syntactic form

There is a single spelling: `@@@=NAME`. It is part of the POO²L grammar, so it blends into a prompt that already uses `@@@param`, `@@@{include }`, and so on. The double-brace `{{NAME}}` form that some web templating systems use does **not** exist in POO²L and is never substituted — a literal `8 octobre 2026` in a prompt body stays verbatim in the output, because the engine simply does not look for it.

41.3 — Substitution

Substitution is a **single pass**. The engine walks the resolved content once and replaces each `@@@=NAME` reference with its value. A consequence worth knowing: if a parameter's value itself contains an `@@@=` reference, that reference is **not** re-scanned or re-substituted — the pass does not loop back over the values it has already produced. Do not rely on nested substitution; compute the final text where you provide the value.

There is no separate « system variables are substituted last » stage : system variables, constants and parameters are all resolved as `@@@=name` references in the same pass, from the values the engine was given.

41.4 — Practical use cases

Timestamped signature. An email-drafting prompt can end with `Best regards, @@@=USER — @@@=DATE`. At injection time, the AI receives *Best regards, jean.dupont — 2026-05-17*.

Conversational context. A system prompt can start with *You are interacting with @@@=USER, member of groups @@@=GROUPS* ; the values are substituted with no user input.

Traceability. Putting `@@@=DATETIME` in a comment at the top of a prompt lets you locate in chat history the exact moment it was injected.

41.5 — Limits and guarantees

Three things to know about system variables :

- **The format is locked.** You cannot get `@@@=DATE` in `17/05/2026` format : the output is always `YYYY-MM-DD` regardless of the user's language. The motivation is reproducibility — a prompt injected by Alice in French and by Bob in English must produce the same render. If you want a localized date, declare a parameter and compute its value on the UI side.
- **`@@@=WEEKDAY` is an exception.** The day of the week is returned in the user's language (`Lundi` in French, `Monday` in English), because a day name only makes sense verbalized.
- **Values are computed server-side.** The extension receives already-rendered text. This guarantees that the clock used is the server's, not the machine's — useful when international teams work on the same instance and want a consistent timestamp.

41.6 — Unknown references

An `@@@=X` whose name the engine cannot resolve — neither a reserved system variable, nor a declared parameter or constant in scope — is **not** a warning that is quietly ignored. It is a **blocking** `UNDEFINED_REFERENCE` diagnostic : the render fails and the API refuses to expose the partial output. If you genuinely want the literal text `@@@=X` to appear in the render, wrap it in a verbatim block — `@@@verbatim … @@@end-verbatim` — which the engine passes through uninterpreted.

41.7 — Adding a variable (operator task)

Adding a new system variable is a server-side operation, so an operator task — see Part VIII. Because the engine owns the grammar, the steps are :

1. **Reserve the name in the engine.** Add it to the system-variable list in `pool-core` (`src/engine/crates/pool-core/src/parse.rs`) and rebuild the bindings (`src/engine/build_python.sh`, and `wasm-bindgen` for the browser build). A name that is not reserved would be treated as an undefined reference.
2. **Compute the value in the backend.** Add the entry to the `variables` dictionary in the render endpoints of `src/api/templates.py` ; the adapter routes reserved names into the engine's `context.system`. Keep the computation **deterministic** — never call an external API mid-render.
3. **Document it** in `CLAUDE.md`, in the *Variables Système pour les Templates* section.
4. **Add translations** if the value is localized (as `WEEKDAY` is).

Older templates that already contained the new syntax need no migration — they simply failed to resolve before, and will be substituted from the update onward.

41.8 — Variables and security

System variables are not secrets — their value can be guessed or observed. `@@@=USER` is your login, `@@@=GROUPS` the list of your groups. **Never** store sensitive information in these variables : no password, no token, no API key. If you need to inject such values, use a parameter entered at injection time (which stays local to your session) or a server-side secret managed by your integration.

This is an intentional boundary in IziBuddy : templates are shareable artifacts, and their system variables are public by construction.

Partie 7

Administration

Chapter XLII — The admin dashboard

IziBuddy administration does not live in a separate console. It is a panel of the same web application, exposed only to users carrying the system-administrator marker. If you have that marker, you see a few extra menu entries, a few extra buttons, and the diagnostics page. If you do not, those areas are invisible — the interface is exactly the same as a regular user's. This part of the book targets administrators and operators; it assumes that you have read the early chapters and that you understand the model of templates, groups, and containers.

42.1 — Where admin lives

Administration is served by the same routes as the rest — the distinction happens at authentication time. The `require_admin_user()` function in `src/api/admin.py` checks, for every sensitive call, that the current user is an admin. It calls `UserSettingsManager.is_admin(username)`, which **delegates to the `AuthManager`** (it is not read from the user's `web.json`). A non-admin gets a `403`; an admin passes through to the `/api/admin/*`, `/api/admin/alerts/*`, `/api/license/*` (write) and `/api/analytics/dashboard/system` endpoints.

Both on the backend and frontend, admin is not a separate role in the model; it is an attribute of the user held by the `AuthManager`. Promoting someone sets that attribute; demoting them clears it. No restart required, no session to invalidate — the next API call sees the new value.

42.2 — The first administrator

At install time, IziBuddy does not ask for a *root* password. The first user who registers automatically receives the admin marker. This happens before the database migration, at `UserSettingsManager`

initialization — see Chapter III for the install, and Chapter LII for the network hardening that prevents a third party from being the first to land on a fresh instance's registration page.

The corollary : a public deployment must close public registration long enough for you to create your account. The `registration.self_registration_enabled` key in `izibuddy.ini` controls this (see Chapter XLVIII).

42.3 — Admin menu entries

From the interface's point of view, promotion to administrator adds four extra areas.

The **Users panel** lists every account on the server, their groups, their admin status, and the date of their last connection. This is Chapter XLIII.

The **License panel** displays the current key, its status (one of `active`, `cancelled`, `expired`, `inactive`) and lets you enter a new one or deactivate it. The licensing integration (Lemon Squeezy) is **disabled for the initial release** — the default status is `inactive`. This is Chapter XLIV.

The **Analytics dashboard** offers three views : system, groups, and user. It aggregates timestamped events in SQLite and exposes CSV/JSON exports. This is Chapter XLV.

The **Diagnostics page** (`/static/diagnostic.html`) gathers system information, database status, performance counters, and the backend migration wizard. This is Chapter XLVII.

To these four panels are added global actions only the admin can trigger : server restart, test alert triggering, opening the license customer portal, and switching the database connector via `/api/admin/migration/config`.

42.4 — The « restart server » button

The admin can restart the backend without SSH access. The `POST /api/admin/restart` endpoint accepts a *countdown* in seconds (zero = immediate) and a `clear_console` flag. During the delay, `/api/admin/restart/status` lets the UI display a countdown visible to all connected users. The actual restart runs `os.execv(python, [python] + args)` — the same process restarts itself, keeping the Docker volumes and the PID.

This feature is precious after a database connector change or an `izibuddy.ini` modification that needs to be re-read. It is reserved for admins : if you restart while another admin is editing a prompt, their save will fail. The practice : announce in the room, wait a few seconds, restart.

42.5 — Group admin vs system admin

Do not confuse the two roles. The **group administrator** manages the members and templates of a specific group (Chapter XXV) ; they see nothing else. The **system administrator** carries the marker described above ; they have access to the `/api/admin/*` endpoints and see the global analytics dashboard.

The same user can be both — for instance, the server owner, who is also admin of a « leadership team » group. But the role distinction is crucial for auditability : a group admin cannot read the analytics of other groups, nor promote another system admin. The *analytics dashboard endpoints* (`src/api/analytics.py`) strictly enforce this separation : `_require_system_admin` for system views, and a combined check for the group view.

42.6 — Auditability of admin actions

Every sensitive action goes through a named *logger* and lands in the `logs/izibuddy.log` file. Restarts, connector switches, license activations, and admin promotions are logged with the identity of

the admin who triggered them. Read Chapter XLIX for rotation and formats (text vs JSON), and Chapter XLVI for the audit policy recommended in regulated environments.

Destructive actions (user deletion, prompt purge) are not reversible from the UI. Chapter L covers the backup strategy that lets you roll back if needed.

Chapter XLIII — User management

The Users panel is the second most-used screen after the templates page itself. It is the one that answers the recurring question : who has access to this instance, in which groups, with which rights ? It relies on the server-side `AuthManager` for accounts and admin status, and on the `UserSettingsManager`, which holds for each account a `~/.izibuddy/users/{username}/` directory containing the web preferences, the extension preferences, and group memberships.

43.1 — Listing users

The admin menu's « Users » button calls `GET /api/auth/users` (served by `src/api/auth.py`), which returns the accounts known to the `AuthManager`. Admin status is managed through a companion endpoint, `GET /api/users/admins` (`src/api/users.py`), which the panel uses to show and change who is an administrator. For each account you see the username, the admin flag and the group memberships.

There is no `GET /api/admin/users` endpoint — the panel reads `/api/auth/users`, as the frontend code makes explicit (`index-admin.js` calls `/api/auth/users`).

43.2 — Creating a user

The primary path is **self-registration** : the visitor clicks « Create an account », enters their credentials, and (when email verification is enabled) confirms through a link. This flow is driven by the `[registration]` section of `izibuddy.ini` — you can close it (`self_registration_enabled = false`) or restrict it to one or several email domains, see Chapter XLVIII. The endpoints are `POST /api/auth/register` and, for the email-verified variant, `POST /api/auth/register/initiate` then `POST /api/auth/register/complete`.

For onboarding several people at once, use **instance invitations** (Chapter XXVI), which are reserved to system admins. There is no admin-side « create a user with a provisional password » form : the admin governs who is an admin, not the setting of other users' passwords.

43.3 — Editing a user and admin status

The admin panel's main lever on an account is its **admin status**. Granting admin goes through `POST /api/users/admins/{username}` ; revoking it through `DELETE /api/users/admins/{username}` . The `UserSettingsManager.is_admin` check that guards every sensitive endpoint delegates to the `AuthManager` , so the change takes effect on the next call.

Passwords are self-service. A user changes their own password via `POST /api/auth/change-password` , and a forgotten password is recovered through `POST /api/auth/password-reset/initiate` (which issues a one-time token, delivered by email when SMTP is configured) followed by the reset-completion step. The admin does not set or see other users' passwords.

43.4 — Promoting and demoting an admin

Promotion and demotion are instant — no restart, no logout. The next API call from the target user's session sees the new rights, and the admin areas appear or disappear at the next page refresh. Admin status is not a group role and not a `web.json` boolean ; it is held by the `AuthManager` and toggled through the `/api/users/admins/...` endpoints above.

Security convention : **keep at least two admins** on a shared instance. The loss of access of a single admin can block critical actions (license, restart, migration). If you accidentally demote the last admin, recovery goes through command-line intervention on the server rather than through the UI.

43.5 — Deleting a user

Deleting an account goes through `DELETE /api/auth/users/{username}` and requires an explicit confirmation. It removes the account from the `AuthManager` and its `~/.izibuddy/users/{username}/` settings directory.

This operation is irreversible from the UI. If you triggered it by mistake, restore the latest backup (Chapter L). In practice, prefer keeping historical accounts inactive over deleting them when you need to preserve an audit trail (Chapter XLVI). Note that the analytics anonymization-on-deletion described in some designs is **not currently wired** (Chapter XLV).

43.6 — The role of groups in user management

Groups have two levels only : **admins** and **members** — there is no « editor » or « reader » role (`groups_manager.py`). The Users panel shows a user's groups but does not manage them directly. To create a group, invite or remove a member, or change someone's group-admin status, you go through the Groups panel (Chapter XXV).

This is intentional. Group membership management is also open to group admins (who are not necessarily system admins), and it therefore lives in its own panel, with its own permission matrix.

43.7 — Password policy

IziBuddy enforces a minimum of **6 characters** (`core/auth.py`). Beyond that, no constraint is imposed by default (the username itself must be at least 3 characters and alphanumeric, with `-` and `_`). For regulated environments, prefer an **SSO** upstream of the reverse proxy (Chapter LII) — with its own password and multi-factor policies — over a strict local password policy : it is simpler to maintain and more robust.

Chapter XLIV — License management

IziBuddy carries a licensing integration built on **Lemon Squeezy**, but it is **disabled for the initial release** : the default license status is `inactive` and no key is required to run the product. This chapter describes the mechanics as they exist in the code — activation, lifecycle, states — with the clear caveat that reactivation is planned for a later release, not wired for this one.

44.1 — The license model

The license is an alphanumeric key issued by the distributor through Lemon Squeezy, IziBuddy's payment provider. It is tied to an *instance* — not to a user. The `src/api/license.py` module implements the server-side surface : `GET /api/license/status` , `POST /api/license/activate` , `POST /api/license/deactivate` , `GET /api/license/portal` , `GET /api/license/checkout/{variant_id}` , and the WEBHOOK `POST /api/webhooks/lemonsqueezy` (with a `GET /api/webhooks/lemonsqueezy/test` reachability probe).

The `LicenseManager` (in `core/license_manager.py`) stores the license in `~/.izibuddy/config/license.json` . The key is kept **in clear** in that JSON file — it is not encrypted. It knows **four** statuses :

- **active** — valid key, current subscription.
- **cancelled** — the subscription was cancelled.
- **expired** — the subscription has lapsed.
- **inactive** — no active license (the default, and the state the product ships in for this release).

Two **plans** govern capacity : **freemium** (a **10-template limit**) and **pro** (no limit). With licensing disabled, instances run on the freemium plan.

44.2 — Entering a license

The License panel is accessible only to system admins (see Chapter XLII). The form takes the key and an activation button. On the backend, `POST /api/license/activate` validates the key format, queries Lemon Squeezy, and — if the response is positive — persists the key and flips the status to `active`. If the validation call fails, activation fails with an explicit message rather than hiding the error.

The status displayed in the panel is refreshed at every opening via `GET /api/license/status`. It reports the plan (freemium/pro), the status (active/cancelled/expired/inactive) and, when relevant, the expiration date and a link to the customer portal.

44.3 — Renewing and transferring

While the integration is active, Lemon Squeezy notifies IziBuddy of subscription events by webhook (renewal, cancellation, refund), and the `LicenseManager` updates the recorded status accordingly.

Transferring a key to another instance is a two-step move : **deactivate** it on the source instance (`POST /api/license/deactivate`), then **activate** it on the target instance. Deactivation frees the slot.

Upgrading a plan is done through the **customer portal** : the panel button opens the URL returned by `GET /api/license/portal`, a session into your Lemon Squeezy account where you change the plan, update billing and download invoices. A one-off checkout can be started through `GET /api/license/checkout/{variant_id}`.

44.4 — The Lemon Squeezy webhook

The `POST /api/webhooks/lemonsqueezy` endpoint receives the subscription lifecycle events (`order_created`, `subscription_created`, `subscription_updated`, `subscription_cancelled`, `subscription_expired`, `license_key_created`); any other event name is

logged and ignored. The signature is verified against the shared secret configured in `config/lemonsqueezy.yaml` (which supports `${VAR}` environment substitution).

A security note for whoever reactivates this path : the endpoint **fails closed** — a request whose signature header is missing *or* invalid is rejected with a `401` (`src/api/license.py`). With no webhook secret configured (the state this release ships in), the check fails closed and every webhook is refused, so the route is inert until Lemon Squeezy is wired up and the shared secret is set. Chapter LII explains how to expose only this route to Lemon Squeezy while keeping the rest of the public surface strict.

44.5 — Behavior on the freemium plan

Without an active license, IziBuddy runs on the **freemium** plan : every feature is available, capped at **10 templates**. There is no separate « community » or « trial » or « grace » mode — those states do not exist in the model. Activating a valid `pro` license lifts the prompt limit ; deactivating or letting it expire returns the instance to the freemium cap.

44.6 — Backing up and restoring the license

The `license.json` file is part of the instance's configuration state and should be included in your backup strategy (Chapter L). Because it is a plain file, restoring a backup restores the recorded license state as-is ; if you move to new hardware you complete the transfer on the Lemon Squeezy side as described above. Again, none of this is active in the initial release — it is documented for the planned reactivation.

Chapter XLV — Analytics and event log

For each instance, IziBuddy records an event log that serves two purposes : feeding an admin-facing dashboard, and providing an exportable audit trail for environments that require one. The system is intentionally modest — no page-by-page tracking, no heatmaps, no third-party shipping — but it covers every business event useful to honest usage feedback.

45.1 — The data model

The tracker writes each event to a **JSONL** log (one event per line, one file per day) for fast, crash-tolerant appends. A separate **SQLite** database (`analytics.db`) holds the aggregated tables (`daily_stats`, `top_items`, `summaries`) that the dashboard queries. That database is populated by an aggregation routine (`AnalyticsAggregator.aggregate_day`) — which, as noted below, exists in the code but is **not currently scheduled** to run on its own. The architecture lives in `core/analytics/`.

The main event types tracked today include :

- `template_view` — a prompt has been opened in the editor.
- `template_create` — a new prompt has been saved.
- `template_edit` — an existing prompt has been modified.
- `template_instantiate` — a prompt has been rendered (typically before injection by the extension).
- `template_share` — a prompt has been shared or duplicated.
- `user_login` / `user_login_failed` — login successes and failures.
- `group_create` / `group_join` — group lifecycle events.

Each event carries a timestamp, a user id, a *correlation id* (the same one propagated in HTTP logs, see Chapter XLIX), and a bag of type-specific attributes (`template_id`, `version`, `preset`, `sections`).

45.2 — The analytics dashboard

The admin menu contains a *Dashboard* tab that presents, over a sliding 30-day window by default, four cards: total events, active users, templates created, total instantiations. This view is served by `GET /api/analytics/dashboard/system` and is strictly reserved to system admins — the `_require_system_admin()` function in `src/api/analytics.py` rejects any other identity.

Two secondary views complete the picture. The **per-group** view (`/api/analytics/dashboard/groups` then `/api/analytics/dashboard/group/{id}`) lists groups sorted by event volume and, when a group is selected, gives the member-by-member detail. This view is accessible to group admins for their own group only. The **per-user** view (`/api/analytics/dashboard/user/{username}`) is accessible to the user themselves and to an admin; it counts personal actions without revealing prompt content.

The time window is tunable from 1 to 365 days through the `days=` parameter. Beyond that, IziBuddy falls back on quarterly aggregates stored in SQLite — faster to query, less fine-grained.

45.3 — CSV and JSON export

Two endpoints cover raw export: `GET /api/analytics/export/csv` and `GET /api/analytics/export/json`. Both accept the same filters: `start_date`, `end_date`, `template_ids` (comma-separated list), `event_types`. The response is a downloadable file with a *Content-Disposition* attachment — useful to open directly in Excel or to ingest into a BI pipeline.

For more polished dashboards (with formatting, totals and charts), `GET /api/analytics/export/dashboard/system` produces **XLSX** and **PDF** outputs in addition to CSV. The `format=` parameter determines the rendering. These exports are designed to serve as a monthly report for non-technical stakeholders.

The `[analytics]` section of `izibuddy.ini` carries auto-export keys (`auto_export_enabled`, `auto_export_format`, `auto_export_path`), but **no scheduler currently reads them** — automatic nightly export is not wired in this release. For now, treat exports as an on-demand operation through the endpoints above ; feed an archiving system by scripting those calls yourself.

45.4 — Tracking from code

If you extend IziBuddy with a business module, you can add your own tracking. The API is deliberately simple :

```
from core.analytics import track_event

track_event(
    "template_instantiate",
    user="alice",
    template_id="abc123",
    data={"preset": "long", "sections": ["intro", "body"]}
)
```

`track_event` is exported from `core.analytics` ; the `EventType` enum is not — pass the event type as a string. The write is **synchronous** but cheap (a JSONL append) ; on write error the tracker logs the incident to `izibuddy.log` and does not raise, so an analytics failure never blocks your business code.

For ad-hoc needs, the `POST /api/analytics/track` endpoint records an event from an external client. It requires the `X-IziBuddy-User` header and a `template_id` in the body, alongside the `event_type` — handy to trace application events from the extension or a business script.

45.5 — Retention and purge

The intended retention is expressed by `analytics.retention_days` in `izibuddy.ini` for the raw JSONL events, with the SQLite aggregates kept indefinitely. The purge routine (`cleanup_old_events`) and the aggregation routine (`aggregate_day`, `analytics.aggregation_time`) both exist in the code — but **neither is currently scheduled**: no background job calls them, and `aggregation_time` is not read anywhere. In this release, aggregation and purge are effectively manual/administrative operations, not automated nightly jobs. Treat this section as the design intent, and do not rely on automatic purge or aggregation happening on its own.

45.6 — Privacy considerations

IziBuddy analytics record who did what on which prompt, but **never the content** of the prompt or of the AI's response. No string typed by the user in the editor is copied to the log. This is a deliberate choice, motivated by GDPR requirements and by the wish to reduce the attack surface in case of log leak.

The `docs/PRIVACY.md` document details the full matrix: collected fields, legal basis (legitimate interest of the operator), retention duration, access rights. Publish it or an equivalent in the *privacy policy* of your instance, especially if you open it to users outside your organization.

Automatic anonymization of a deleted account's past events (replacing the `user_id` with a stable hash) is a documented design intent but is **not currently wired** — deleting a user does not retroactively rewrite past analytics events. Plan any anonymization requirement as a manual/administrative step for now. See Chapter XLIII §deleting a user.

Chapter XLVI — Audit and alerts

The event log from Chapter XLV serves first to understand usage ; it also doubles as an **audit trail** when your environment requires it. This chapter describes what is logged from a security standpoint, how alerts are computed and dispatched, and which policy to apply in a regulated environment.

46.1 — What is logged for audit

Beyond business events, IziBuddy records a second category of events with a purely *audit* purpose : successful and failed login attempts (with source IP address when available), admin promotions and demotions, user creations and deletions, license activations and deactivations, restarts triggered from the UI, and database connector changes.

These events live in two places. On the **log** side (Chapter XLIX), each action is traced by the structured logger with its `correlation_id`, the admin's identity, and its context (target, action detail). On the **analytics** side (Chapter XLV), a subset is stored in SQLite to allow queries such as *« how many login failures on this account in the last 24 hours »*. The two trails do not substitute for each other : logs have rotation and short retention, analytics have long retention but less textual detail.

46.2 — The alerting system

The `core/alerts/` module continuously inspects the event stream and triggers an alert whenever a predefined rule is crossed. Seven types are provided by default :

- `error_rate` — more than N errors per hour (50 by default).
- `failed_login` — N consecutive login failures (5 in 15 min).
- `brute_force` — more aggressive variant (20 in 5 min).

- `slow_requests` — requests above the `logging.slow_request_threshold_ms` threshold (1,000 ms).
- `user_locked` — an account has just been locked.
- `security_event` — business-level security event (to be emitted from application code).
- `system_error` — non-recoverable system error.

Each alert carries a severity (info, warning, critical), a unique id, a message, and a bag of context data. They are persisted in an alerts table viewable from the Admin → Alerts panel.

46.3 — Notification channels

The `[alerts]` configuration in `izibuddy.ini` accepts a list of channels to activate simultaneously (`channels` key, comma-separated values) :

- **log** — always active, writes a line `[ALERT] <type> - <message>` in `izibuddy.log` — for instance `[ALERT] error_rate - 55 errors detected in the last hour`.
- **email** — sends a formatted email to the `admin_emails` list. Requires the `[email]` section to be configured.
- **webhook** — POSTs JSON to the `webhook_url` URL. The body follows a Slack/Discord/Teams-compatible convention :

```
{
  "text": "🚨 IziBuddy Alert",
  "attachments": [{
    "color": "danger",
    "title": "Error Rate Alert",
    "text": "55 errors in the last hour",
    "ts": 1704621045
  }]
}
```

A *cooldown* (`alert_cooldown_minutes`, 30 by default) prevents spam : if the same alert triggers several times in the window, only the first one is dispatched — the next ones are aggregated and logged.

The Admin → Alerts panel lists unacknowledged alerts and lets you acknowledge them one by one (`POST /api/admin/alerts/{id}/acknowledge`) or in bulk (`POST /api/admin/alerts/acknowledge-all`). An acknowledged alert stays in the database for audit but no longer appears in the active list. Old acknowledged alerts can be purged manually : `get_alert_manager().cleanup_old_alerts(days=90)`.

46.4 — Testing an alert

The `POST /api/admin/alerts/test` endpoint generates an artificial alert that runs through the whole chain (channels included). This is the right reflex after a configuration update : send a test alert, verify that it lands in Slack or in the admin's mailbox, adjust if necessary.

On the code side, your business module can emit an alert directly :

```
from core.alerts import get_alert_manager

mgr = get_alert_manager()
mgr.trigger(
    alert_type="security_event",
    severity="critical",
    message="Privilege escalation attempt detected",
    data={"username": "bob", "endpoint": "/api/admin/restart"}
)
```

The call is non-blocking : dispatching over the channels happens in the background.

46.5 — Audit policy for regulated environments

In an environment subject to regulatory requirements (SOC2, ISO27001, HDS, banking sector), IziBuddy provides the raw material but does not impose a policy. Here is the recommended configuration to adopt in production :

- **JSON format for logs** (`logging.format = json`) so they can be ingested by a centralized sink (ELK, Splunk, Datadog). Text format is reserved for development.
- **Long analytics retention** (`analytics.retention_days = 730` for two years) and indefinite preservation of SQLite aggregates.
- **Scheduled CSV export** to an external directory read by a third-party archival system. Note that IziBuddy does not currently run this export on its own (Chapter XLV), so drive it from an external scheduler calling the export endpoints.
- **Redundant alert channels** : `channels = log, email, webhook` . At least one external channel to avoid depending on the IziBuddy server itself in case of failure.
- **Named service accounts** rather than shared. The audit log is useless if everything goes through a generic `admin` account.
- **Upstream SSO switch** (Chapter LII) to benefit from your identity provider's multi-factor authentication, password policies and session expiration.

Chapter XLIX details integration with an external log aggregator.

46.6 — Reading alerts through the API

The following endpoints are available for external integrations (CMDB, ticketing, monitoring) :

- `GET /api/admin/alerts` — paginated list filtered by type, acknowledgement status.
- `GET /api/admin/alerts/recent` — recent unacknowledged alerts.

- `GET /api/admin/alerts/stats` — counters by type and severity.
- `GET /api/admin/alerts/config` — current configuration (useful to verify after a restart).

All require a system admin (`X-IziBuddy-User: <admin>`). Responses are JSON, with a structure documented in `src/api/alerts.py`.

46.7 — Limits and complements

The IziBuddy alerting system covers application events ; it does not replace infrastructure monitoring (CPU, memory, network, disk). A well-equipped instance combines :

- IziBuddy for business alerts and auth failures.
- The operating system (auditd, journald) for kernel and file access events.
- A process supervisor (systemd, Docker healthchecks) for restarts and crashes.
- A log aggregator for cross-source correlation.

Chapter LII describes how to expose the `/api/status` page (public, no authentication) to an external monitoring tool to check application availability.

Chapter XLVII — The diagnostics page

`/static/diagnostic.html` is the least visible and most useful page in IziBuddy. It is not part of the main menu — it lives at its own URL, accessible only to authenticated admins, and concentrates everything you need to know about an instance's state to solve a remote support ticket. If you open a case with the publisher or diagnose an incident on your own deployment, this page is your starting point.

47.1 — How to reach it

Two paths. The first : from the admin menu, the *Diagnostic* entry opens the page in a new tab, forwarding the auth token through `localStorage.izibuddy_token_temp`. The second : typing the URL `https://<your-instance>/static/diagnostic.html` directly in the browser. Either way, the page requires a valid admin token ; without that token, it shows an error screen and invites you to log in from the main dashboard.

The page is intentionally served as static HTML — no server template, no JS engine from the main application. This allows loading it even when the rest of the application has a problem (broken rendering, blocking JS error). It queries the backend through the same REST API as the dashboard, but stays functionally autonomous.

47.2 — The report's structure

The `DiagnosticReport` returned by `GET /api/dev/diagnostic` aggregates these sections (see the `DiagnosticReport` model in `src/api/diagnostic.py`) :

- **server** — IziBuddy version, uptime, Python version, connected frontends, installed packages.

- **system** — operating system and version, CPU architecture, hostname, network info, CPU count/percent, memory usage.
- **storage** — information about the prompt repository (path, connector type, Git repository state if applicable).
- **users_stats** — user counts (total, admins, …).
- **content** — templates and containers (collections, folders).
- **versions** — statistics of the version index.
- **components** — presence flags for the internal managers (`config_manager`, `templates_manager`, `analyzer`, `concatenator`, `backup_manager`) plus the templates and Git status.
- **disk** — disk usage of the application directory, percent used, status (ok / warning / critical).
- **recent_errors** — the latest **ten** error lines read from the log (truncated).
- **config_status** + **config_file** — the configuration state and the raw content of `izibuddy.ini`.

Each category has its own *status* (ok / warning / critical / error) that propagates to a global status displayed at the top of the page. A single `critical` is enough to turn the banner red.

47.3 — What the components check actually does

There is **no network health-checking**. The `components` section does not open a database connection, list an object-storage bucket, run an SMTP handshake, `git ls-remote` a mirror or query the license server. It reports whether the in-process managers are initialized (booleans such as `config_manager is not None`), plus the local templates and Git status. The report is therefore fast and side- effect-free : it inspects the running process, it does not probe external services.

If you need to verify that a database, SMTP server or Git remote actually answers, do it from the connector’s own configuration screen or from the host, not from this report.

47.4 — The database migration wizard

The page embeds a *Database Migration Wizard* tab that guides switching from one database connector to another. This is the graphical interface for `POST /api/admin/migration/config` (Chapter XLII §restart button). The wizard lets you :

- see every supported connector (filesystem, SQLite, PostgreSQL, MySQL, MongoDB, Oracle, DB2, S3, GCS, Azure, MinIO, CockroachDB, Elasticsearch, MS Access),
- distinguish those that are *enabled* (configured in `izibuddy.ini`) from those that are only *visible* (no section in the `.ini`),
- enter the target connector's parameters (host, port, database, credentials),
- test the connection before switching,
- export the templates from the old backend and import them into the new one,
- update `izibuddy.ini` automatically (with a backup of the previous version).

The final migration requires a server restart — which you can trigger from the same page. The detail of available connectors is covered in Part IX of the manual (Chapter LV).

47.5 — Database export and import

Beyond the wizard, two modals offer the basic import/export operations :

- **Export Database** — produces a dump in the chosen format (compressed JSON `.json.gz` or SQLite `.db`). Options : include Git history, include templates. The file is written on the server in `backups_dir` and a download link appears when the operation completes.
- **Import Database** — accepts the same format as input. The « Clear existing data » option replaces the current state ; wi-

thout it, the import is additive. The admin must confirm explicitly — an additive import on an already-populated database may produce duplicates.

These operations are distinct from the automatic backups of Chapter L ; they are designed for instance transfers rather than for disaster recovery.

47.6 — Copy the report

The « Copy Report » button serializes the entire JSON report to the clipboard. This is the format to paste into a support ticket : it carries the context a supporter needs.

Security note — read before sharing. The `config_file` section contains the **raw** content of `izibuddy.ini` and is **not** secret-masked : `get_config_file_content()` in `src/api/diagnostic.py` returns the file as-is. Any secret present in your `.ini` (SMTP password, database credentials, a webhook secret, …) will appear in the copied report. Before pasting the report into an external channel, review it and redact those values by hand. (The separate database-config endpoint does mask credentials, but the diagnostic report's `config_file` does not.)

47.7 — A page that loads on its own

The value of serving the page as static HTML is that it **loads even when the rest of the application is broken** (broken rendering, a blocking JS error) : the page does not depend on the main app's scripts. It still needs the REST API to populate the report — if the backend answers nothing, the report cannot be built, and you are already at the point of looking at the logs on the host. There is no separate browser-only « degraded report » mode ; the page simply reports that it could not reach the backend.

Partie 8

Operations

Chapter XLVIII — The `izibuddy.ini` configuration file

IZIBUDDY.INI is the main server configuration file. It lives at the root of the application's working directory — the same root from which you launch `./izibuddy` or which the Docker container points to. Its structure follows the classic Python `configparser` format : named sections in brackets, `name = value` keys inside, comments prefixed by `#` or `;`. This chapter reviews each section, its purpose, and the key keys it contains.

48.1 — Overview and reload policy

The file is read **at startup** only. For a change to take effect, you must restart the application — either through the « Restart server » button of the admin panel (Chapter XLII §restart button), or through your process manager (systemd, Docker Compose). There is no *hot reload* in production, for simplicity's sake and to avoid inconsistent states while a request is in flight.

Two notable exceptions. The **database** configuration : it is read at every startup *and* when the migration wizard switches (Chapter XLVII), which writes to the file before restarting. The `GET /api/admin/alerts/config` endpoint exposes the current alert thresholds **read-only** — it does not change them ; a new threshold still requires editing `izibuddy.ini` and restarting. **User settings** (theme, language, UI preset) are not in `izibuddy.ini` — they live in `~/.izibuddy/users/{username}/settings/web.json`.

The example file shipped at the repository root contains French comments that document each key. Keep those comments when you make changes — they will save you trips back to the documentation at a future restart.

48.2 — `[server]` section — network listening

```
[server]
port = 8765
host = 0.0.0.0
workers = 4
```

`port` is the FastAPI backend's TCP listening port. `host` is the bind address : `127.0.0.1` for an instance accessible only locally (recommended behind a reverse proxy), `0.0.0.0` to listen on every interface (recommended in a container). `workers` is the number of Uvicorn processes — leave 4 as a starting point, raise it based on available CPUs and your load profile. Note that in the current build `workers` is read but not applied — the server runs a single Uvicorn process ; scale out by running several instances behind the reverse proxy instead.

Chapter LII explains how to align these values with a Caddy or Nginx reverse proxy.

48.3 — `[paths]` section — data tree

```
[paths]
templates_dir =
backups_dir = backups
output_dir = output
logs_dir = logs
```

In the current build these keys are largely inert. Logs are written to `~/.izibuddy/logs` and backups to `~/.izibuddy/backups` regardless of `logs_dir` / `backups_dir`, and an empty `templates_dir` defers to the directory configured by the `[database]` section. What actually holds persistent state is therefore the `~/.izibuddy/` tree : in containerized production it must point inside a persistent volume (see Chapter LII) — otherwise your backups and logs will disappear at image rebuild.

48.4 — `[processing]` section — processing limits

```
[processing]
max_tokens_summary = 4000
max_tokens_concat = 100000
default_encoding = utf-8
```

Internal limits used by prompt summarization and concatenation operations. The default values suit most uses. `default_encoding` is applied to files read from disk — keep `utf-8` except in extreme cases.

48.5 — `[ui]` section — interface parameters

```
[ui]
check_status_interval = 5000
queue_stats_interval = 2000
theme = light
```

`check_status_interval` and `queue_stats_interval` are the poll periods in milliseconds for server-status widgets. Shorter = more responsive, but heavier. `theme` is the default visual theme for new users (`light`, `dark`, `system`). Each user can then override that value in their preferences.

48.6 — `[logging]` section — logging

```
[logging]
level = INFO
format = text
rotation_enabled = true
rotation_max_size_mb = 50
rotation_backup_count = 10
access_log_enabled = true
slow_request_threshold_ms = 1000
```

Every key is detailed in Chapter XLIX. Summary : `level` filters by severity, `format` chooses between human text and JSON for ELK ingestion, `rotation_*` controls file rotation, `access_log_enabled` enables HTTP request logging, `slow_request_threshold_ms` marks slow requests for the alerting system.

48.7 — `[alerts]` and `[analytics]` sections

```
[alerts]
enabled = true
channels = log
error_threshold_per_hour = 50
failed_login_threshold = 5
failed_login_window_minutes = 15
webhook_url =
admin_emails =
alert_cooldown_minutes = 30
```

Detailed in Chapter XLVI. The `channels` key accepts a comma-separated list among `log`, `email`, `webhook`. Thresholds control the alert triggers.

```
[analytics]
retention_days = 365
aggregation_enabled = true
aggregation_time = 02:00
auto_export_enabled = false
auto_export_format = csv
auto_export_path = exports/analytics
```

Detailed in Chapter XLV. `retention_days` is the retention duration for raw JSONL events ; SQLite aggregates are kept indefinitely. `aggregation_time` is intended as the local hour of the nightly aggregation *job*, but that scheduled job is not wired in the current build (the key is not read at runtime).

48.8 — `[backups]` section — backup policy

```
[backups]
enabled = true
max_count = 50
auto_archive = false
archive_after_days = 30
archive_after_versions = 100
```

Detailed in Chapter L. `max_count` is intended to cap the number of backups kept, and `auto_archive` / `archive_after_*` to enable a long-term archival tier. Be aware that in the current build these keys are **declarative only** — no `max_count` enforcement and no archival tier are wired ; the one active backup path is the pre-import snapshot.

48.9 — `[registration]` and `[email]` sections

```
[registration]
self_registration_enabled = true
allowed_domains =
verification_link_expiry_hours = 168
password_reset_expiry_hours = 24
```

Controls registration. `self_registration_enabled = false` closes self-registration — recommended in production. `allowed_domains` restricts authorized emails to a list of domains. The two `*_expiry_hours` keys set the lifetime of emailed links.

```
[email]
enabled = false
smtp_host = smtp.example.com
smtp_port = 587
smtp_user =
smtp_password =
smtp_use_tls = true
```

```
smtp_from_email = noreply@example.com
smtp_from_name = IziBuddy
```

SMTP configuration for invitations, email verifications and alert notifications. If `enabled = false`, links are printed to the server console for testing. In production, set `enabled = true` and provide valid credentials.

48.10 — `[database]` section and its subsections

```
[database]
active = multi-user-with-git
```

`active` designates the current connector ; its value must match a `[database.<name>]` subsection present in the file. Every declared subsection defines a backend *available* to the migration wizard. Supported subsections : `[database.multi-user-with-git]` (the default, JSON + Git), `[database.fs-multi-user]`, `[database.fs-single-user]`, `[database.sqlite]`, `[database.mysql]`, `[database.postgresql]`, `[database.oracle]`, `[database.db2]`, `[database.cockroachdb]`, `[database.mongodb]`, `[database.elasticsearch]`, `[database.access]`, `[database.s3]`, `[database.gcs]`, `[database.azure]`, `[database.minio]`.

Each subsection carries the keys specific to the backend (`host`, `port`, `database`, `user`, `password`, or `path`, or `bucket`, `region`, `access_key`, `secret_key` depending on the case), plus backend-specific flags such as `secure` on `[database.minio]` (the TLS toggle). Two cross-cutting subsections complete the block : `[database.replication]` (`enabled`, `slaves`, `health_check_interval`, `read_from_master`) mirrors writes to replica backends, and `[database.cache]` (`enabled`, `redis`, `prefix`) fronts reads with a cache. The per-backend detail is the subject of Chapter LV.

48.11 — The `[git]` section

`[git]` configures the auto-sync behaviour of the file-based Git mirror. The whole section applies **only** when no `[database]` sub-section overrides the default `FileSystemDatabase` storage — on installs with a real DB configured, the keys here are ignored and the Git UI is hidden (Chapter XXVII §when git is applicable). The values shipped with `izibuddy.ini` are the recommended defaults ; tune them only if you run the file backend across two machines via a shared remote.

Key	Default	Effect
`auto_push`	`true`	Push to the remote after every auto-commit. Set to `false` to keep commits local and push manually from the *Git Settings* modal.
`pull_strategy`	`ff-only`	Strategy used by manual *Pull* and by the auto-sync hooks. `ff-only` refuses merges (recommended multi-backend), `rebase` replays local commits over the remote, `merge` accepts merge commits on divergence.
`sync_on_startup`	`true`	Fetch + pull once when the backend starts, catching up on commits pushed from another instance or directly on the forge.
`sync_on_read`	`false`	Fetch + pull before every prompt read (open, copy, render). Only useful when two file backends share one git remote — adds ~100-500 ms per action.
`sync_throttle_seconds`	`10`	Minimum delay between two fetches triggered by `sync_on_read` ; protects against rate-limits on rapid actions.

All five keys are surfaced in the *Auto-sync* block of the *Git Settings* modal for admins. See Chapter LIX §multi-backend behaviour for the flow they wire up.

48.12 — Auxiliary sections

Four small sections govern application behaviours. `[i18n]` sets `default_language` (`en` by default; fr, en, es, de, it and pt are supported). `[templates]` carries the include-diamond policy: `default_diamond_policy` (`merge` or `refuse`; `refuse` by default) and `allow_user_diamond_override` (`true` by default), which lets a prompt's own `structure_policy` take precedence over the global setting. `[attachments]` selects the attachment payload store via `storage_backend`. `[features]` exposes `allow_no_scoring` (`true` by default), which allows the per-template **Note** flag (quality-scoring opt-out). The key is unchanged.

`[projects]` and `[current]` are internal-state sections — you generally do not need to modify them. They track the current project and the list of known projects in deployments that use the projects feature.

A handful of module-level defaults in `src/config/settings.py` read environment variables (`LOG_LEVEL`, `DEBUG`, `ENV`, `PORT`), but the running server reads its port, host and logging from `izibuddy.ini` through its config manager — so **`izibuddy.ini` is authoritative** for those values. There is no general environment-variable override layer over the file; to change the listening port or host, edit the file and restart.

48.13 — Conventions and best practices

- **Version `izibuddy.ini` but not the secrets.** Keep the SMTP password and database passwords out of version control — redact them in the committed copy and supply the real values only in the deployed file (or through your secret manager's file provisioning).

- **Document your changes** with `#` comments above each non-default key, with date and author. The file is read rarely but re-read after an incident — every line must be explainable.
- **Back up before each change.** The migration wizard does it automatically (`izibuddy.ini.YYYYMMDD_HHMMSS.bak`); do the same for manual changes.
- **Test in a pre-production environment** before applying to production, especially for database connector or SMTP configuration changes.

Chapter XLIX — Logging

Logs are the first thing an operator looks at when something goes wrong. IziBuddy provides a structured logging system capable of writing in two formats (human text or JSON for ELK/Splunk), with automatic rotation, a *correlation id* propagated across an entire request's chain, and optional integration with an external log aggregator.

49.1 — The log files

By default, IziBuddy writes to two files, both under `~/.izibuddy/logs` (the `logs_dir` key of `[paths]` is not honored in the current build — see Chapter XLVIII) :

- `izibuddy.log` — main application log. Every business operation, error, alert, restart. This is the reference file.
- `access.log` — HTTP request log. Every incoming request produces a line with method, path, status, duration. Toggleable through `logging.access_log_enabled`.

As a complement, the `analytics` module writes its raw events as JSONL in `analytics/YYYY-MM-DD.jsonl` (see Chapter XLV — these are business events, not logs in the classical sense).

49.2 — Levels and filtering

The `logging.level` key accepts the standard values of the Python library : `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`. The recommended value is `INFO` in production, `DEBUG` only to reproduce a reproducible incident. `DEBUG` mode produces a high volume — make sure your rotation keeps up, otherwise you will saturate the disk in a few hours on a busy instance.

A few rules of thumb worth knowing :

- `INFO` — normal events (startup, request served, prompt saved).
- `WARNING` — non-blocking incidents (slow request, webhook failure, expired token).
- `ERROR` — blocking incidents (save failure, database unreachable).
- `CRITICAL` — integrity incidents (detected corruption, emergency shutdown).

Filtering is done at the global level, but each module has its own named logger (`izibuddy.main` , `izibuddy.api.templates` , `izibuddy.core.git_manager` …). To enable `DEBUG` on a specific module without noise elsewhere, edit the code of `src/core/logging/structured_logger.py` — or use the `LOG_LEVEL` environment variable for a global override.

49.3 — Text format vs JSON format

The format is driven by `logging.format` :

```
 # Text mode (development)
2025-01-07 10:30:45 [INFO] izibuddy.main: Server started | port=8765

# JSON mode (production)
{"timestamp":"2025-01-07T10:30:45.123Z","level":"INFO",
 "logger":"izibuddy.main","message":"Server started",
 "correlation_id":"abc-123-def","port":8000}
```

The JSON format is designed to be ingested directly by an external collector (Filebeat, Fluentd, Vector) that pushes them to ELK, Splunk, or Datadog. Each line is a complete JSON object, with the fields `timestamp` (ISO 8601), `level` , `logger` , `message` , `correlation_id` , and log-specific attributes.

The text format is more readable *manually* but harder to parse. Reserve it for development environments and for incidents you diagnose in SSH directly on the machine.

49.4 — Rotation and retention

The `logging.rotation_*` parameters control rotation by size :

- `rotation_enabled = true` — enables rotation.
- `rotation_max_size_mb = 50` — size beyond which a new file is created.
- `rotation_backup_count = 10` — number of archived files kept.

Beyond `backup_count`, the oldest archives are deleted. With the default values (50 MB × 10 files + 1 active), you have about 550 MB of logs at most — typically two to four weeks of history on a medium-sized instance.

For longer retention, do not crank `backup_count` to the extreme. Prefer shipping the logs to an external aggregator (see next section) and let the local files serve as a short-term buffer.

49.5 — The correlation id

Every incoming HTTP request receives, upon arrival, a unique identifier in UUID format. This identifier is placed in the response's `X-Correlation-ID` header and propagated to every log produced during request handling. The `core/logging/correlation.py` middleware handles propagation.

In practice, that means that from a correlation id (seen in the response of a failed request, or found by searching in the JSON log), you can reconstruct the entire server-side sequence of events :

```
grep '"correlation_id":"abc-123-def"' izibuddy.log
```

This discipline is precious when several concurrent requests produce an apparently similar failure — the correlation id lets you separate their sequences unambiguously.

49.6 — HTTP request logging

The `request_logger.py` middleware produces, for each request, a summary line :

```
2025-01-07 10:30:45 [INFO] izibuddy.access:
  POST /api/templates/create 201 145ms cid=abc-123 user=alice
```

Requests longer than `slow_request_threshold_ms` are marked with a `[SLOW]` prefix and may trigger a `slow_requests` alert (Chapter XLVI). This gives you a precious weak signal for spotting bottlenecks before they become visible to users.

Disable request logging (`access_log_enabled = false`) if you already record it at the reverse-proxy level — do not pay for the volume twice.

49.7 — Integration with an external aggregator

The JSON format is the royal road to an aggregator. Three common patterns : **Filebeat → Elasticsearch → Kibana** (JSON parsing, daily indexing, no regex) ; **Fluent Bit → Loki** (Grafana, compression and label-based search) ; **Vector → S3** (long-term archiving at controlled cost, later read via Athena or DuckDB). In all three cases, keep the correlation id intact in the pipeline — it is the main tool to reconstruct incidents *after the fact*.

49.8 — Logging from your code

If you extend IziBuddy with a Python module, use the structured logger rather than `print()` or the standard `logging` :

```
 from core.logging import get_logger

logger = get_logger('my.module')

# Simple message
logger.info("Task completed")

# Message with structured data (attributes go in a data dict)
logger.info_data("Template created", data={"template_id": "abc", "user":

# Error with context
try:
    ...
except ValueError as e:
    logger.error_data("Validation failed", data={"field": "email", "valu
```

The `_data` suffix adds the attributes of the `data` dict to the log as separate JSON fields (useful for Kibana / Grafana queries). Without the suffix, the message is free text.

49.9 — What to log, what not to log

The log must not become a secondary vault. Simple rules : **never** a password, token, API key, or user prompt content ; never a plaintext email in a GDPR environment (a stable hash of the account suffices for correlation) ; **always** the action and its outcome (success/failure, duration, target identifier), the correlation id when inside a request, and the admin's identity for privileged actions. There is **no automatic field masking** in the current build — it is the calling code's responsibility never to pass a secret to the logger in the first place.

Chapter L — Backups

IziBuddy ships with a built-in backup manager that takes timestamped, compressed snapshots. In the current build the single wired trigger is the **pre-import backup**: before a bulk import overwrites templates, the manager snapshots the affected scope so the import can be undone. The manager also exposes the primitives (`create_backup`, `cleanup_old_backups`, `restore_backup`) that an operator can drive, but there is no automatic per-write snapshot today. This chapter covers the internal mechanics, the recommended off-host strategy and the restore procedure.

50.1 — The `BackupManager`

The module `src/core/backup_manager.py` implements a simple but robust class. When it takes a snapshot (the pre-import path, or a manual `create_backup` call) the manager:

1. Verifies that the source file exists.
2. Generates a timestamped name with millisecond precision (`<basename>_YYYYMMDD_HHMMSS_mmm.json.xz`).
3. Compresses the content with LZMA2 (preset 6) — a good trade-off between compression ratio and CPU cost.
4. Writes under `~/.izibuddy/backups/` (the default base directory; the `backups_dir` key of `[paths]` is not honored in the current build).
5. Cleanup of older backups can be triggered according to the retention policy (`cleanup_old_backups`).

The `.xz` compression typically divides the size of prompt JSON files by 5 to 10 — useful when you keep several dozen versions per prompt.

50.2 — Retention policy

Retention is defined by a string of the form `<number><unit>`: `8h`, `10d`, `7m`, `2y` for hours, days, months and years respectively. The `BackupManager` reads this value, converts it into a Python `timedelta`, and on every write deletes backups beyond the threshold.

The month is approximated to 30 days and the year to 365 days — that is enough for practical retention policies. For a finer policy (business days, exact calendar days), combine it with an off-host strategy (see the next section) rather than trying to refine the internal retention.

The default retention is 7 days (`7d`). The `[backups]` section of `izibuddy.ini` also declares `max_count`, `auto_archive`, `archive_after_days` and `archive_after_versions`, intended as a global backup ceiling and a long-term archival tier. Be aware that in the current build these keys are **declarative only** — no `max_count` enforcement and no archival tier are wired (there is no `archive_manager.py`), so plan retention around the off-host strategy below rather than around those settings.

50.3 — The multi-user Git mode

When your base connector is `multi-user-with-git` (the default mode), every change also produces a Git commit in the local repository `~/.izibuddy/templates.repo`. This gives you, in addition to the `.xz` backups, a granular history with commit messages, diff and blame — see Chapter XXVIII for user-side versioning and Chapter XXVII for group Git repositories.

The two mechanisms are complementary: Git captures the semantics (who changed what, when, and why), the `.xz` backups capture the complete state of the file at a point in time, without dependency on Git. If your Git repository is corrupted, the `.xz` backups let you rebuild the state without having to dig into `.git`.

50.4 — Off-host backup

Local backups protect against software corruption and human error, but not against loss of the host (hardware disaster, accidental deletion of the container, ransomware). For that, you need an off-host copy. Three common schemes :

External pull (recommended). An external backup server periodically opens an SSH session on the IziBuddy instance and copies the contents of `~/.izibuddy/backups` (and ideally also `izibuddy.ini`, `templates.repo/`, and the `users/` folder) to long- term storage (S3, NAS, tape). This is the most robust scheme : the IziBuddy server does not know the destination, so a compromise of the instance does not compromise the backups. Daily cron on the backup server, `rsync` or `restic` on the tooling side.

Push from the host to S3. A cron job on the IziBuddy host synchronizes `~/.izibuddy/backups` to an S3 bucket in *write-once* mode (S3 Object Lock or *append-only* IAM policy). Simpler to set up, less robust : if the machine is compromised, the S3 credentials are accessible to the attacker.

Volume snapshot. On cloud platforms (AWS EBS, GCP Persistent Disk), volume snapshots are an efficient and cheap mechanism. Combine with a provider-native retention policy and with cross- region copy for resilience against a regional outage.

The example in Chapter LII shows a `docker run` that produces a `.tar.gz` archive of the Docker volume — that's the basis of the *push from host* scheme.

50.5 — Backing up the configuration

Templates are not the only thing to back up. The complete configuration of an instance comprises :

- `izibuddy.ini` (server configuration, with secrets masked or versioned separately).
- `templates.repo/` (the Git templates repository + history).

- `~/.izibuddy/users/` (account preferences and state).
- `analytics/` (the raw JSONL and the aggregated SQLite database).
- `logs/` (recent logs — not strictly necessary but useful for *post-mortem* investigations).
- The license key (lives in the application configuration, see Chapter XLIV).

A complete backup plan must cover all of these. The simplest scheme : a single Docker volume mounted at `/data` containing all these paths, and a daily backup of the whole volume.

50.6 — Restoring a backup

The `BackupManager` exposes a `restore_backup(backup_path, target_path)` method that decompresses an `.xz` backup and writes it to the target location. It first creates a *safety backup* `.bak` of the current file to protect you against a bad restore.

In practice, you rarely restore a single file — you restore the entire instance. The recommended procedure :

1. **Stop the server** (`docker compose down` or equivalent).
2. **Back up the corrupted state** for later analysis (`tar czf corrupted-state-YYYYMMDD.tgz /data`).
3. **Decompress the off-host backup** into a temporary directory.
4. **Compare the configuration files** between the backup and the current state — verify that no secret has been rotated in the meantime.
5. **Overwrite the volume** with the backup contents.
6. **Restart the server** (`docker compose up -d`).
7. **Check the diagnostics page** (Chapter XLVII) — all components must be green.

50.7 — Restore drill

An untested backup is not a backup. The recommended practice : a quarterly *restore drill* on a test instance, with the real data from the previous day. The drill measures two things :

- **The RTO** (Recovery Time Objective) — how long it takes to bring a functional instance back up from zero. For IziBuddy, count 15 minutes for a standard Docker Compose installation, plus the time to transfer the backup from external storage.
- **The RPO** (Recovery Point Objective) — how much data you lose in the worst case. With daily backups, the RPO is 24 hours ; with hourly backups, 1 hour ; with continuous mirroring (EBS snapshot every 15 minutes), a few minutes.

Adjust the frequency to your tolerance. For most IziBuddy instances, daily off-host backups with a `30d` retention cover the need.

Chapter LI — The REST API

All of IziBuddy can be driven through a REST API. The web interface, the browser extension and any third-party clients all consume this same API — it is the source of truth. This chapter covers the authentication mechanics, the inventory of endpoints by resource, and a few `curl` recipes to get started. The exhaustive reference (object formats, precise error codes) lives in the companion volume *IziBuddy Backend API Reference* (`docs/books/09-backend-fasta-pi/`).

51.1 — Authentication

IziBuddy uses a single authentication mechanism : a JWT bearer token.

Every authenticated request carries an `X-IziBuddy-User: <username>` header for the current identity together with a session token in `Authorization: Bearer <jwt>`. The JWT is a signed HS256 token obtained from `POST /api/auth/login`. Its lifetime is **24 hours** by default, or **7 days** when the login requested « remember me » ; once it expires, the client must re-authenticate. There is no permanent, non-expiring API token — scripts log in the same way and refresh their JWT when it expires.

The public endpoints (landing page, `/api/status`, Lemon Squeezy webhook) do not require authentication. The `/api/admin/*` endpoints additionally require the user to be a system admin ; the write-side `/api/license/*` endpoints do too.

51.2 — The response envelope

All routes return JSON. Endpoint-level responses commonly carry a `success` flag alongside their payload :

```
{ "success": true, "prompt": { ... } }
```

Handled errors at that level return the flag plus a message :

```
{ "success": false, "error": "Template <id> not found" }
```

There is **no** uniform `{success, data}` wrapper and no stable machine-readable `code` field. Framework-level errors raised as FastAPI `HTTPException` instead return the standard `{ "detail": "..." }` body with the appropriate HTTP status : 400 for a malformed request, 401 for missing authentication, 403 for insufficient rights, 404 for an absent resource, 422 for a Pydantic validation error, 500 for a server error. Clients should key off the HTTP status and tolerate both shapes.

51.3 — Inventory — Templates

The `/api/templates/*` router (in `src/api/templates.py`) is the largest one. The main routes (see the endpoint index, appendix A of the *Backend API Reference*, `docs/books/09-backend-fastapi/`) :

- `GET /api/templates/list` — the prompt list (it accepts `includeDeleted` ; tag, marker and text filtering happen client-side, see Chapter XII).
- `GET /api/templates/get/{template_id}` — a complete prompt.
- `POST /api/templates/create` — create.
- `PUT /api/templates/update/{template_id}` — update (partial patch).
- `PUT /api/templates/update/bulk` — update N templates in a single request.
- `DELETE /api/templates/delete/{template_id}` — delete (soft delete to the trash).
- `GET /api/templates/render/{template_id}` — render with substitution of system and POO²L variables.

- `GET /api/templates/trash/list`, `POST /api/templates/restore/{id}`, `DELETE /api/templates/purge/{id}` — trash management.
- `POST /api/templates/share/{id}`, `POST /api/templates/duplicate/{id}` — sharing and duplication.
- `POST /api/templates/import`, `GET /api/templates/export` — import/export.

51.4 — Inventory — Tags, containers

Tags live under `/api/tags/*` (`src/api/tags.py`, ~698 lines): `list`, `create`, then rename or delete a tag through `PUT` / `DELETE /api/tags/{tag}`, plus a `merge-into` operation and the hierarchical endpoints in the v2 router (`POST /api/tags/v2/{source}/merge-into/{target}`, `/api/tags/v2/*`). There is **no** separate categories API — categorization is realized through (hierarchical) tags, so there is no `/api/templates/categories/*` router.

The **containers** (folders and collections) have their own endpoints documented in `src/api/template_structure.py`.

51.5 — Inventory — Groups and users

Groups under `/api/groups/*` (`src/api/groups.py`, ~511 lines): `list`, `create`, `GET/PUT/DELETE /api/groups/{id}`, `GET /api/groups/{id}/members`, `POST/DELETE /api/groups/{id}/members/{username}` (membership) and `POST/DELETE /api/groups/{id}/admins/{username}` (the group-admin toggle — there are only two levels, member and group admin, not a role hierarchy).

Users under `/api/users/*` (`src/api/users.py`): `GET /api/users` (listing), `GET /api/users/me`, `GET/POST /api/users/{username}`, `PUT /api/users/me/web` and `PUT /api/users/me/extension` (language / theme / preset / features preferences), `PUT /api/users/{username}/web` (admin only). **Invitations** under `/api/invitations/*` (Chapter XXVI).

51.6 — Inventory — Admin, analytics, alerts, license

Administration under `/api/admin/*` (`src/api/admin.py`) covers restart (`POST /api/admin/restart`, `restart/status`, `restart/cancel`) and backend migration (`GET/POST /api/admin/migration/config`, `GET /api/admin/migration/storages`). The admin user listing is `GET /api/auth/users` (`src/api/auth.py`, admin only) — there is no `/api/admin/users` route.

Analytics lives under `/api/analytics/*` (Chapter XLV), **alerts** under `/api/admin/alerts/*` (Chapter XLVI), the **license** under `/api/license/*` (Chapter XLIV), **versioning** under `/api/versions/*` (Chapter XXVIII), and **diff** under `/api/diff/*` (`src/api/diff.py`, ~473 lines).

51.7 — Pagination, filtering, search

Filtering and search are **not** uniform across the API. The prompt list, in particular, returns the full collection and is filtered and searched client-side (Chapter XII); pagination is applied only where a specific endpoint implements it, not as a global convention. For bulky data, prefer the export endpoints that produce a downloadable file rather than assembling it from list calls.

51.8 — `curl` recipes

A few examples covering authentication and the classic patterns. All assume `TOKEN` is exported and an instance domain is substituted.

```
 # List templates (filter client-side on the returned collection)
curl -H "X-IziBuddy-User: alice" -H "Authorization: Bearer $TOKEN" \
     "https://izi.example.com/api/templates/list"

# Create a prompt
curl -X POST -H "X-IziBuddy-User: alice" \
```

```
     -H "Authorization: Bearer $TOKEN" -H "Content-Type: application/jso
     -d '{"name":"My prompt","tags":["dev"],"text":"Hello"}' \
     "https://izi.example.com/api/templates/create"

# Export analytics
curl -H "X-IziBuddy-User: admin" -H "Authorization: Bearer $TOKEN" \
     -o analytics.csv \
     "https://izi.example.com/api/analytics/export/csv?start_date=2026-04-01"

# Restart with a countdown
curl -X POST -H "X-IziBuddy-User: admin" \
     -H "Authorization: Bearer $TOKEN" -H "Content-Type: application/jso
     -d '{"countdown":30}' \
     "https://izi.example.com/api/admin/restart"
```

51.9 — Discovering routes

The endpoint `GET /api/routes` (served by `src/api/status.py`) returns the exhaustive list of routes of the running server. This is more reliable than the static documentation for development versions.

A second endpoint `GET /api/routefile` produces an **HTML page** that displays, for each route, the source file and the method. This page is designed for administrators who want to quickly see where a given feature is implemented — handy when reading code.

FastAPI also exposes `/docs` (Swagger UI) and `/redoc` (ReDoc) with interactive schema documentation. These two pages are *public* by default — consider closing them or protecting them on public deployments (see `src/main.py` to switch them to `include_in_schema=False`).

51.10 — API versioning

The API does not follow prefix-based versioning (no `/v1/`). Backwards-incompatible changes are rare and signalled in the release notes (Chapter LIII). *Additive* changes (new endpoint, new optional field in a response) are published without notice — a robust client tolerates unknown fields in the response.

For critical external integrations, follow the release notes at each update and test in pre-production before switching over.

Chapter LII — Deploying behind a reverse proxy

Exposing IziBuddy directly to the Internet is not a good idea. The FastAPI backend does its job correctly but is not designed for TLS termination, rate-limiting or absorbing basic HTTP attacks. The recommended deployment scheme places a reverse proxy between the Internet and IziBuddy : it handles the certificate, applies security headers, and proxies requests to the backend. This chapter covers proxy choices, the headers to forward, WebSocket support, and the hardening checklist to apply before opening to the world.

52.1 — The target architecture

The recommended starting point is the compose stack shipped with the repository (`docker-compose.yml` + `Caddyfile`): Caddy at the front on ports 80/443, IziBuddy on port 8765 in an internal Docker network, TLS certificate managed automatically by Let’s Encrypt. An Nginx variant does exactly the same thing for teams that prefer Nginx.

```
Internet (443/HTTPS + 80/HTTP)
        |
        v
  +---------+        +-----------+
  |  Caddy  |  --->  | IziBuddy  |
  | (TLS)   |        |  :8765    |
  +---------+        +-----------+
```

The backend is not exposed directly. It listens on `0.0.0.0:8765` *inside* the Docker network — invisible from outside the container. That is what keeps the configuration simple : no TLS management in the backend, no special IP-forwarded logic, the proxy does everything.

52.2 — Prerequisites and first start

The compose stack assumes a public Linux server (Ubuntu 22.04+, Debian 12+ or equivalent) with a public IP, Docker Engine 24+ with Compose v2, a DNS `A` (and/or `AAAA`) record pointing at the server, and inbound ports **80** and **443** open at both the cloud firewall and the OS level. Port 80 matters even for an HTTPS-only deployment : Let's Encrypt uses it for the HTTP-01 challenge.

Configuration starts from the template : `cp .env.example .env`, then set the two variables the stack reads :

```
IZIBUDDY_DOMAIN=izibuddy.example.com
IZIBUDDY_TLS_EMAIL=admin@example.com
```

The e-mail is passed to Let's Encrypt for expiry notifications. Certificates are persisted in the `caddy_data` Docker volume, so container rebuilds do not re-trigger issuance. Inside the container the backend listens on port `8765` ; leave `server.port` and `server.host` alone in `izibuddy.ini` unless the `Caddyfile` and `docker-compose.yml` are adjusted in the same move. Then `docker compose up -d --build` — on first start Caddy requests the certificate automatically, observable with `docker compose logs -f caddy`.

52.3 — The minimal `Caddyfile`

```
{$IZIBUDDY_DOMAIN} {
    tls {$IZIBUDDY_TLS_EMAIL}

    encode zstd gzip

    header {
        Strict-Transport-Security "max-age=31536000; includeSubDomain
        X-Content-Type-Options "nosniff"
        X-Frame-Options "SAMEORIGIN"
        Referrer-Policy "strict-origin-when-cross-origin"
```

```
    }

    reverse_proxy izibuddy:8765 {
        header_up X-Forwarded-Proto {scheme}
        header_up X-Forwarded-Host {host}
    }
}
```

Caddy automatically obtains and renews the Let's Encrypt certificate from the variables `$IZIBUDDY_DOMAIN` (the deployment's domain name) and `$IZIBUDDY_TLS_EMAIL` (the contact address for notifications). The `zstd gzip` compression shrinks the volume of JSON and HTML responses, which compress very well.

52.4 — The headers to propagate

IziBuddy expects a few headers: `X-Forwarded-Proto: https` (to generate absolute links in verification and password-reset e-mails), `X-Forwarded-Host` (same), `X-Forwarded-For` (to log the real IP in logs and the connection history), `X-Correlation-ID` (optional — if you originate the correlation id at the proxy in a multi-service architecture). Caddy adds the first two automatically with `reverse_proxy`; Nginx requires explicit `proxy_set_header` directives. Do **not** propagate `Authorization` coming from an upstream auth mechanism — mixing SSO + IziBuddy auth requires a specific integration.

52.5 — WebSocket

IziBuddy does not use *native* WebSocket as standard — the extension communicates via HTTP polling. Some experimental modules (`docs/plans/MULTIPANE-PLAN.md`, *labs*) may introduce a WebSocket connection for real-time panel synchronization. In that case, allow the *upgrade* on the proxy side: for Caddy, propagate `Connection` and `Upgrade` via `header_up`; for Nginx, `proxy_http_version 1.1` plus `proxy_set_header Upgrade $http_upgrade`. Verify with `wscat` or the browser DevTools that the connection establishes.

52.6 — Bootstrapping the first user

On a fresh deployment accessible to the Internet, the first visitor who registers receives the admin flag (Chapter XLII). To prevent a random attacker from grabbing that role, first disable public registration (`registration.self_registration_enabled = false`), start the instance, *temporarily* re-enable registration with an `allowed_domains` restricted to your domain, create your admin account, verify the status on the diagnostics page, then re-set the registration to your policy. A simpler but less safe alternative : start with open registration, create the admin account within the same minute, then close it. The window of attack is narrow but real — prefer the first procedure for public deployments.

52.7 — Verifying the deployment

Open `https://<domain>` — the dashboard must load over HTTPS with a valid padlock. The diagnostic page (`/static/diagnostic.html`) should report `Environment: NATIVE` and an internal Docker-network IP : that combination confirms the request travelled through the proxy and reached the backend on the internal network rather than a host port left exposed.

52.8 — Hardening checklist

Before opening to the Internet, walk through this list :

- **TLS only.** Redirect all HTTP traffic to HTTPS. The Caddy configuration does that by default.
- **HSTS enabled** with `max-age=31536000; includeSubDomains` and ideally `preload` for stable domains.
- **Defensive headers** : `X-Content-Type-Options: nosniff`, `X-Frame-Options: SAMEORIGIN`, `Referrer-Policy: strict-origin-when-cross-origin`.
- **Content-Security-Policy** adapted to your usage (not enabled by default because it depends on user extensions — first test in `Content-Security-Policy-Report-Only` mode).

- **Rate-limiting** on sensitive endpoints : login, register, password-reset. Implementable at the proxy level with a basic `req/s` per-IP rule.
- **Self-registration closed** or restricted to your e-mail domains.
- **Authenticated SMTP** (`email.enabled = true`) so that e-mail verification works.
- **Off-host backups** operational (Chapter L).
- **JSON logs shipped** to an aggregator (Chapter XLIX).
- **Alerts configured** on at least two channels (Chapter XLVI).
- **The `/docs` page** (Swagger UI) closed if you do not want to expose your API schema publicly.
- **Read-only reverse proxy on `izibuddy.ini`** — the IziBuddy container must not be able to rewrite its configuration without human intervention (except for the database migration via the wizard, which already requires an authenticated admin).

52.9 — Backing up the deployment

Three things carry the deployment's state : the `izibuddy_data` volume (templates, configuration, users), the `izibuddy.ini` file, and the `caddy_data` volume — the latter is regenerable, but keeping it avoids hitting Let's Encrypt rate limits after repeated rebuilds. A one-liner snapshots the data volume :

```
docker run --rm -v izibuddy_data:/data -v $(pwd):/backup alpine \
  tar czf /backup/izibuddy-data-$(date +%F).tar.gz -C /data .
```

Chapter L covers application-level backups ; this snapshot is the infrastructure-level complement.

52.10 — External reverse proxy on top of Caddy

If your infrastructure already imposes an upstream proxy/load-balancer (Cloudflare, AWS ALB, host Nginx), remove the `caddy` service from `docker-compose.yml` and expose IziBuddy on a local host port :

```
services:
  izibuddy:
    ports:
      - "127.0.0.1:8765:8765"
```

Your external proxy handles TLS termination and proxies to `127.0.0.1:8765`. Make sure to correctly propagate the headers listed above and to enable compression at the right place (only once — not in Caddy and ALB simultaneously).

52.11 — Troubleshooting

Caddy stuck on « obtaining certificate » — verify that ports 80 and 443 are open from the Internet, that DNS points to the right IP (`dig +short`), and check `docker compose logs caddy`. **502 Bad Gateway** — backend not started or crashed ; check `docker compose logs izibuddy` first, then verify `izibuddy.ini` has `host = 127.0.0.1` **or** `host = 0.0.0.0` — the `IZIBUDDY_HOST` environment variable forces `0.0.0.0` by default inside the container, and a stricter value breaks the internal routing. **Mixed content** (HTTP assets on an HTTPS page) — normally impossible, every asset is served on relative paths ; if it appears anyway, verify that `X-Forwarded-Proto: https` is reaching the backend (the provided `Caddyfile` forwards it). **Let’s Encrypt rate-limit** — limit creation/destruction cycles of the `caddy_data` volume ; as a last resort, point the global `Caddyfile` block at the staging CA while testing : `acme_ca https://acme-staging-v02.api.letsencrypt.org/directory`.

Chapter LIII — Upgrading IziBuddy

An IziBuddy upgrade is never risk-free, even when it is generally simple. This chapter covers the recommended procedure for a production instance : pre-upgrade checklist, the upgrade steps themselves, post-upgrade checks, and the rollback procedure if something goes wrong.

53.1 — The release cadence

IziBuddy follows a continuous release cadence. Minor changes (fixes, non-blocking additions) are published frequently ; major changes (UI overhaul, data model migration, JSON format change) are explicitly flagged in the release notes and require careful reading before upgrading.

The numbering follows *semver* with some liberties : `MAJOR.MINOR.PATCH`. A change in `MAJOR` indicates a break in compatibility (data format or API breaking change) ; a change in `MINOR` adds features without breaking the existing ones ; a change in `PATCH` fixes bugs.

The **release notes** are published in the repository with each Git tag (`git tag` or GitHub releases) and mirrored on the documentation site. They list : new features, fixes, backwards- incompatible changes, and necessary migrations.

53.2 — The pre-upgrade checklist

Before any upgrade : read the release notes between the current version and the target (cumulatively for multi-step jumps) ; back up the instance (Chapter L) and **verify that the backup is readable**, not just present ; announce the maintenance to users if the instance is shared (5 to 10 minutes is enough for most upgrades) ; check the disk resources on the host (1 to 2 GB of headroom for a Docker re-

build) ; verify system-dependency compatibility flagged in the release notes ; test in pre-production for major releases and for any critical instance.

53.3 — Upgrading with Docker Compose

The standard scheme from Chapter LII.

```
 # Fetch the new version
cd /opt/izibuddy
git fetch origin
git checkout v1.2.0  # or the desired version

# Rebuild the image and restart
docker compose up -d --build

# Tail the logs immediately
docker compose logs -f izibuddy
```

The data (`templates.repo/` , user configuration, backups) is persisted in the Docker volume `izibuddy_data` — it survives the rebuild. The configuration (`izibuddy.ini`) lives outside the container and is not replaced by the upgrade ; you keep your customizations.

The restart typically takes 15 to 45 seconds. The landing page remains accessible behind Caddy but returns `502 Bad Gateway` during the window — that is expected.

53.4 — Upgrading a native installation

Without Docker, the procedure requires more manipulation : `sudo systemctl stop izibuddy` , back up `/opt/izibuddy` into a dated folder, `git fetch && git checkout v<version>` , update Python dependencies (`pip install -r requirements.txt --upgrade` in the venv), restart (`sudo systemctl start izibuddy`), tail the logs (`sudo journalctl -u izibuddy -f`). Adapt to your process mana-

ger (systemd, supervisord, PM2). If you have customized some source files — which is not recommended — `git checkout` rewrites them ; keep your modifications in a separate patch.

53.5 — Data migrations

Some versions introduce format migrations. They live in the `src/migrations/` package and are run **manually as command-line tools** — they are **not** applied automatically at startup. The release notes tell you which migration to run and when. Each pass is run directly as a module, e.g. `python -m migrations.grammar_v2_migration --root <store>` or `python -m migrations.engine_migration --root <store>` :

- `grammar_v2_migration.py` — POO²L syntax migrations (updating legacy delimiter and separator forms to the current `@@@` grammar).
- `text_to_content_migration.py` — JSON field renaming (`text` → `content`).
- Other modules depending on the versions.

Most migrations write `*.json.bak` backups and are safe to rerun ; run them with `--dry-run` first where supported, and watch their output to confirm they finish without error.

For long migrations (on large repositories), run them inside the maintenance window with the server stopped. If a run does not fit your window, abort and restore the backup — you will resume with more time.

53.6 — Database connector migrations

Changing the database connector (for example switching from filesystem to PostgreSQL to support more users) is not a standard upgrade — it is a **backend migration**. The wizard on the diagnostics page (Chapter XLVII) handles it : export from the old backend, import into the new one, update of `izibuddy.ini` , restart.

Before switching : back up the complete instance, verify that the new backend is reachable (test from the wizard), estimate the transfer time on a subset, plan a longer maintenance window (30 minutes to several hours depending on volume). Once the switch is done, keep the backup of the old backend for at least one month — a silent issue can take a few weeks to surface.

53.7 — Post-upgrade checks

After the restart, within 10 minutes : open the diagnostics page (Chapter XLVII) and check that all components are green ; verify the version reported by `GET /api/status` ; list the routes via `GET /api/routes` to spot a missing expected route ; test a key user journey (login, create, list, delete) ; review the logs of the first few minutes to spot an unexpected `ERROR` or `CRITICAL` ; verify that the extension still connects to the chatbots ; verify SMTP by an invitation or alert test. If everything is green, you can close the maintenance window.

53.8 — Rollback procedure

If something went wrong — broken UI, unreachable route, migration in error — the rollback procedure :

With Docker Compose, revert to the previous tag (`git checkout v<old>`) and rebuild (`docker compose up -d --build`). If the upgrade has applied an incompatible data migration, the previous image will refuse to start ; in that case, restore the `izibuddy_data` volume from your archive (`docker compose down` , recreate the volume, `tar xzf backup.tgz -C /data` in a throwaway container, restart).

With a native installation, restore the backup folder, redeploy the Python dependencies for the previous version, restart the service.

In all cases, open a ticket with the source version and the target version, the complete diagnostics report (the « Copy Report » button of the diagnostics page), the last 50 lines of logs (ideally with `correlation_id` for the requests in error), and the exact nature of the incident.

53.9 — Upgrading the extension

The browser extension has its own release cycle. For versions distributed via the Chrome Web Store, the upgrade is automatic. For side-loaded versions (development, internal distribution), you need to restart the browser and reload the extension from the dedicated panel.

Check extension/backend compatibility from the release notes : an extension built against an older backend may miss newer endpoints. There is no automatic version handshake or dashboard banner — after a backend upgrade, reload the extension and confirm it still connects and injects on your target chatbots.

Partie 9

Connectors

Chapter LIV — Connectors — overview

This part of the manual describes every point of contact between IziBuddy and the outside world. Everything that is not internal server code — the database that stores everything (user accounts, templates, tags and metadata), the chatbots into which the extension injects templates, the SMTP server that sends invitations, the remote Git that mirrors the file-based prompt store, the outbound alert webhook — goes through a connector. In IziBuddy, a connector is a pluggable module behind a stable interface, chosen by configuration and loaded at startup.

54.1 — What a connector is

On the server side, the central interface is the abstract `Database` class defined in `src/repositories/database.py`. Each concrete implementation — `PostgreSQLDatabase`, `MongoDBDatabase`, `S3Database`, `FileSystemDatabase` — inherits this interface and provides the same list of methods: `get_user()`, `save_template()`, `list_tags()`, etc. A single database backend is active at a time, and it stores the whole state — accounts, templates and metadata. The application code (the FastAPI routers, the *managers* in `src/core/`) only knows `Database`; it never knows which physical platform it is running on. That is what makes connectors interchangeable without touching the business code.

On the extension side, the principle is the same but the target changes. The prompt text is inserted into the page's input with a **generic** insertion path that is the same for every platform. What differs per platform is file **attachment**: the router `content-platform-router.js` dispatches to a per-platform handler (`attachFileToClaude`, `attachFileToChatGPT`, `attachFileToGrok`, `attachFileToGemini`, …) based on the URL of the current tab. There is no `injectTemplate`/`readResponse` reading protocol — the extension writes the prompt in, it does not read the model's answer back.

54.2 — The families

Connectors fall into a few families, which correspond to the following chapters of this part :

- **Databases** (Chapter LV). One backend is active at a time and stores everything. Sixteen backends are shipped : twelve « regular » ones — JSON files (multi-user-with-git, fs-multi-user, fs-single-user), relational DBMSs (PostgreSQL, MySQL/MariaDB, Oracle, DB2, CockroachDB, Access, SQLite), document-store (MongoDB), search engine (Elasticsearch) — plus four cloud-storage backends (AWS S3, Google Cloud Storage, Azure Blob, MinIO) where the whole database lives in a bucket.
- **Attachment payload store** (Chapter LVI). Not a family of external connectors — a single small key selects how attachment *bytes* are stored : `base64` (inline, the default), `sqlite`, or `filesystem`. There is no separate object-storage layer for prompt content.
- **AI platforms** (Chapter LVII). The chatbot targets of the extension (Claude, ChatGPT, Grok, Gemini, and other providers), each with its own DOM specifics for the attachment step.
- **External services** (Chapters LVIII to LX). SMTP for outgoing mail, Git for mirroring the file-based prompt store, and a single outbound alert webhook.

54.3 — Choosing a configuration

You pick **one** database backend via `[database] active = <name>` ; it holds the entire state. The attachment payload store, SMTP and the alert webhook are configured independently on top. Common choices :

- **Multi-user with Git** (the default) : everything lives under `~/.izibuddy/` as JSON, versioned in a local Git repository that can mirror to a remote. Zero external database dependency ; suited to teams, demos and small-to-medium installations.

- **PostgreSQL** (or another relational DBMS) : typical production configuration when concurrency and a proven, backed-up DBMS matter.
- **MongoDB** : self-hosted document-store alternative for organizations that prefer a NoSQL store.
- **Elasticsearch** : relevant when full-text indexing over thousands of templates is the dominant concern.
- **S3 / GCS / Azure / MinIO** : the whole database persists in an object-storage bucket — a cloud-native option that needs no separate DB server.

SMTP and the alert webhook are independent of the database choice — enable or disable each on its own. Git mirroring only applies to the file-based backends.

A subtlety on Git : when `[database]` selects anything other than a file-based backend, the Git connector does not apply — the database itself is the source of truth, and IziBuddy hides the Git UI accordingly (the Git endpoints return 409). Chapter XXVII §when git is applicable covers the choice.

54.4 — Anatomy of a connector configuration

The database backend is selected by an `active` key under `[database]`, whose value names a `[database.<name>]` subsection that carries the backend-specific keys (`host`, `port`, `database`, `user`, `password`, or `path`, or `bucket`, `region`, `access_key`, `secret_key`). The file is parsed with plain `configparser` : there is **no `${VAR}` substitution** — values are taken verbatim (the one exception is the Lemon Squeezy licence file, Chapter LXI). Keys read secrets straight from the file, so keep them out of version control.

```
 [database]
active = postgresql

[database.postgresql]
host = db.local
```

```
port = 5432
database = izibuddy
user = izibuddy
password = <secret>
```

There is no separate `[storage]` object-storage section. Attachment bytes are handled by `[attachments]` (`storage_backend = base64 | sqlite | filesystem`), not by an external connector.

Loading is done at startup by `load_database_from_config()` in `src/repositories/database_config.py`. If the configuration is missing or a backend fails to load, IziBuddy **does not refuse to start** — it logs the problem and falls back silently to the default `FileSystemDatabase` (`src/repositories/__init__.py`). So after a config change, check the startup logs and the actual storage reported on the diagnostics page rather than assuming a bad connector would have blocked the boot.

54.5 — Testing a connector after installation

There is **no *Test connection* button** and the diagnostics page does **not** run live network probes (no `SELECT 1`, `head_bucket` or `EHLO`) — Chapter XLVII reports the configured storage type, not a live reachability check. Validate a connector by performing **real application operations**: log the configured storage type from the startup logs, then create an account, write a prompt, and send an invitation. If all three succeed, the database + attachments + SMTP combination is working end to end. The backend migration wizard (Chapter XLVII) is the supported way to move data between backends.

54.6 — Map of the chapters

The rest of Part IX details each connector in the following order: Chapter LV covers the database backends, grouped by use case; Chapter LVI follows with the attachment payload store; Chapter

LVII describes the AI-platform targets on the extension side ; Chapter LVIII details the SMTP configuration ; Chapter LIX lays out the single-instance remote Git mirror for the file-based prompt store ; Chapter LX walks through the outbound alert webhook. Each chapter follows the same outline : purpose, configuration keys, prerequisites, operational considerations, and troubleshooting.

Chapter LV — Database connectors

IziBuddy offers twelve database connectors — the file family, the SQL family and two NoSQL backends — plus four cloud-storage backends (AWS S3, GCS, Azure Blob, MinIO) covered in Chapter LVI, plus optional cache and replication wrappers. All of them conform to the abstract `Database` interface from `src/repositories/database.py` : one application code base, many physical implementations. **One backend is active at a time.** The choice is made in `izibuddy.ini` via `active = <name>` under `[database]`, which names a `[database.<name>]` subsection carrying the backend's keys ; it is resolved at startup by `load_database_from_config()` (`database_config.py`). This chapter walks through the backends grouped by use case rather than one by one : the decision depends primarily on the infrastructure you have and the scale you target, not on the alphabetical order of names.

55.1 — Selection matrix

Four use families, plus the wrapper tooling :

- **Getting started and single-user** : the minimum required for an isolated workstation or a demo. No binary dependency. Backends : Filesystem (default), Pure Filesystem, Single User.
- **Small team server** : a single host, fewer than fifty users. Backends : SQLite, MySQL, MariaDB.
- **Standard production** : a dedicated server, centralized backups, anticipated growth. Backends : PostgreSQL, enterprise MySQL, MongoDB.
- **Large-scale production** : high availability, geographic distribution, search-dominant. Backends : CockroachDB, Elasticsearch, plus the Replicated and Cached wrappers.
- **Mandated enterprise standards** : the hosting infrastructure dictates the choice. Backends : Oracle, DB2, Access.

Two optional wrappers apply *on top of* an existing backend : `CachedDatabase` adds a Redis cache and `ReplicatedDatabase` adds a master/slaves dispatcher. A separate migration utility (`database_migrator.py`) moves data between backends.

55.2 — Getting started : the file family

Filesystem (`filesystem_database.py`). The default backend. Stores everything under `~/.izibuddy/` in pure JSON, with no index. It is the initial installation choice and the demo choice. No dependency, no configuration, instant startup. Limits : no serious concurrency beyond a handful of users, no index, which makes search linear. Suited to development, discovery instances and labs.

Pure Filesystem (`pure_filesystem_database.py` , the `fs-multi-user` backend). A multi-user JSON store like Filesystem but **without the Git versioning layer** — plain JSON files, no commit history. Useful when you want the simplicity of file storage without a repository. Trade-off : no per-change Git history and, like Filesystem, linear search beyond a few hundred templates.

Single User (`single_user_database.py` , the `fs-single-user` backend). A mono-user store under `~/.izibuddy/` . No notion of multiple users — authentication is effectively single-identity. It is selected explicitly by setting `active = fs-single-user` in `[database]` ; there is no automatic environment detection.

55.3 — Small team server

SQLite (`sqlite_database.py`). A single `.db` file on disk, standard SQL behind the `Database` interface. Inherits from `SQLDatabase` (`sql_database.py`) which carries the generic schema. Typically suits fewer than fifty users on a single host and remains the zero-configuration option beyond Filesystem. No third- party driver to install — `sqlite3` ships with Python. Backups are done by simply copying the file (cold or via SQLite’s `.backup`).

MySQL / MariaDB (`mysql_database.py`). Same role as PostgreSQL but on the MySQL ecosystem. Dependency : `mysql-connector-python` or `PyMySQL`. The configuration resembles PostgreSQL aside from the port (3306) and the connection string (`mysql://user:pass@host:3306/db`). Note : MariaDB is used without modification to the connector — the protocol remains compatible.

55.4 — Standard production

PostgreSQL (`postgresql_database.py`). This is the recommended choice for any installation beyond a trial team. Also inherits from `SQLDatabase` and uses the `SQL_SCHEMA_POSTGRESQL` schema. Binary dependency : `psycopg2`. Compatible with PostgreSQL 10 and above. Benefits without additional configuration from the engine's usual capabilities : ACID transactions, online backups, *logical* or *streaming* replication at the DBMS level itself, full-text search extensions if you need them before switching to Elasticsearch.

MongoDB (`mongodb_database.py`). NoSQL document-store, compatible with MongoDB 4.4 and above. Dependency : `pymongo`. Relevant when the prompt volume is very large, when the schema changes often, or when the organization already has MongoDB in service. The connector stores users, groups, templates, tags and important tags in distinct collections, as one would expect from a document-store. For advanced analytics, MongoDB is often combined with external search (Elasticsearch) rather than pushing on the aggregation engine.

55.5 — Large-scale production

CockroachDB (`cockroachdb_database.py`). Wire-protocol compatible with PostgreSQL — so we reuse `psycopg2` — but the underlying engine is distributed and strongly consistent at planetary scale. To consider when you need geo-distributed writes or regional tolerance. The trade-off is operational cost : CockroachDB is not a DBMS to administer like PostgreSQL.

Elasticsearch (`elasticsearch_database.py`). To use when full-text search over a library of several thousand or tens of thousands of templates is the dominant load. Dependency : `elasticsearch>=8.0.0`. The connector indexes name, description, content, tags and allows fast aggregations. Often deployed *in addition to* a primary DBMS (PostgreSQL, MongoDB) rather than as the sole store. For use as the sole store, size carefully : no transactions, no referential integrity constraints.

55.6 — Mandated enterprise standards

Oracle (`oracle_database.py`). For environments where Oracle is mandated by IT. Dependency : `oracledb` (formerly `cx_Oracle`). Inherits the same `SQLDatabase` as the other SQL backends, with a schema adapted to the Oracle dialect.

IBM DB2 (`db2_database.py`). Same role as Oracle, in the IBM ecosystem. Dependency : `ibm-db`. Mostly present in banks, insurance companies and government.

Microsoft Access (`access_database.py`). Compatibility backend for legacy small Windows deployments. ODBC driver required. To use only when you absolutely must coexist with an existing Access database — otherwise SQLite fills the same role with fewer constraints.

A note on `SQLDatabase` (`sql_database.py`). This is the **abstract** common parent of all relational connectors — it carries the shared schema and query logic but declares abstract methods, so it is **not selectable** as a backend on its own. There is no generic ODBC/JDBC catch-all connector : to support a DBMS you select one of the concrete backends above.

55.7 — The three wrappers

Replicated (`replicated_database.py`). Master/slaves wrapper : all writes go to the master, reads are *round-robin*'d across slaves with automatic *failover* when a slave becomes unavailable. Periodic health checks. Configured in `[database.replication]` :

```
 [database.replication]
enabled = true
slaves = postgresql://izi:<secret>@db-r1.local/izi,
         postgresql://izi:<secret>@db-r2.local/izi
health_check_interval = 30
read_from_master = false
```

(Values are literal — there is no `${VAR}` substitution in `izibuddy.ini` ; write the real credentials in the deployed file and keep it out of version control.)

Cached (`cached_database.py`). Redis cache in front of any backend. Reads Redis-first, writes backend-first then Redis invalidation. Configurable TTL per data type. Dependency : `redis`. To enable as soon as you observe repetitive hot reads — for example on the landing pages of heavily consulted groups.

Single User. Already described above ; it is technically also a wrapper in that it *replaces* the full interface with a simplified, concurrency-free implementation.

55.8 — Migrating between backends

`database_migrator.py` exports the contents of a backend as compressed JSON and re-injects them into another. The standard procedure is to stop the instance, export from the old backend, switch the configuration, import into the new one, restart. Migrations *upward* (Filesystem → SQLite → PostgreSQL → …) hold no surprises. Migrations *downward* (PostgreSQL → SQLite) are possible as long as the volume fits ; you then need to think about reducing DBMS-specific indexes from the source.

Special case : switching from a SQL backend to a document-store (PostgreSQL → MongoDB) requires an implicit re-modeling that the migrator handles by serializing joins as embedded documents. Check the migration report printed at the end of the operation to make sure no object was lost.

55.9 — Testing the connector

There is no *Test connection* button and the diagnostics page does not run a live `SELECT 1` or its NoSQL equivalent (reminder of Chapter LIV). Validate the backend by exercising it : after restart, confirm the configured storage type in the startup logs, then write a prompt, share it, delete it and restore it. If the complete chain passes, the backend is working. Remember that a misconfigured backend does not block startup — IziBuddy falls back silently to the default `FileSystemDatabase`, so always verify the active storage rather than assuming the boot would have failed.

Chapter LVI — Object storage : cloud backends and attachments

It is worth stating plainly what IziBuddy does **not** have : there is no separate object-storage layer that splits prompt *content* away from prompt *metadata*, no `[storage]` section in `izibuddy.ini`, and no generic « content store » behind the database. A single database backend holds the entire state. Object storage enters the picture in just two, unrelated ways : a database backend can *itself* live in a cloud bucket, and attachment bytes have their own small payload store. This chapter covers both.

56.1 — Cloud-storage database backends

Four of the shipped database backends persist the **whole database** in an object-storage bucket rather than on a local disk or in a DBMS : AWS S3 (`s3_storage.py` , `S3Database`), Azure Blob (`azure_storage.py` , `AzureDatabase`), Google Cloud Storage (`gcs_storage.py` , `GCSDatabase`) and MinIO (`minio_storage.py` , `MinIODatabase`). They are database backends like any other — you select one with `active` under `[database]` , and it stores accounts, templates, tags and metadata. There is no separate metadata DBMS alongside them.

Because they are database backends, they are configured under a `[database.<name>]` subsection, not a `[storage]` section :

```
[database]
active = s3

[database.s3]
bucket = izibuddy-prod
region = eu-west-1
```

```
access_key = <secret>
secret_key = <secret>
```

The other three follow the same shape :

```
[database.gcs]
bucket = izibuddy
project = my-gcp-project

[database.azure]
container = izibuddy
connection_string = <secret>

[database.minio]
endpoint = minio.intra.local:9000
bucket = izibuddy
access_key = <secret>
secret_key = <secret>
```

Values are literal — `izibuddy.ini` is parsed with plain `configparser` and performs **no `${VAR}` substitution**. Write the real credentials in the deployed file (kept out of version control), or omit them and let the cloud SDK pick up ambient credentials (an IAM role, `GOOGLE_APPLICATION_CREDENTIALS`, an Azure managed identity) as those SDKs do by default.

MinIO is the self-hosted, S3-compatible option : run it on your own infrastructure and point `endpoint` at it. For MinIO over a self-signed certificate, add the certificate to the system truststore rather than disabling verification.

56.2 — The attachment payload store

Independently of the database, the **bytes** of a prompt's attachments are handled by a small payload store selected by a single key :

```
[attachments]
storage_backend = base64
```

Three backends are available (`src/core/attachment_store.py`):

- **`base64`** (the default). Attachment bytes are stored inline, base64-encoded, inside the prompt record. Zero external moving parts; fine for small attachments and keeps everything in one place with the prompt.
- **`sqlite`**. Bytes live in a dedicated SQLite database at `~/.izibuddy/attachments.db`, content-addressed by SHA-256 with reference counting — identical files are stored once and shared. A good choice when attachments repeat across templates.
- **`filesystem`**. Bytes are written as individual files under `~/.izibuddy/attachments/`, also content-addressed. Useful when you want attachment blobs kept out of the prompt records and visible on disk.

There is no S3/Azure/GCS attachment backend and no per-provider object-storage connector for attachments — the choice is exactly these three.

56.3 — Choosing between them

For a cloud-native deployment with no DB server to run, a cloud-storage database backend (S3, GCS, Azure, MinIO) keeps the whole instance in a bucket. For everything else, the default `multi-user-with-git` or a relational backend is simpler, and the attachment store stays on `base64` unless you have many or large attachments — in which case `sqlite` (with its de-duplication) or `filesystem` avoids bloating the prompt records.

56.4 — Durability and cost

The durability of a cloud-storage database backend is that of the bucket — eleven nines on S3/GCS/Azure Standard classes, or whatever your MinIO topology provides. Cost is billed per stored vo-

lume plus per request; since IziBuddy usage is read-dominated, request counts often matter as much as storage. Provider-side lifecycle rules (version expiry, cold-class transitions, Object Lock) are configured on the bucket, not in IziBuddy — leave enough delay before a cold-class transition so that occasional reads are not penalized.

56.5 — Verifying the configuration

There is no *Test connection* button and the diagnostics page does not run a live `head_bucket` or equivalent probe. Validate the setup by exercising it: after restart, confirm the configured storage type in the startup logs, then create a prompt with an attachment, and verify the object appears in the bucket (for a cloud-storage backend) or that the attachment store file exists (for `sqlite`/`filesystem`). Remember that a misconfigured backend does not block startup — IziBuddy falls back silently to the default `FileSystemDatabase`, so always check the active storage rather than assuming the boot would have failed.

Chapter LVII — AI-platform connectors

The IziBuddy browser extension injects templates into five conversational AI chatbots : Claude, ChatGPT, Grok, Gemini and DeepSeek. (There are six provider *configs* under `src/extension/config/ai_providers/` — the five web chatbots plus a `claude-code` entry — but only the five web platforms are targeted by content scripts.) Each platform exposes a different DOM and different behaviours. This chapter reviews how the extension handles them and the skeleton to follow when adding one.

57.1 — The connector model

The extension does **two** things on a chatbot page, and they are not symmetric :

- **Template insertion is generic.** The rendered prompt text is written into the page's editor with a shared insertion path that is the same across platforms — the extension does not maintain a per-platform `injectTemplate` function.
- **File attachment is per-platform.** When a prompt carries an attachment, the platform router (`content-platform-router.js`) dispatches to a per-platform handler (`attachFileToClaude`, `attachFileToChatGPT`, `attachFileToGrok`, `attachFileToGemini`, …) based on the URL of the current tab, because the upload DOM differs on each site. These handlers try several methods in cascade (hidden file input, paperclip button, drag-and-drop) since the target DOM moves frequently.

There is **no `readResponse` / response-reading protocol** : the extension writes the prompt in and, optionally, attaches files — it does not read the model's streamed answer back. (The extension's Server-Sent-Events client talks to the IziBuddy *backend*, not to the chatbot — see the dedicated section below.)

57.2 — Claude (Anthropic)

`content-platform-claude.js`. The connector with the most platform-specific code surface. Target : `claude.ai`. Specifics handled :

- **Central editor**. The prompt text is inserted into the `contenteditable` editor of the central panel.
- **Attachments via three paths**. `attachFileToClaude()` tries successively : direct file input, button `aria-label*="attach"`, drag-and-drop on the `contenteditable`. The first one to succeed wins.

The extension does not read Claude's response, count its tokens, or inspect its reasoning blocks — it inserts the prompt and, if present, attaches files. Quirk to watch : Anthropic regularly changes its DOM structure ; the attachment handler is protected by fallback paths but the extension must be updated on major redesigns.

57.3 — ChatGPT (OpenAI)

`content-platform-others.js` (section `attachFileToChatGPT`). Target : `chatgpt.com`. Specifics :

- **Lexical editor**. ChatGPT uses Meta's Lexical editor. Inserting text requires an `execCommand('insertText')` to keep Lexical's state coherent ; a plain `textContent = …` is invisible to the engine.
- **Attachments**. Method 1 targets the hidden file input ; method 2 clicks the paperclip button and waits for an input to appear ; method 3 uses drag-and-drop.

Quirk : ChatGPT sessions expire on the browser side ; when IziBuddy injects a prompt on an expired session, OpenAI redirects to the login page and the injection is lost.

57.4 — Grok (xAI)

`content-platform-others.js` (Grok section). Target : `x.com/i/grok` and `grok.com`. The most discreet connector in terms of code size because Grok inherits the X React DOM. Specifics :

- **Input zone on Twitter/X**. The editor is a `contenteditable` but protected by the Twitter framework ; injection goes through a synthetic *paste event* rather than direct content modification, otherwise Twitter overwrites the value.
- **Attachments**. No upload protocol separate from Twitter's — IziBuddy uses the same path (drag-and-drop on the editor).

Quirk : Grok does not expose a *developer* mode to distinguish its versions ; the connector treats the variants (Grok 2, Grok 3, Grok on the web vs on X) with the same DOM-detection logic.

57.5 — Gemini (Google)

`content-platform-others.js` (Gemini section). Target : **`aistudio.google.com`** (Google AI Studio — this is the injected host ; `gemini.google.com` is *not* targeted). Specifics :

- **Attachments**. AI Studio accepts images, text documents and some binaries ; the connector uses the classic paperclip button.

Quirk : Google A/B-tests frequently ; two users on the same domain can see two different interfaces. The connector carries fallback paths on the editor selector.

57.6 — DeepSeek

`content-platform-others.js` (DeepSeek section). Target : `chat.deepseek.com`. The simplest of the five, both in DOM surface and in supported features. Specifics :

- **Plain editor**. A classic `textarea` ; insertion is a `value = text` followed by an `input` event. No fallback required most of the time.

- **Limited attachments**. DeepSeek accepts images and simple PDFs ; the paperclip button works through the classic `attachFileToInput()` .

Quirk : DeepSeek uses short sessions ; refreshing the page before each work session avoids surprises.

57.7 — The Server-Sent Events channel

`content-sse.js` . This is the source of a common misconception, so be precise : the SSE client connects to the **IziBuddy backend** (`GET /api/events/stream`), **not** to the chatbot. It is how the extension receives server-side notifications — alerts, events, server-availability signals — pushed by the instance. It has nothing to do with reading a model's response. The reconnection logic is conservative (bounded attempts, a minimum delay between two attempts) so an unreachable backend does not produce reconnection spam.

57.8 — Themes per platform

Theme handling is driven by the provider YAML configs under `src/extension/config/ai_providers/` : each platform declares a `theme.token_map` describing how to read the host page's theme, rather than IziBuddy hard-coding a single attribute like `data-chat-theme` . Because the mapping lives in configuration, a platform's theme wiring can be updated without touching code.

57.9 — Adding a new platform

The standard procedure :

1. Add a provider config `src/extension/config/ai_providers/<name>.yaml` (target hosts, `theme.token_map` , attachment settings) and register it in `_index.json` .
2. Add the attachment handler and, if needed, DOM specifics in the relevant `content/content-platform-*.js` .

3. Declare the content script host in `src/extension/manifest.json` (`matches`) with the right load order.
4. Add URL detection in `content-platform-router.js` so the panel routes attachment handling to the new platform.

No server-side modification is necessary — platform handling lives entirely in the extension.

Chapter LVIII — E-mail / SMTP connector

The IziBuddy e-mail connector opens a single door to the outside world : an SMTP server, through which the product's transactional messages pass. No external queue, no broker, no third-party API — just SMTP. This chapter describes the connector's role, the configuration keys, the degraded mode when SMTP is not configured, and the operational considerations to keep in mind before putting the instance into production.

58.1 — The connector's role

Three families of messages go through SMTP :

- **Address verification** when creating a user account. A time- limited link is sent to the provided address ; until the user clicks it, the account is pending.
- **Password reset**. Same scheme, time-limited link, single-use token on the server side.
- **Group invitations**. When a group administrator invites a user, the invitation goes out by e-mail ; it contains an acceptance link and expires after the configured delay (Chapter XXVI).

A few audit notifications (Chapter XLVI) can also go through e-mail when the operator wishes — the same SMTP configuration is reused.

58.2 — Degraded mode : no SMTP configured

When `enabled = false` (or the `[email]` section is absent), the `EmailService` (`src/core/email_service.py`) falls back to *log- only* mode. Links that would have been sent by mail are written to the application log and printed to the server's standard output :

```
============================================================
EMAIL SERVICE DISABLED - EMAIL VERIFICATION
To: alice@example.com
Link: https://izi.example.com/#/verify-email?token=abc123…
============================================================
```

This mode is useful in development, in integration testing and on the first installation : the operator can complete the registration journey without needing a working MTA. It is **strongly discouraged** in production : no SMTP provider means no invitations sent, and an administrator who has to fish for a link in the logs is not doing team management at scale.

58.3 — Configuration

Section `[email]` in `izibuddy.ini` :

```
[email]
enabled = true
smtp_host = smtp.example.com
smtp_port = 587
smtp_user = izibuddy@example.com
smtp_password = <secret>
smtp_use_tls = true
smtp_from_email = no-reply@example.com
smtp_from_name = IziBuddy
```

The keys in detail :

- **`enabled`** . `true` to enable real sending ; `false` to stay in log-only mode.
- **`smtp_host`** , **`smtp_port`** . The MTA address. Port 25 is rarely usable from a modern application server (blocked by hosters and cloud providers). Prefer 587 (STARTTLS submission).
- **`smtp_user`** , **`smtp_password`** . SMTP authentication. Keep the password out of version control — `izibuddy.ini` is parsed ver-

batim (there is no `${VAR}` substitution), so redact it in any committed copy and write the real value only in the deployed file.

- **`smtp_use_tls`**. Enables STARTTLS on port 587. This is the recommended path for most providers (Postfix, Sendgrid, Mailgun, Amazon SES, OVH, Gandi). There is no separate `use_ssl` / implicit-SMTPS key.
- **`smtp_from_email`**, **`smtp_from_name`**. The identity displayed in the *From* field of messages. The address must be authorized by the MTA (SPF, DKIM, DMARC) ; otherwise messages go to spam or are refused.

58.4 — TLS, authentication, sender identity

Three related but distinct topics :

- **Transport TLS**. STARTTLS on 587 (`smtp_use_tls = true`). Verifies the SMTP server identity via its certificate. No reason to do without it in 2026 — most MTAs refuse an unencrypted connection coming from the Internet.
- **SMTP authentication**. AUTH PLAIN or AUTH LOGIN depending on what the provider accepts. `EmailService` negotiates automatically. Some providers (Amazon SES, Sendgrid) require *SMTP credentials* distinct from the console credentials — generate them on the provider side before filling in `smtp_user` / `smtp_password` .
- **Sender identity**. The SPF / DKIM / DMARC triptych. SPF authorizes the MTA's IP to emit for your domain ; DKIM cryptographically signs messages ; DMARC publishes the policy in case of failure. These settings are done on your domain's DNS zone, not in IziBuddy. To check with a tool like `mail-tester.com` before opening to users.

58.5 — Testing the connection

There is **no *Send a test* button and no *Test connection* button** for SMTP. Validate the configuration by exercising a real flow : create an account with a real address and verify that the verifica-

tion link arrives. This is the ultimate test, as it goes through the whole journey, including template rendering. Watch the application log at the same time — a send marked *sent* there tells you the failure (if any) is downstream (deliverability), not in IziBuddy.

58.6 — Operational considerations

Volume. An instance with a few dozen users typically sends a few messages per day (invitations, occasional resets). A multi- tenant instance with hundreds of users can reach several hundred messages per day on campaign days. Choose an SMTP provider accordingly.

Bounces and permanent errors. IziBuddy does not process bounces. The 5xx codes returned by the recipient MTA come back as a *bounce report* in the `from_address` mailbox ; it is up to the operator to configure a monitoring mailbox and, possibly, to deactivate accounts whose address bounces consistently.

Queue. No queue on the IziBuddy side. If the MTA is unavailable at the moment of sending, the operation is marked in error, the user sees a message in the interface, and they can retry (click on *Resend invitation*, *Forgot password*). For a *fire-and-forget* operation with automa-
tic retry, place a local MTA in front — Postfix configured as a *smart host* with a *deferred queue*.

Delays. Commercial MTAs sometimes inflict *greylisting* of a few minutes on the first message issued from a new IP. The first mail sent after installation can therefore be delayed ; warn the first users if you are on this configuration.

Confidentiality. The links sent contain single-use tokens with a limited lifetime — by default **168 hours (7 days) for address verification** and **24 hours for password reset**, both configurable in `[registration]` (`verification_link_expiry_hours` , `password_reset_expiry_hours`). Even intercepted after expiry, they are unusable. This does not dispense with using TLS transport.

58.7 — Troubleshooting

No mail arrives. First, check the IziBuddy server log : if the mail is marked *sent*, the error is further along. Then check bounces in the sender mailbox. If nothing, the problem is probably SPF/DKIM/DMARC — your `from_address` is not signed correctly and the recipient moved it to spam or rejected it silently.

Authentication error. Check that `smtp_user` and `smtp_password` are the provider's SMTP credentials (often distinct from the web interface credentials). Verify that `smtp_use_tls` is consistent with the port (587 → STARTTLS).

Timeout. If the connection times out, verify that the port is not filtered by the host's outbound firewall. Cloud hosters frequently block 25 ; 587 is usually open.

Chapter LIX — Git remote connector

The IziBuddy Git connector treats the instance's single file-based prompt store — `~/.izibuddy/templates.repo` — as one Git repository, and can mirror it to a remote repository (GitHub, GitLab, Bitbucket, self-hosted Git server). There is one repository per instance, not one per user or per group. This mirroring is not a disguised emergency backup : it is a first-class version-control mechanism that records each change, allows diffing between two revisions and gives technical teams a familiar synchronization point — a SYNCHRONISED REMOTE. This chapter describes the connector's model, the authentication options, the synchronization flow and the recovery procedures.

> ***When this chapter applies.*** *The Git connector is only active on installs that use the file-based storage backend — the default when no `[database]` section is set in `izibuddy.ini`. With a real database configured, templates live in the database itself ; the Git layer is hidden from the UI and the `/api/git/*` endpoints return HTTP 409. See Chapter XXVII §multi-backend topologies for which mode fits which deployment.*

59.1 — The mirror as connector

On the server side, the `GitManager` module (`src/core/git_manager.py`) manages the **single** instance repository `~/.izibuddy/templates.repo`. Its remote and branch are configured in `izibuddy.ini`, under the `[database.multi-user-with-git]` subsection :

```
 [database.multi-user-with-git]
path = ~/.izibuddy/templates.repo
```

```
git_remote = https://github.com/acme/izibuddy-templates
git_branch = main
```

There is no per-user or per-group repository and no per-user opt-in ; the whole file-based store is one repository.

Each creation, modification or deletion of a prompt triggers an *auto-commit* with a standardized message (`update prompt foo-bar` , `create prompt baz` , etc.). When `auto_push` is on (the default) and a remote is configured, the commit is pushed immediately. Bulk operations use a *deferred* mode (`_deferred_files` field of `GitManager`) that accumulates modified files and groups them into a single commit at the end of the operation — avoiding N pushes for N modifications in the same *bulk update*.

59.2 — Authentication models

`~/.izibuddy/templates.repo` is an ordinary Git working tree, so the instance authenticates to the remote through the **standard Git credential mechanisms** of the account the server runs as — there is no IziBuddy-specific key scheme and no `~/.ssh/id_izibuddy_<scope>` per-scope keys. Two common approaches :

SSH. Configure an SSH key for the server account and register its public part as a *deploy key* (with write rights) on the remote repository, then set `git_remote` to the `git@…` SSH URL. A deploy key scoped to the single templates repository is the most restrictive and safest option.

HTTPS token. A *personal access token* (GitHub PAT, GitLab PAT, Bitbucket app password) can be embedded in the HTTPS remote URL :

```
git_remote = https://oauth2:<token>@github.com/acme/izibuddy-templates.git
```

The value is used verbatim — `izibuddy.ini` performs **no `${VAR}` substitution**, so the token sits in the file (redact it in any committed copy) or is supplied through a Git credential helper. Rotate the token regularly ; a compromised token grants access to everything its scope authorizes.

59.3 — The synchronization flow

The write path is straightforward *auto-commit then auto-push* :

1. The application operation runs (create, modify, delete a prompt) and produces a commit with a standardized message.
2. If `auto_push` is on (the default), `git push` sends it to the configured `git_branch` immediately.

There is **no** pull-rebase-before-push and **no** automatic retry-once loop on the write path. Staying aligned with the remote is handled separately by the pull hooks (`sync_on_startup` , `sync_on_read`) and the PULL STRATEGY (`pull_strategy`) setting, described in the *Multi-backend behaviour* section below. The design choice is to *never* do an automatic `push --force` : on a divergence the pull strategy applies (`ff-only` refuses to merge), the warning is logged, and the operator resolves it explicitly.

The target branch is `git_branch` (default `main`). You can choose `master` for legacy repositories, or a branch per environment when mirroring several IziBuddy instances to the same central repository.

59.4 — Conflict surface and resolution

Git conflicts on templates are rare because a prompt lives in a dedicated file and two users do not generally modify the same file in parallel. The cases when they happen :

- **Concurrent editing of the same prompt** by two users on two instances mirroring to the same repository.
- **Manual remote modification** by a developer editing the Git repository outside IziBuddy.

- **History rebase** on the remote side (to be formally avoided).

When a divergence occurs, IziBuddy does not auto-resolve : the warning is logged and the local view is kept. Resolution is a manual Git operation by the operator on `~/.izibuddy/templates.repo`. The recommended mode remains editing from a single side at a time and using IziBuddy as the source of truth, with `pull_strategy=ff-only` so an unexpected divergence surfaces rather than being silently merged.

59.5 — Recovering a broken repository

When the local repository is in an unexpected state (interrupted commit, saturated disk, unfortunate manual intervention) :

1. Stop IziBuddy.
2. Back up `~/.izibuddy/templates.repo` elsewhere (the files are the templates themselves, do not lose them).
3. Delete the `.git/` subfolder.
4. Reinitialize the repository : `git init`, `git remote add origin <url>`, `git fetch origin`, `git reset --hard origin/main`.
5. Compare the recovered files with the backup from step 2 ; carry over manually any divergences.
6. Restart IziBuddy.

The normal mode remains *not* to touch the repository outside IziBuddy. This procedure is an emergency exit, not a workflow.

59.6 — History retention and repository size

Git keeps all history. For a library of a few hundred templates updated daily over years, the repository remains small (typically < 100 MB) because templates are text. Two situations that degrade this profile :

- **Binary attachments in the repository**. When you attach a 5 MB PDF to a prompt and modify it twenty times, the repository grows. For bulky attachments, prefer a non-inline attachment

store (`sqlite` or `filesystem`, Chapter LVI) so the bytes live outside the prompt records rather than base64-inline in the versioned JSON.

- **Very frequent versions**. IziBuddy can be configured to version on every save (Chapter XXVIII). Combined with a Git auto- commit, this produces many short commits. Acceptable for dozens of users ; to watch beyond.

Manual cleanup (`git gc`, `git filter-repo`) is possible but requires coordinating with users since it rewrites history.

59.7 — Testing the connector

There is no *Test connection* button and no server-side `git ls-remote` probe. Validate the connector by exercising it :

- **First real push**. Modify a prompt and verify that the change appears in the remote repository within seconds.
- **From the shell**. On the instance, `git -C ~/.izibuddy/templates.repo status` and `git -C ~/.izibuddy/templates.repo log --oneline -5` confirm the working tree is clean and commits are being produced.

59.8 — Operational considerations

- **Latency**. Each write triggers a push. On a nearby repository (GitHub, GitLab.com), this is typically 200 to 800 ms. Beyond, the interface becomes noticeably slow ; consider a local mirror (Gitea, Forgejo) on the internal network.
- **Remote outages**. When GitHub is unreachable, IziBuddy continues to operate — commits are made locally and the failed push is logged as a warning. There is no dedicated push queue or queue-depth notification ; the next successful auto-push (or the startup sync) catches the local commits up to the remote.
- **Remote repository confidentiality**. Templates sometimes contain sensitive information (parameter sets, identities).

Choose a **private** repository on the remote side and audit the collaborator list regularly.

59.9 — Multi-backend behaviour

The connector covers a niche but legitimate setup : a single user running two file-based IziBuddy backends that share one git remote — the canonical example being a laptop mirrored to a small cloud VM. Auto-commit + auto-push cover the write path ; for the read path, the `[git]` section of `izibuddy.ini` (Chapter XLVIII § `[git]`) wires up two hooks :

- **`sync_on_startup`** (default `true`) runs `git fetch` + a pull with the configured strategy once when the backend boots. Cost : one network round-trip at startup, no impact on steady- state.
- **`sync_on_read`** (default `false`) runs the same fetch + pull before each prompt read (open, copy, render), throttled by `sync_throttle_seconds`. Cost : ~100-500 ms per first read in a throttling window — keep off unless the two-backend scenario applies.

Both hooks call `GitManager.fetch_and_pull_if_behind`, which short-circuits when `git rev-list HEAD..@{u}` reports zero behind. On a divergence that the configured strategy cannot resolve (`ff-only` against a non-fast-forward, or a conflict on `rebase` / `merge`), the warning is logged and the read proceeds on the local view — IziBuddy never auto-resolves. The user clears the divergence from the *Git Settings* modal.

Recommendations by topology (Chapter XXVII §multi-backend topologies) :

- **Single backend.** Defaults are fine — `sync_on_startup=true` is harmless, `sync_on_read=false`.
- **Multi-backend on a shared DB.** Git is not applicable ; the `[git]` section is ignored. Leave the defaults.
- **Multi-backend on a shared git remote.** Turn `sync_on_read=true` on the secondary backend ; keep `pull_s-`

`trategy=ff-only` and accept the occasional manual resolution rather than letting auto-merge invent content.

Chapter LX — Webhook connector

IziBuddy has exactly **one** outbound webhook, and it is narrow in scope : the **alert webhook**. It is not a generic event bus. There is no `[webhooks]` section, no catalogue of `prompt.*` / `user.*` / `group.*` events, no per-destination subscriptions, no HMAC-signed envelope, no delivery retries and no admin webhook endpoints. The only thing IziBuddy POSTs to the outside world is an **alert**, to a single URL, in a Slack-compatible format. This chapter documents that one mechanism as it actually exists.

60.1 — What the webhook is

The webhook is one of the delivery **channels** of the alert system (Chapter XLVI). When the alert manager raises an alert — for example an error-rate threshold crossed or repeated failed logins — it fans the alert out to the channels listed in `[alerts] channels`. If `webhook` is among them and `[alerts] webhook_url` is set, the `WebhookChannel` (`src/core/alerts/channels.py`) sends the alert as an HTTP POST.

That is the whole surface : no other part of IziBuddy emits webhooks. Template creation, sharing, user registration, group changes, Git outcomes — none of these produce an outbound webhook.

60.2 — Configuration

There is a single key, in the `[alerts]` section of `izibuddy.ini` :

```
[alerts]
enabled = true
```

```
channels = log, webhook
webhook_url = https://hooks.slack.com/services/T00/B00/XXXX
```

- **`channels`** must include `webhook` for the webhook to fire (it is a comma-separated list among `log`, `email`, `webhook`).
- **`webhook_url`** is the single destination URL. There is no second URL, no per-event routing, and no secret — the value is used verbatim (`izibuddy.ini` performs no `${VAR}` substitution), so protect the URL itself as the credential it effectively is (a Slack incoming-webhook URL is unguessable by design).

Alerts are subject to the alert cooldown (`alert_cooldown_minutes`, default 30) that de-duplicates similar alerts, so the webhook does not fire more often than the alert engine itself.

60.3 — Payload shape

The POST body is built for Slack-style *incoming webhooks* and is also broadly compatible with other receivers that accept that shape. It carries a top-level `text`, the full alert object, and a Slack `attachments` block:

```
{
  "text": "[CRITICAL] Error threshold exceeded",
  "alert": { "...": "the full alert object" },
  "attachments": [{
    "color": "#d00000",
    "title": "IziBuddy Alert: error_threshold",
    "text": "50 errors in the last hour",
    "fields": [
      { "title": "Severity", "value": "critical", "short": true },
      { "title": "Time", "value": "2026-05-17T14:32:00Z", "short": true
    ],
    "footer": "IziBuddy Alert System"
  }]
}
```

The `color` reflects severity: green (`#36a64f`) for info, orange (`#daa038`) for warning, red (`#d00000`) for critical. The request is sent with `Content-Type: application/json` and a 10-second timeout.

60.4 — Common receivers

- **Slack**. The native target — point `webhook_url` at a Slack incoming-webhook URL and the message renders directly, using the `text` and the coloured attachment.
- **Discord / Mattermost**. These accept the Slack-compatible payload on their Slack-compat endpoints (Discord's `.../slack` webhook variant, Mattermost incoming webhooks).
- **Microsoft Teams and arbitrary systems**. Teams expects its own *MessageCard* shape and will not render this payload as-is; the same is true of PagerDuty/Opsgenie. To reach those, POST to a small adapter (a serverless function) that receives this JSON and reshapes it. IziBuddy itself emits only the one Slack-style body.

60.5 — Delivery behaviour

Delivery is best-effort and stateless:

- **One attempt.** There is **no** retry loop and **no** persistent queue. If the POST fails, IziBuddy does not resend it.
- **Timeout.** Requests time out after 10 seconds.
- **Failure handling.** A response status ≥ 400, a timeout, or a transport error is written to the application log (`izibuddy.alerts`) as an error and the send returns failure. The alert itself is still recorded and still delivered on any other configured channel (log, email).

Because there is no retry or queue, treat the webhook as a convenience notification, not a guaranteed audit trail — the `log` channel and the alerts page (Chapter XLVI) remain the source of truth.

60.6 — Testing the connector

Use the alert system's own test endpoint rather than a dedicated webhook button (there is none) :

- **Alert test.** `POST /api/admin/alerts/test` (Chapter XLVI) raises a synthetic alert through the configured channels ; with `webhook` in `channels` and a valid `webhook_url` , you should see the test message arrive at the receiver.
- **Manual probe.** From the server, `curl -X POST -H 'Content-Type: application/json' -d '{"text":"ping"}' "$webhook_url"` isolates a network or URL problem independently of IziBuddy.

60.7 — Troubleshooting

- **Nothing arrives.** Confirm `webhook` is present in `[alerts] channels` and that `webhook_url` is non-empty and correct ; check `izibuddy.log` for `izibuddy.alerts` errors.
- **`404` / `410` from Slack.** The incoming-webhook URL was mistyped or revoked ; regenerate it on the receiver side.
- **The message renders as raw JSON.** The receiver does not accept the Slack payload shape — put an adapter in front (see *Common receivers*).
- **Timeouts.** The receiver is slow or unreachable ; since there is no retry, the alert is simply logged as a webhook failure and you rely on the other channels.

Chapter LXI — Lemon Squeezy integration

IziBuddy carries an integration with **Lemon Squeezy** for payment, subscription and invoicing. Read this first : **the paid tier is intentionally disabled for the initial release.** Out of the box the license status is `inactive` and every instance runs on the **freemium** plan (a 10-template limit). The plumbing described below exists in the code and can be wired up, but it is not activated in the shipped product ; treat this chapter as a reference for what is present, not as a go-live checklist. The license lifecycle proper (states, expiration, restoring on a new host) is covered by Chapter XLIV.

61.1 — What Lemon Squeezy does

Lemon Squeezy is the merchant-of-record : it hosts checkout, processes payments, computes VAT, issues invoices, and exposes a customer portal. IziBuddy is *notified* of subscription events through webhooks and adjusts the license accordingly — no IziBuddy code touches a card number or a tax form.

Notably, license activation does **not** require a Lemon Squeezy API key : Lemon Squeezy provides a public endpoint to validate license keys, and manual key validation works with no server-side secret. The webhook secret is the only optional secret, and only if you enable webhooks.

61.2 — The configuration file

Configuration lives in `src/config/lemonsqueezy.yaml` — there is **no `[lemon_squeezy]` section in `izibuddy.ini`** and no `LEMONSQUEEZY_API_KEY` / `LEMONSQUEEZY_STORE_ID` requirement. The file declares the plans and the checkout base URL, and lists products for display only :

```
products:
  pro_lifetime:  { variant_id: "", type: one_time,    name: "Pro Lifetim
  pro_monthly:   { variant_id: "", type: subscription, name: "Pro Monthl
  pro_yearly:    { variant_id: "", type: subscription, name: "Pro Annual

plans:
  freemium:
    template_limit: 10
    name: "Freemium"
  pro:
    template_limit: null   # unlimited
    name: "Pro"

checkout_base_url: "https://izibuddy.lemonsqueezy.com/checkout/buy"

# webhook_secret: "${LEMONSQUEEZY_WEBHOOK_SECRET}"
```

Two things to note. First, the `products` block is **display metadata only** — there is no variant-to-plan mapping table that the manager consults on each webhook; the effective plan is `freemium` or `pro`, with limits taken from the `plans` block. Second, this YAML file **does** support `${VAR}` substitution (unlike `izibuddy.ini`), so the optional `webhook_secret` can be sourced from an environment variable.

61.3 — The webhook endpoint

If you enable webhooks in Lemon Squeezy, point the endpoint at `https://<your-izibuddy>/api/webhooks/lemonsqueezy` and paste the secret you generated (which becomes `webhook_secret` in the YAML). Each request carries an `X-Signature` header; the server verifies it as `HMAC-SHA256(webhook_secret, raw_body)` in constant time and rejects a bad signature with 401.

> ***Security note.*** *Signature verification is currently* ***skipped when the `X-Signature` header is absent*** *(`src/api/license.py`). Since the paid tier is disabled for the initial release the webhook is not in use, so the gap is not exploitable today — but it must be closed (reject unsigned requests) before the integration is turned on. See Chapter XLIV.*

The events actually handled are `order_created`, `subscription_created`, `subscription_updated`, `subscription_cancelled` / `subscription_expired`, and `license_key_created`; any other event name is logged and ignored. There is no handling of `subscription_payment_success` / `_failed` or `order_refunded` as distinct events.

To verify reachability without spending a test subscription, hit `GET /api/webhooks/lemonsqueezy/test` from outside your network; it returns `{"status": "ok", "message": "Webhook endpoint is reachable"}` with no auth. Chapter LII explains how to expose only this single route while keeping the rest of the surface private.

61.4 — The checkout link

The user-facing « Buy IziBuddy Pro » link is a bare Lemon Squeezy checkout URL built from the `checkout_base_url` and a variant id :

```
https://<your-store>.lemonsqueezy.com/checkout/buy/<variant_id>
```

If you mint it from inside IziBuddy, `GET /api/license/checkout/{variant_id}` returns exactly this URL (`{"url": "…"}`). It does **not** append `checkout[email]` or a `checkout[custom][user_id]` field — there is no automatic account-attribution bridge in the current build ; attribution is handled from the webhook/license-key flow, not from custom checkout data.

61.5 — Customer portal

For self-service (update card, cancel, download invoices), `GET /api/license/portal` returns the Lemon Squeezy portal URL when a `lemon_customer_id` is known locally. In the current build this is the **static** `https://app.lemonsqueezy.com/my-orders` URL, **not** a per-customer signed portal session obtained through the Lemon Squeezy API. The admin panel exposes it as a « Manage subscription » link.

61.6 — Going live (roadmap)

Because the paid tier is off by default, enabling it is a deliberate future step, not a config toggle in this release. When the integration is activated it will need, at minimum : closing the unsigned-webhook gap, populating real `variant_id`s and the `webhook_secret`, and deciding how accounts are attributed to subscriptions. Until then, every instance operates on freemium with the 10-template limit.

Appendices

Appendix A — Appendix A — Glossary

Short definitions of the terms in the IziBuddy domain. Each entry points to the chapter of the manual where the notion is introduced or detailed. Transversal reading : useful at the start of a career on the product, and as an aide-mémoire during reading.

A.1 — A — D

Administrator. User with extended rights on the instance (user management, license, diagnostics). Distinct from the *group administrator* (→ Chapters XXV, XLII).

Group administrator. A group has just two membership levels : plain *members* and *admins*. A group admin manages the group's members and shared templates. There is no editor/viewer role tier and no per-group Git configuration (→ Chapter XXV).

REST API. The HTTP interface exposed by the dashboard ; every action available in the UI is also callable over HTTP (→ Chapter LI).

Contextual tip. Non-blocking toast suggesting an optimization based on actual usage (« Did you know that… ») (→ Chapter XXXIX).

Audit. Log of sensitive actions on the instance, configurable to meet regulatory requirements (→ Chapter XLVI).

Auto-commit. Automatic Git commit triggered by every prompt modification when the Git mirror is active (→ Chapter LIX).

Backend. In the *connector* sense, the single active database implementation (filesystem, postgresql, mongodb, s3…). One backend is active at a time and stores the whole instance (→ Chapters LIV–LVI).

Floating button. The visible trigger in a chatbot's page that opens the IziBuddy panel (→ Chapter XXX).

Bulk update. Grouped update : N prompt modifications applied via a single API call (→ Chapter XIV).

Category. Categorization in IziBuddy is realized through **tags** (including hierarchical tags) — there is no separate « category » field or API. Where older material says « category », read « tag » (→ Chapters IX, X).

CLAUDE.md. Repository instruction file, read by AI assistants working on the source code ; no direct relation to product usage (internal reference).

Collection. Container of *class* type (POO²L L5 level) that supports inheritance and virtual templates (→ Chapter XI).

Connector. Pluggable module behind a stable interface : database backend, AI platform, SMTP, Git mirror, and the single outbound alert webhook (→ Chapter LIV).

Container. Organizational element hosting templates : either a folder (pure storage) or a collection (POO²L class) (→ Chapter XI).

Content script. JavaScript script injected by the extension into a chatbot's page to detect the editor and drop the prompt in it (→ Chapter LVII).

Cross-pane F5/F6. In the split view, `F5` copies the selection to the other pane and `F6` moves it ; `Tab` switches the active pane (→ Chapter XIII, Appendix C).

Diff. Comparison between two versions of a prompt, in unified or side-by-side mode (→ Chapter XXVIII).

Diagnostics. Administration page that reports the configured storage type and instance state. It does not run live network health checks against the connectors (→ Chapter XLVII).

Folder. Container of pure storage type, with no specific POO²L semantics (→ Chapter XI).

A.2 — E — L

Smart editor. The dashboard's web editor, with POO²L syntax highlighting, autocompletion and contextual validation (→ Chapter XVI).

Browser extension. Manifest V3 add-on that injects the floating panel into the page of target chatbots (→ Chapters IV, XXX).

Feature. Unitary UI element that can be enabled / disabled independently, grouped by category and by preset (→ Chapter XXXVIII).

Negative filter. Filter whose semantics is inverted by the `filtersExclude` option (→ Chapter XII).

Group. Multi-tenant scope sharing templates and tags between members. Two membership levels only : members and admins (no editor/viewer roles) (→ Chapter XXV).

Git applicable. Property of an IziBuddy install whose storage backend is the file-based default (no `[database]` configured). Only on git-applicable installs does the auto-sync layer (`sync_on_startup`, `sync_on_read`) and the *Git Settings* modal apply (→ Chapter XXVII).

Inheritance. Relationship between two templates by which the second one inherits the first one's parameters, constants and structure (→ Chapter XX).

Include. Inclusion of another prompt inside a prompt at render time (→ Chapter XX).

Injection. The action by which the extension drops the rendered prompt into the target chatbot's editor (→ Chapters XXXII, XXXV).

Instance. A distinct installation of IziBuddy (server + data) (→ Chapters III, LIV).

Invitation. Time-limited link sent by e-mail to make a user join a group (→ Chapter XXVI).

JWT. Authentication token format used by the REST API (→ Chapter LI).

Lab tests. Experimental tab for self-evaluation of templates against test cases (feature `labTests`) (→ Chapters XXXIX, XLVII).

License. Key controlling activatable features and limits (→ Chapter XLIV).

Locked. Marker of a prompt write-locked (→ Chapter VIII).

A.3 — M — P

Manifest V3. Specification of Chromium browser extensions used by IziBuddy (→ Chapter IV).

Marker. State flag attached to a prompt : Important, Locked, WIP, Disabled, Template, Hidden (→ Chapter VIII, Appendix B).

Migration. Switch from one database backend to another, driven by the CLI migration utility (`database_migrator.py`) and the backend migration wizard — not applied automatically at startup (→ Chapters LV, LVI).

MTA. Mail Transfer Agent — the SMTP server that relays outbound messages (→ Chapter LVIII).

Multi-backend. Deployment topology where two or more IziBuddy backends run in parallel against a shared source of truth — either a shared database (DB backend) or a shared git remote (file backend). Different topologies enable different sync mechanisms (→ Chapter XXVII).

Owner. The single user who can edit a prompt. Shared templates are read-only for everyone else ; sharing never grants write access (→ Chapters XXIV, XXVII).

Multi-selection. Selection of multiple templates via Ctrl-click, Shift-click or *rubber band* for bulk operations (→ Chapter XIV).

Floating panel. The IziBuddy panel injected by the extension into a chatbot's page (→ Chapter XXX).

Parameter. Value declared in a prompt and requested from the user at injection time (→ Chapter XVII).

Parameter set. Named set of values that can be selected in one click at injection time (→ Chapter XXXV).

Attachment. File attached to a prompt, transmitted to the chatbot at injection time (→ Chapters XIX, LVII).

POO²L. *Prompt Object-Oriented² Language*, IziBuddy's declarative language for composing and inheriting templates (→ Chapter I).

UI preset. Profile of feature activation — *Simple*, *Standard*, *Advanced*, *Expert* (→ Chapter XXXVII).

Template. The block of text (and metadata) that IziBuddy stores, edits, shares and injects (→ Chapter VI).

Rendered prompt. The final text resulting from parameter substitution, inheritance application and include expansion (→ Chapter XXII).

A.4 — R — Z

Rendering. The process of transforming a POO²L prompt into final text destined for the chatbot (→ Chapter XXII).

Rubber band. Selection by elastic frame drawn with the mouse (feature `actionRubberBand`) (→ Chapter XIV).

Read-only prompt. A prompt the current user can view and use but not modify, because they are not its owner. The dashboard hides the editor's *Save* button (→ Chapter XXIV).

Scope. Ownership perimeter of a prompt : user or group (→ Chapter XXV).

Shared prompt. A prompt whose owner has granted view + use access to other users (personal scope) or to all group members (group scope). Sharing never transfers ownership and never grants write access (→ Chapter XXIV).

Synchronised remote. A git remote that two file-backed IziBuddy backends both push to and pull from, acting as the shared source of truth across machines. Enabled by the `sync_on_startup` and `sync_on_read` keys of `izibuddy.ini` (→ Chapters XLVIII, LIX).

Scoring. A **computed quality score out of 10** produced by the analysis engine — not a user star-rating system (→ Chapter XXIX).

Section. Named, delimited fragment of a prompt body, declared with `@@@section[?|!] [virtual|final] <name>`. The presence toggle (`?` / `!`) glues to the `section` keyword and the inheritance modifier (`virtual` / `final`) comes **after** the keyword, before the name. There is no section numbering. The unit of override under inheritance (→ Chapter XXI).

Section toggle (`?` / `!`). Suffix glued to the `section` keyword that turns the section into a *user choice at instantiation* : `?` defaults the dialog to OFF (opt-in), `!` defaults it to ON (opt-out). Both require a description, which is the text the user reads in the include/exclude dialog — and nothing else : it never reaches the rendered text, whose heading is the section's name. Purely a presence toggle — does **not** affect scoring or weight (→ Chapter XXI).

Virtual section. Section marked `@@@section virtual <name>` : *abstract*, carries no body, every concrete descendant in the inheritance chain must implement it (→ Chapter XXI).

Final section. Section marked `@@@section final <name>` : *sealed*, no descendant may redeclare it — not even to change the toggle or the description (→ Chapter XXI).

Self-hosted. Installation hosted by the using organization, as opposed to an instance hosted by the publisher (→ Chapter III).

Server-Sent Events. Unidirectional HTTP channel the extension uses to receive server → client notifications from the IziBuddy backend (`/api/events/stream`). It is not used to read chatbot responses (→ Chapters XXXIV, LVII).

Soft delete. Reversible move to the trash, as opposed to definitive purge (→ Chapter VII).

Stash. A set of templates kept handy in the extension. It is **not** extension-local storage : stashed templates are backend templates tagged `STASH` (→ Chapters XXXII, XXXV).

Git divergence. A state where the local instance repository and its remote have diverged ; IziBuddy logs it and keeps the local view rather than auto-resolving (→ Chapter LIX).

Tag. Free label, flat or hierarchical, attached to a prompt (→ Chapter X).

Template. Marker of a prompt serving as an educational base for others (→ Chapter VIII).

Tip. See *contextual tip*.

Token. Access token — in the sense of a time-limited invitation or an HTTPS Git credential (→ Chapters XXVI, LIX).

Tristate. Three-position toggle : on, partial, off, used in the features panel (→ Chapter XXXVIII).

UI level. Former three-position interface level, now replaced by presets (backend compat preserved) (→ Chapter XXXVII).

Verbatim. Block of a POO²L prompt whose content is not interpreted ; wrap content in `@@@verbatim…@@@end-verbatim` to pass it through unchanged (→ Chapter XLI).

Version. Dated snapshot of a prompt, accessible for diff and rollback (→ Chapter XXVIII).

Split view. Two-pane layout for editing and consulting simultaneously (→ Chapter XIII).

Webhook. The single outbound alert webhook : a Slack-format HTTP POST sent when the alert system fires, if `webhook` is among `[alerts] channels`. There is no general event webhook (→ Chapter LX).

WIP. *Work In Progress* marker on templates under construction (→ Chapter VIII).

Appendix B — Appendix B — Icon legend

Annotated catalogue of the prompt markers, container-kind icons, the score badge and the shared badge used in the dashboard and the extension's floating panel. Each entry gives the exact semantics of the symbol, its scope, and the chapter that details its behavior.

B.1 — Markers on templates

Markers are state flags attached to templates. A prompt can carry several markers simultaneously ; each marker corresponds to a dedicated filter in the prompt list (→ Chapters VIII, XII).

⭐ **Important.** Marks a prompt deemed priority or favorite. Used for sorting and quick filtering. Does not affect any other behavior : the prompt remains editable, shareable, deletable. Toggled by clicking the star in the list, or via the context menu.

🔒 **Locked.** Marks a prompt write-locked. Editing is refused (a warning message shows up if you try to modify it) ; the content can still be consulted, duplicated and injected normally. Unlocking requires the owner or admin role.

WIP. *Work In Progress*, prompt under construction. No blocking effect ; signals to other users (and to oneself the following week) that the prompt is not considered stable. Filterable via `filterWip`.

Disabled. Deactivated prompt. Hidden from lists by default ; only visible when the *Include disabled* filter (`filterDisabled`) is active. Used to set aside an obsolete prompt without deleting it.

Template. Marks a prompt serving as a pedagogical base or duplication pattern. A *Template* prompt is typically referenced by others through POO²L inheritance. Filterable via `filterExample`.

Hidden. Template hidden from default views, for example for infrastructure templates the user does not want to see in the daily list. Visible when `filterHidden` is active.

B.2 — Frequent marker combinations

A few typical combinations encountered in the field :

- ⭐ **+ Locked.** Favorite, frozen, shared as-is prompt. The *team reference* pattern (for example, *follow-up email template*).
- **WIP + Hidden.** Template under construction that you don't want to see in the list until it is shippable. To remove on publication.
- **Template + Locked.** Reference pattern you want to protect from any accidental modification. Inheritance children override sections without touching the parent.
- **Disabled + ⭐.** Archived variant of a prompt you want to keep handy (Important so as not to lose it in filtering, Disabled so as not to use it inadvertently).

B.3 — Container kind icons

Containers carry an icon that reflects their **kind** (`src/static/index-container-kind.js`):

📁 **Folder.** Pure storage container. No POO²L semantics attached. The folder may contain other folders, containers, or directly templates.

📚 **Collection.** POO²L L5 container — a *class* of templates. Supports inheritance, virtual templates, and same-member shadowing via the collection axis (`^parent`). A folder may be *promoted* to a collection when the organization warrants it (→ Chapter XI).

📦 **Library.** A library container.

🛠️ **Project.** A project container (boundary folder).

There are no transient promote/demote arrows and no truncated-path ornament — the container simply shows the icon of its current kind.

B.4 — Score badge

Score. When scoring is active (feature `viewScoring`), the badge shows a **computed quality score out of 10** produced by the analysis engine — it is not an average of user ratings, and it is not a 0-100 scale (→ Chapter XXIX).

There are no per-template *New*, *Modified*, *Versions*, *Syncing* or *Conflict* badges in the list; those states are not surfaced as prompt badges.

B.5 — Scope indicator

🎁 **Shared.** A prompt shared to a group shows the gift badge (`template-shared-badge`, `src/static/index-templates-render.js`). There is no 🤝 icon and no per-connector status dots or tag-tree ornaments (▸ ▾ • ↑) in the interface.

B.6 — Editing conventions

A few signs used by the smart editor:

~ Red squiggles. Unresolved reference in the POO²L source.

~ Yellow squiggles. Deprecated reference or include towards an obsolete prompt.

ⓘ **Info bubble.** Hover that opens a contextual explanation — typically for markers and badges.

↩ Indented carriage return. Formatted multi-line output in the editor; useful for `verbatim` blocks.

B.7 — Visual consistency

The dashboard icons and the floating panel icons come from a single set ; the same marker has the same appearance on both sides. The icon font is embedded — no network dependency. The colors are driven by design tokens (→ Chapter XXXVI) ; a custom theme can modify the indicator hues without changing their semantics.

Appendix C — Appendix C — Keyboard shortcuts — reference card

Reference card of the shortcuts that actually exist in IziBuddy. Unless otherwise noted, the combinations apply to Linux and Windows ; replace `Ctrl` with `Cmd` (⌘) on macOS. Shortcuts not listed here are not bound — IziBuddy deliberately keeps a small set.

C.1 — Split pane

When the split pane is active (feature `viewSplitPane`), two panes are shown side by side. `Tab` switches which pane is active ; `F5` and `F6` move the current selection **between** the panes. These keys take no modifier — any modifier makes them fall through to the browser.

Shortcut	Action
`Tab`	Switch the active pane
`F5`	Copy the selection to the other pane
`F6`	Move the selection to the other pane

C.2 — Multi-selection on the prompt list

Shortcut	Action
`Ctrl + click`	Add / remove a prompt to the selection
`Shift + click`	Extend the selection to the clicked prompt
Rubber-band drag	Elastic frame selection in an empty zone
Rubber-band + `Ctrl`	Extend the current selection instead of replacing it

Shortcut	Action
`Del`	Move the selected templates to the trash (in advanced-selection mode)
`Ctrl + X`	Cut to the IziBuddy clipboard
`Ctrl + C`	Copy to the IziBuddy clipboard
`Ctrl + V`	Paste into the current container

Drag-and-drop of templates and containers always performs a **move** — there are no `Ctrl` / `Shift` / `Alt` drag modifiers.

C.3 — Undo / redo

Shortcut	Action
`Ctrl + Z`	Undo the last operation
`Ctrl + Y`	Redo the undone operation
`Ctrl + Shift + Z`	Redo (macOS / Linux variant)

The undo history is application-level (not just textual) : a deletion, a move, a tag change are all undoable. The history depth is limited to the current session.

C.4 — Editing a prompt

Shortcut	Action
`Ctrl + S`	Save the current prompt

`Esc` does **not** close the prompt editor (it was intentionally removed there) ; it still closes the category dialog and the colour picker. Autocompletion of POO²L constructs is triggered by typing `@@@`, not by a shortcut.

C.5 — Extension floating panel

Inside the extension's floating panel:

Shortcut	Action
`Ctrl + Shift + I`	Open / close the floating panel
`↑` / `↓`	Navigate the prompt search results
`Enter`	Select the focused result
`Esc`	Close the search dropdown

The parameters dialog uses `Enter` to confirm and `Esc` to cancel. There is no `Space`, `Ctrl + Enter` or `S` shortcut in the panel.

C.6 — Customizing shortcuts

The set above is fixed in version 1.0. The one place you can rebind is the browser-level extension shortcut (`Ctrl + Shift + I`): open your browser's shortcut configuration page (Chrome / Edge: `chrome://extensions/shortcuts`).

Appendix D — Appendix D — Error catalog

There is an important correction to make up front : IziBuddy has **no** custom REST error envelope. The API does **not** return an `{"error": {"code", "message", "details", "error_id"}}` object, and the string codes people expect (`BAD_REQUEST`, `VALIDATION_ERROR`, `DB_*`, `STORAGE_*`, `SMTP_*`, `GIT_*`, `LICENSE_*`, `POOL_*`, `WEBHOOK_DELIVERY_FAILED`, …) **do not exist** in the code. This appendix documents what IziBuddy actually returns.

D.1 — The two response shapes

Errors come back in one of two shapes :

- **FastAPI `HTTPException`** — the common case — returns `{"detail": "..."}` with an appropriate HTTP status (400, 401, 403, 404, 409, 422, 500). The `detail` is free-form text, not a coded identifier.

```
json { "detail": "Template not found" }
```

- **Endpoint-level handled errors** in some routers return a `success` flag plus a message :

```
json { "success": false, "error": "Template <id> not found" }
```

There is no `code` field to switch on and no `error_id`. A client should key off the **HTTP status** and treat the message as human-readable text. Pydantic validation failures produce FastAPI's standard 422 body listing the offending fields.

D.2 — Correlating an error in the logs

The per-request identifier is the **correlation id**, not an `error_id`. Every request is tagged with one, returned in the `X-Correlation-ID` response header and attached to every log line for that request (Chapter XLIX). To escalate an incident, capture that value and search the log :

```
grep '"correlation_id":"<value>"' izibuddy.log
```

D.3 — Git endpoints on a non-file backend (HTTP 409)

One coded-looking behaviour is real, though the *label* is not : every endpoint under `/api/git/*` returns **HTTP 409** when the active storage backend is not file-based. The database is itself the shared source of truth and the Git layer does not apply — there is nothing to push, pull or inspect. This is expected behaviour on DB installs, not a configuration error ; the dashboard hides the *Git Settings* entry on the same signal (→ Chapter XXVII §when git is applicable). The response is a plain `{"detail": "..."}` 409, with no `GIT_*` code.

D.4 — The real typed codes : POO²L engine diagnostics

The only place IziBuddy exposes **stable, typed** codes is the POO²L **engine**. When you render or validate a prompt, the engine (`pool-core`) reports diagnostics using the specification's 51 ERROR codes and 3 WARNING codes (`src/engine/crates/pool-core/src/diagnostic.rs`). These are engine diagnostics surfaced by the render/validate paths — they are **not** HTTP status codes and do not travel in an `error.code` field. The most relevant to the render / inheritance / virtual cases :

Code	When it fires
`INVALID_SYNTAX`	The source does not parse (the old `POOL_PARSE_ERROR`).
`UNDEFINED_REFERENCE`	A `@@@=name` / include target is not defined (the old `POOL_REFERENCE_UNRESOLVED`). It is a **blocking** error, not a warning.
`UNKNOWN_COLLECTION`	An `inherits`/include names a collection that does not exist.
`CIRCULAR_INHERITANCE`	A cycle in the prompt inheritance chain (the old `POOL_INHERITANCE_CYCLE`).
`CIRCULAR_COLLECTION_INHERITANCE`	A cycle in collection inheritance.
`CIRCULAR_INCLUDE`	A cycle among includes.
`AMBIGUOUS_INCLUDE`	An unqualified include resolves to more than one target.
`UNIMPLEMENTED_VIRTUAL_TEMPLATE`	A `virtual` prompt is instantiated/rendered without a concrete implementation.
`UNIMPLEMENTED_VIRTUAL_SECTION`	A `virtual` section reaches render with no concrete override.
`OVERRIDE_OF_FINAL_TEMPLATE`	A derived prompt overrides one marked `final`.
`OVERRIDE_OF_FINAL_SECTION`	A descendant redeclares a `final` section (even changing only the toggle or description).

Code	When it fires
`OVERRIDE_CONTRACT_VIOLATION`	A section override tightens the toggle contract in a disallowed way.
`VIRTUAL_TEMPLATE_HAS_BODY`	A `virtual` prompt carries a body.
`VIRTUAL_SECTION_HAS_BODY`	A `virtual` section carries body text (only whitespace / inline comments are allowed).
`VIRTUAL_SIGNATURE_MISMATCH`	An implementation's signature does not match the virtual contract.
`SECTION_SHAPE_MISMATCH`	A section is redeclared with an incompatible shape (e.g. body vs subsections).
`INCLUDE_OF_VIRTUAL_SECTION`	An include resolves to a `virtual` section that has no body.
`TYPE_MISMATCH` / `CONSTRAINT_VIOLATION` / `SELECT_INVALID_OPTION`	A parameter value has the wrong type, violates its constraint, or is not among a select's options.
`SYSTEM_VARIABLE_OVERRIDE`	A parameter or constant shadows a reserved system-variable name.

Note that `VIRTUAL_FINAL_CONFLICT`, `POOL_PARSE_ERROR`, `POOL_INHERITANCE_CYCLE` and `POOL_REFERENCE_UNRESOLVED` are **not** real codes — they were never part of the catalogue. The three WARNING codes (advisory, never blocking a render) are `MISSING_DESCRIPTION`, `DEPRECATED_TEMPLATE` and `PARAMETER_NAME_IS_PREFIX`.

D.5 — Client-side strategy

A client of the REST API should :

1. Read the **HTTP status** and branch on it (4xx : the caller must fix the request ; 5xx : retry with backoff).
2. Tolerate **both** body shapes (`{"detail": …}` and `{"success": false, "error": …}`) and present the message text.
3. Capture the `X-Correlation-ID` header for support escalations.
4. For template-authoring errors, read the engine diagnostic code and message returned by the render/validate call — that is where the precise, typed information lives.

D.6 — Reference

REST errors are raised per router as `HTTPException(status_code, detail)` in `src/api/*.py` . The engine diagnostic codes are defined once in `src/engine/crates/pool-core/src/diagnostic.rs` and mirror the POO²L specification’s §6.2 catalogue verbatim.

Appendix E — Appendix E — FAQ

Twenty-two questions that come up regularly on forums and in support tickets, grouped by broad theme. The cross-references point to the chapters that cover each subject in depth.

E.1 — Accounts and first logins

1. How do I create an account on IziBuddy ?

On a self-hosted instance, the first registered user becomes administrator ; the next ones register via the landing-page form and receive a verification e-mail (if SMTP is configured). On a hosted instance, the organization's administrator invites each user by e-mail (→ Chapters III, XXVI).

2. I lost my password, what should I do ?

Click *Forgot password* on the login page, enter the e-mail address, follow the received link. The token expires after 24 hours. If the e-mail doesn't arrive and you are the administrator of the instance, check the application log : the link is displayed there in degraded mode when SMTP is not configured (→ Chapter LVIII).

3. Why is my account marked « pending verification » ?

Because the verification e-mail has not yet been clicked. Check spam ; ask an administrator to resend the link from the user management page (→ Chapter XLIII).

4. Can I use IziBuddy without creating an account ?

Not on a multi-user instance. The single-user desktop launcher (Chapter I) offers an account-less mode that uses the *Single User* connector locally.

E.2 — The extension

5. The extension does not appear on the chatbot page, what should I do ?

Check in order : (a) the extension is installed and enabled in `chrome://extensions` ; (b) the open chatbot is one of the five supported platforms — Claude, ChatGPT, Grok, Gemini, DeepSeek (→ Chapter I) ; (c) the extension has permission to act on this domain ; (d) the panel is collapsed — click the floating button. If nothing works, reload the page.

6. How do I pair the extension with my dashboard ?

Config tab of the extension, enter the server URL (for example `https://izi.example.com`) and the credentials. The *Remember me* option avoids re-entering at every session (→ Chapter XXXI).

7. Template injection does not work — the chatbot's editor stays empty.

This is often due to a recent redesign of the target platform (→ Chapter LVII). Update the extension. If the error persists, the panel exposes a *Copy rendered prompt* mode that lets you paste manually.

8. Are my templates sent to the servers of OpenAI / Anthropic / Google ?

Only when you click *Inject* on a prompt and send the message on the chatbot side. IziBuddy itself does not contact any AI provider (→ Chapter I).

E.3 — Writing templates

9. The render of my prompt shows `@@@=XXX` unresolved instead of the expected value.

The substitution form is `@@@=XXX` (there is no `{{XXX}}` form). Either `XXX` is not a declared parameter of the prompt, or `XXX` is not a known system variable — an unknown `@@@=XXX` is a blocking `UNDE-`

`FINED_REFERENCE` error, not a silent passthrough. Check the list of system variables (→ Chapters XXII, XLI) and the parameter declaration (→ Chapter XVII).

10. What is the difference between a parameter and a constant ?

A parameter is requested from the user at injection time (via a dialog). A constant is fixed at writing time of the prompt and does not appear in the injection dialog (→ Chapters XVII, XVIII).

11. When to use a collection rather than a folder ?

A folder is only for storage. A collection introduces the POO²L *class* notion — it supports inheritance, virtual templates and same-member shadowing via the collection axis (`^parent`). If you want to factor templates that share a common structure, use a collection (→ Chapter XI).

12. Is the diff between two versions of my prompt safe to review before a rollback ?

Yes, the rollback applies only after explicit confirmation and itself creates a new version (the rollback is undoable). The diff view supports two modes — unified and side-by-side (→ Chapter XXVIII).

E.4 — Collaboration and groups

13. What is the difference between sharing and duplicating ?

Sharing : another user can see and use your prompt, but cannot edit it. Duplicating : another user receives an independent copy that they can edit and that is no longer linked to yours (→ Chapter XXIV).

14. How do I leave a group ?

Groups page > select the group > *Leave the group*. If you are the sole administrator, transfer the role to another member before leaving (→ Chapter XXV).

E.5 — Git synchronization

15. The remote has diverged from my instance. What do I do ?

There is no *Synchronize* button and no automatic pull-rebase. On a divergence, IziBuddy logs a warning and keeps the local view ; with `pull_strategy=ff-only` a non-fast-forward is refused rather than merged. Resolve it manually on `~/.izibuddy/templates.repo` with ordinary Git commands (→ Chapter LIX).

16. How do I configure Git with an SSH key rather than an HTTPS token ?

`~/.izibuddy/templates.repo` is an ordinary Git working tree, so use standard Git auth : generate a key for the server account (`ssh-keygen -t ed25519`), register the public key as a *deploy key* with write access on the remote, and set `git_remote` in the `[database.multi-user-with-git]` section to the `git@…` SSH URL. There is no SSH selector in the UI and no `~/.ssh/id_izibuddy` convention (→ Chapter LIX).

17. Can I disable the Git mirror entirely ?

The Git mirror only applies to file-based backends and is controlled from the `[git]` and `[database.multi-user-with-git]` sections of `izibuddy.ini` — there is no *Enable Git* checkbox. Set `auto_push = false` to keep commits local, or switch to a non-file backend, in which case the Git layer does not apply at all (→ Chapters XXVII, LIX).

18. Where did the *Git Settings* menu go after my upgrade ?

Your install now uses a database backend (one of `postgresql` , `mysql` , `mongodb` , …) ; IziBuddy hides the entire Git UI when the database is the source of truth, since the git layer would write to a directory nothing reads from. If you want the Git mirror back, remove the custom `[database.*]` from `izibuddy.ini` and restart (→ Chapter XXVII §when git is applicable).

19. My prompt looks out of date on my second machine. What do I do ?

If both machines run file backends pointing at the same git remote, enable `sync_on_read = true` in the `[git]` section of `izi-buddy.ini` (or check the *Pull before every prompt read* box in the *Git Settings* modal). The next read will fetch + pull, throttled by `sync_throttle_seconds`. Manual *Pull* also remains available (→ Chapters XLVIII, LIX).

E.6 — Performance and operations

20. The interface becomes slow when I have thousands of templates.

Several levers, in order of efficacy : (a) switch to an indexed DBMS if you are on Filesystem (→ Chapter LV) ; (b) enable the *Simple* preset to reduce render cost (→ Chapter XXXVII) ; (c) add a `CachedDatabase` wrapper (Redis) in front of the existing base (→ Chapter LV) ; (d) move bulky attachments off the inline `base64` store to the `sqlite` or `filesystem` attachment store (→ Chapter LVI).

21. How do I back up my library ?

Three complementary levels : (a) the built-in backup manager (→ Chapter L) that exports everything as compressed JSON ; (b) the remote Git mirror (→ Chapter LIX) that records each change ; (c) infrastructure backup (volume snapshot, regular SQL dump) on the operator side.

22. My users don’t receive their invitations.

First check the SMTP configuration (→ Chapter LVIII). If SMTP is absent, invitation links are written to the application log — locate the entry to retrieve the link. If SMTP is present and mails are going out but not received, check SPF/DKIM/DMARC : a missing signature sends messages to spam.

E.7 — Quick cross-references

If you are looking to…	See
Get started in five minutes	Chapter V
Understand the folder / collection difference	Chapter XI
Build a parameterized prompt	Chapter XVII
Share with a team	Chapter XXV
Restore an older version	Chapter XXVIII
Configure the database	Chapter LV
Configure object storage	Chapter LVI
Configure SMTP	Chapter LVIII
Configure Git	Chapter LIX
Configure webhooks	Chapter LX
List all markers	Appendix B
Find a shortcut	Appendix C
Decode an error message	Appendix D
Diagnose a symptom	Appendix F

A missing question? The user forums and the publisher's support channel remain the way to flag cases that would deserve a 21st entry.

Appendix F — Appendix F — Troubleshooting decision tree

Symptom-based tree. Choose the branch that describes what you observe, walk down the sub-branches until you reach a diagnosis or a chapter cross-reference. This tree is deliberately linear : one symptom, a binary decision at each node, and a destination. For more complex failures, cross-check with the error catalog (Appendix D) and the FAQ (Appendix E).

F.1 — I cannot log in

- *Does the form show an error ?*
- **Yes** → note the exact message.
  - *« Invalid credentials »* → verify that the user exists (administrator → Users page). Forgotten password → use the reset link (→ Chapters XXVI, LVIII).
  - *« Account pending verification »* → click the link received by mail ; if absent, check spam and the SMTP configuration (→ Chapter LVIII).
  - *« Account disabled »* → an administrator has disabled the account ; contact them (→ Chapter XLIII).
- **No, the page is spinning.**
  - *Does the URL respond (ping/curl) ?*
  - **No** → the server is unreachable. Check the process, the firewall, the reverse proxy (→ Chapter LII).
  - **Yes but with a 5xx error** → the server starts up but a connector is in error. Check `diagnostic.html` (→ Chapter XLVII) and the logs (→ Chapter XLIX).

F.2 — The extension does not appear

- *Is the extension installed ?*
- **No** → install it (→ Chapter IV).
- **Yes.**
  - *Is it enabled in* `chrome://extensions` *?*
  - **No** → enable it.
  - **Yes.**
    - *Am I on one of the five supported platforms ?* (Claude, ChatGPT, Grok, Gemini, DeepSeek)
    - **No** → the extension only injects on these domains (→ Chapter I).
    - **Yes.**
      - *Is the floating button collapsed ?* → click to expand.
      - *Is the button absent ?* → reload the page (a chatbot-side DOM change can shift the injection point ; the content script reruns on reload). If persistent after several reloads, update the extension (→ Chapter LVII).

F.3 — My prompt does not inject into the chatbot

- *Does the panel show an error ?*
- **Yes** → follow the message.
- **No, the panel closes and nothing appears in the chatbot's editor.**
  - *Does the prompt have unfilled parameters ?*
  - **Yes** → the parameters dialog may have been closed prematurely. Restart.
  - **No.**
    - *Is the platform Claude or ChatGPT ?* → a recent redesign may have broken the selector ; use the *Copy rendered prompt* mode pending the update (→ Chapter LVII).

- *Is the chatbot-side session still active ?* → a chatbot-side timeout redirects to the login page and the content is lost. Log in again and retry.

F.4 — My prompt does not render correctly

- *Does the render fail on an unresolved* `@@@=XXX` *?* (the substitution form is `@@@=XXX` ; there is no `{{XXX}}` form)
- **Yes** → `XXX` is neither a declared parameter nor a system variable. This is a blocking `UNDEFINED_REFERENCE` engine error. Declare the parameter (→ Chapter XVII) or use a valid system variable (→ Chapters XXII, XLI).
- *Is the render empty or truncated ?*
- **Yes** → an include or a parent in the inheritance chain is unfound. Open the inheritance chain (→ Chapter XX) and look for the red reference.
- *Does the source fail to parse ?*
- **Yes** → the engine reports `INVALID_SYNTAX` with the line and position (→ Appendix D).

F.5 — Git synchronization fails

- *Did the* Git Settings *menu entry disappear after an upgrade or a config change ?*
- **Yes** → your `izibuddy.ini` now selects a database backend ; Git is hidden by design (→ Chapter XXVII §when git is applicable).
- *Is the second backend's content out of date ?*
- **Yes** → enable `sync_on_read = true` in `[git]` , or trigger a manual pull (→ Chapter XLVIII).
- *Does the log show an authentication failure on push ?*
- **Yes** → invalid credentials (there is no `GIT_AUTH_FAILED` code). Check the HTTPS token in `git_remote` or the SSH deploy key (→ Chapter LIX).
- *Does the log show a divergence / non-fast-forward ?*

- **Yes** → the remote has diverged. There is no *Synchronize, reset hard* or *force push* button ; IziBuddy logs the warning and keeps the local view. Resolve it manually with ordinary Git commands on `~/.izibuddy/templates.repo` (→ Chapter LIX).
- *Does the log mention a broken repository ?*
- **Yes** → follow the recovery procedure of Chapter LIX (backup of files, deletion of `.git/`, reinitialization).
- *Do pushes keep failing ?*
- **Yes** → the remote is probably unreachable. Verify `git -C ~/.izibuddy/templates.repo ls-remote` by hand from the server and check the outbound firewall.

F.6 — My mails are not going out

- *Is the SMTP mode enabled (`enabled = true`) ?*
- **No** → IziBuddy is in log-only mode. Enable SMTP (→ Chapter LVIII).
- **Yes.** (There is no *Test connection* button ; exercise a real flow and watch `izibuddy.log`.)
    - *Does the log show a send failure ?*
    - **Yes** → read the SMTP error text (there are no `SMTP_*` codes) :
        - authentication refused → incorrect `smtp_user` / `smtp_password`.
        - TLS required → set `smtp_use_tls = true`.
        - Timeout → check that port 587 is open on the outbound firewall.
    - **No, but users receive nothing.**
        - *Do mails arrive in spam ?* → improve SPF/DKIM/DMARC (→ Chapter LVIII).
        - *Are there bounces in the sender mailbox ?* → invalid recipient addresses or refusal of acceptance by their MTA.

F.7 — The interface is slow

- *Does the slowness occur on all pages ?*
- **Yes** → global bottleneck. Diagnosis :
  - Filesystem backend with many templates ? → migrate to PostgreSQL (→ Chapter LV).
  - High database latency ? → add `CachedDatabase` (→ Chapter LV).
  - Diagnostics page in orange ? → follow the indicators (→ Chapter XLVII).
- *Does the slowness occur only when opening a prompt ?*
- **Yes** → typical of a cloud-storage database backend (S3/GCS/Azure/MinIO) with high latency. Check the bucket region (→ Chapter LVI) or move to a local backend with a cache in front.

F.8 — The alert webhook does not fire

(This is the single outbound alert webhook — there is no general event webhook, no `[webhooks]` section, no per-event subscriptions.)

- *Is `webhook` listed in `[alerts] channels` ?*
- **No** → add it (→ Chapter LX).
- **Yes.**
  - *Is `[alerts] webhook_url` set to a valid URL ?*
  - **No** → set it.
  - **Yes.**
    - *Does `izibuddy.log` show `izibuddy.alerts` errors ?* → the receiver returned ≥ 400 or timed out ; there is no retry. Check the URL and that the receiver accepts a Slack-format payload (→ Chapter LX).

F.9 — The paid tier / license

Note that the paid tier is intentionally disabled for the initial release : every instance runs on freemium (10-template limit) with the license `inactive`. There are no `LICENSE_*` error codes.

- *Is the instance limited to a handful of templates ?*
- **Yes** → that is the freemium limit ; the paid tier is off in this release (→ Chapters XLIV, LXI).
- *Did a license key fail to activate ?*
- **Yes** → the endpoint returns a plain `{"detail": "..."}` message, not a code. Re-enter the key checking for stray spaces or line breaks (→ Chapter XLIV).

F.10 — I can no longer find a prompt that existed

- *Is the prompt in the trash ?*
- **Yes** → restore (→ Chapter VII).
- **No.**
  - *Is an active filter hiding the prompt ?* → use *Clear filters* in the UI (there is no keyboard shortcut for it) or check the tristate filter state.
  - *Is the right scope selected ?* (user vs group) → switch to the right scope.
  - *Was the purge used ?* → restore from the Git backup (→ Chapter LIX) or from the infrastructure backup (→ Chapter L).

F.11 — When nothing works

Three actions to try in order :

1. Reload the page. A good half of transient symptoms disappear.
2. Consult the diagnostics page (`/static/diagnostic.html`) : it reports the configured storage type and instance state (it does not run live network probes) (→ Chapter XLVII).

3. Open the application log and look for the last entry with level `ERROR` ; note its **correlation id** (the `X-Correlation-ID` value, not an `error_id`) to transmit to support.

Going through the publisher’s support is legitimate as soon as you have a reproducible **correlation id** : the exact trace is in the logs on the server side, support decodes it and identifies the cause without guessing.

Resources

Télécharger

- Format PDF (01-user-manual-outline_v1.17_en_20261008.pdf)
- Format EPUB (01-user-manual-outline_v1.17_en_20261008.epub)
- Format Markdown (01-user-manual-outline_v1.17_en_20261008.md)

Lexique

Auto-push — Toggle of the `[git]` section that conditions whether `auto_commit()` runs `git push` after each commit. Default `true`. Set `false` to keep commits local and push manually. → Ch. XLVIII, Ch. LIX

Backup — Timestamped, xz-compressed snapshot taken by `src/core/backup_manager.py`. The single wired trigger today is the pre-import backup (taken before a bulk import). Backups live under `~/.izibuddy/backups`; there is no full/incremental tiering and `max_count`/archival keys are declarative only. → Ch. L

Browser extension — Manifest V3 add-on (Chrome, Edge and Chromium-compatible browsers) that injects a floating panel into the chatbot page to search a prompt, fill its parameters and send it to the editor. Source : `src/extension/`. → Ch. I, Ch. IV

Bulk operations — Multi-selection actions on templates : Ctrl-click, Shift-click, rubber-band selection ; drag-and-drop move ; cut/copy/paste across panes. All bulk edits go through a single `PUT /api/templates/update/bulk` endpoint. → Ch. XIV

Category — Categorization is realized through tags (including hierarchical tags) — there is no separate category field or `/api/templates/categories` route. Where older material says ‘category’, read ‘tag’. → Ch. IX, Ch. X

Collection — Container of templates with POO²L Level 5 semantics : inheritance between containers, virtual templates, same-member shadowing via the collection axis `^parent`. Sidecar metadata `kind="collection"`. Distinguished from a folder. → Ch. XI

Connector — Pluggable module behind a stable interface, loaded at startup according to `izibuddy.ini`. Families : one active database backend (which stores everything), the attachment payload store, AI platforms (extension side), and external services (e-mail, Git mirror, the single alert webhook). → Ch. LIV

Container — The parent grouping of a prompt. Two flavours : folder (pure organization, all POO²L levels) and collection (OO² entity, L5 only). Promotable and demotable from the UI. → Ch. XI

Contextual tip — Non-blocking toast in the bottom-right corner that suggests an optimization based on actual usage (prompt count, frequency of an action). Managed by `src/static/index-tips.js`, declared in `src/static/data/tips.json`. Anti-spam quota : 5 min minimum between tips, 3 per day, never the same tip twice in one session. → Ch. XXXIX

Dashboard — Web app (FastAPI + JS) where templates are authored, organized, shared and versioned. Self-hosted, team-server, or used via a hosted instance. Entry point : `src/main.py`. → Ch. I, Ch. III

Disabled — Marker that hides a prompt from default listings unless the matching filter is on. Useful for archival without deleting. → Ch. VIII

Duplicate — Take a personal copy of a prompt — decoupled from the original. Distinct from *sharing* (read-only, the recipient sees the original). → Ch. VII, Ch. XXIV

Feature — Granular UI element that can be enabled independently. The registry `src/static/index-features-registry.js` enumerates 44 of them, in 9 categories. Managed by `IziFeatures` (runtime) and a panel in the Options page. → Ch. XXXVIII

Filter — Criterion narrowing the prompt list. Cumulative (13 filters in the *Filters* category of the registry). The meta-toggle `filtersExclude` unblocks the ‘exclude’ state of the tristate cycle ; `filterClearAll` resets them. → Ch. XII, Ch. VIII

Final section — Section declared `@@@section final <name>` : sealed. No descendant may redeclare it under any form (no body, toggle, or description change). Error : OVERRIDE_OF_FINAL_SECTION. → Ch. XXI

Floating panel — Small window launched by the extension inside the chatbot's page. Three tabs : Templates (search + injection), Profiles (themes per platform), Config (URL, pairing). → Ch. XXX

Folder — Pure organizational container, file-system-directory equivalent. Available at all POO²L levels (L0–L5). Sidecar metadata `kind="folder"`. No compositional semantics. → Ch. XI

Git applicable — Property of an install whose storage backend is the file-based default — git versioning and the *Git Settings* modal are only meaningful on these installs. → Ch. XXVII

Group — Multi-tenant scope owning its own templates and tags. Two membership levels only : members and admins (no editor/viewer roles). Managed by `src/core/groups_manager.py`, exposed via `src/api/groups.py`. → Ch. XXV

Hidden — Marker that removes a prompt from default views. Distinct from Disabled in that Hidden is a stronger UI-level filter ; both can be combined with the matching filter feature. → Ch. VIII

Important — Marker that pins a prompt as a favorite. Surfaces via the `filterImportant` filter and the star icon in lists. → Ch. VIII

Injection — The act of sending a rendered prompt to the editor of the open chatbot. The extension detects the platform's input area and inserts the text there ; a dialog may prompt for parameters first. → Ch. XXXV, Ch. XXXII

Invitation — Token sent by e-mail that opens access to the instance (admin-issued). The token has an expiry ; it may be revoked. Source : `src/api/invitations.py`, `src/core/email_service.py`. → Ch. XXVI

izibuddy.ini — Dashboard configuration file at the application root. Sections include `[server]`, `[paths]`, `[processing]`, `[logging]`, `[alerts]`, `[registration]`, `[email]` (SMTP), `[git]`, and `[database]` with its `[database.<name>]` subsections. There is no `[storage]` or `[smtp]` section, and no `${VAR}` substitution. → Ch. XLVIII

Locked — Marker preventing edits to a prompt without explicit unlock. Combined with group permissions, it freezes a validated prompt. → Ch. VIII

Marker — Special label carried by a prompt that influences how it shows up in listings. Six markers : Important ⭐, Locked 🔒, WIP, Disabled, Template, Hidden. Each one is paired with a filter in the *Filters* category. → Ch. VIII

Multi-backend — Deployment topology where two or more IziBuddy backends share one source of truth — either a database or a git remote — to keep their views in sync. → Ch. XXVII

Notes — Private editor text, stripped from the final rendering. Used to document a design choice or leave a TODO without bloating the prompt. Enabled by the `editorNotes` feature. → Ch. XXI

Owner — The single user who can edit a prompt. Shared templates are read-only for everyone else ; sharing never grants write access. → Ch. XXIV, Ch. XXVII

Parameter set — Named collection of values ready to instantiate a prompt in one click. Lives at prompt scope when standalone (`@@@parameter-set ...`), at collection scope otherwise. Selectable at injection time. → Ch. XXXV

POO²L — Declarative language (*Template Object-Oriented² Language*) for writing templates that IziBuddy manages. See Part III, which covers the UI surface used to author POO²L from the dashboard. → Ch. I, Ch. XVI, Ch. XXII

Preset — Ready-made feature profile. Four presets : Simple, Standard, Advanced, Expert — each enables a subset of the registry. The user can then tune individual features. → Ch. XXXVII

Pull strategy — Setting of the `[git]` section selecting how `pull` resolves divergence : `ff-only` (refuses merges, recommended), `rebase`, or `merge`. → Ch. XLVIII, Ch. LIX

Purge — Permanently delete a prompt already in the trash. Irreversible. Endpoint `/purge/{id}` in `src/api/templates.py`. → Ch. VII

Reverse proxy — Intermediate server (Nginx, Caddy, Traefik) between the Internet and the IziBuddy dashboard. Terminates TLS, adds `X-Forwarded-*` headers, handles WebSocket. See the reverse-proxy chapter. → Ch. LII

Scoring — A computed quality score out of 10 produced by the analysis engine (not a user star-rating). Displayed when the `viewScoring` feature is on. The `filterNoScoring` filter targets templates flagged to opt out of scoring. → Ch. XXIX

Section — Named, delimited fragment of a prompt body. Two orthogonal axes : presence toggle (`?` opt-in / `!` opt-out / none, glued to the `section` keyword) and inheritance modifier (`virtual` abstract / `final` sealed / none, placed after the keyword). There is no section numbering. General shape : `@@@section[?|!] [virtual|final] <name> [: <description>]`. Unit of override under inheritance. → Ch. XXI

Section toggle — Optional suffix on the `section` keyword that turns the section into a user choice at instantiation. `?` defaults the include/exclude dialog to OFF (opt-in) ; `!` defaults it to ON (opt-out). Both require a description — it is the text shown to the user in the dialog. Purely a presence toggle ; does **not** affect scoring or weight. → Ch. XXI

Sharing — Grant read-only access to a prompt to another user or group. Distinct from *duplication* (independent copy). → Ch. XXIV

Smart editor — Dashboard editing pane with POO²L syntax highlighting, autocomplete (parameters, constants, ids), inline validation (red/yellow squiggles). Source : `src/static/index-smart-editor.js`. → Ch. XVI

Split pane — Two-panel layout : edit a prompt on the right while browsing the list on the left, or edit two templates in parallel. `Tab` switches the active pane ; `F5` copies the selection to the other pane and `F6` moves it. Enabled by the `viewSplitPane` feature. → Ch. XIII, Ch. XIV

Stash — A set of templates kept handy in the extension. Not extension-local storage : stashed templates are backend templates tagged `STASH`. → Ch. XXXV

Synchronised remote — A git remote that two file-backed IziBuddy backends both push to and pull from, acting as the shared source of truth across machines (`sync_on_startup`, `sync_on_read`). → Ch. XLVIII, Ch. LIX

System variable — Runtime value substituted automatically into a prompt via the `@@@=NAME` form : `@@@=DATE`, `@@@=TIME`, `@@@=DATETIME`, `@@@=YEAR`, `@@@=MONTH`, `@@@=DAY`, `@@@=WEEKDAY`, `@@@=USER`, `@@@=LANG`, `@@@=GROUPS`, `@@@=GROUPS_ADMIN`. There is no `{{NAME}}` form. → Ch. XLI, Ch. XXII

Tag — Multi-value label attached to a prompt. Flat or hierarchical tags (feature `tagsHierarchy`). Clicking a tag filters the list (`clickTagToFilter`). Admin : merge, global rename, and delete — deletion is refused while the tag is still in use (`tagsAdmin`). → Ch. X

Template — The unit of content in IziBuddy. Visible fields : name, description, content, tags, container. Hidden state : markers, status, version, author, group ownership, scores. The POO²L source is distinct from the rendered text sent to the AI. → Ch. VI

Template — Marker tagging a prompt as a base-class hint — typically a parent prompt that other templates inherit from. Surfaced through the `filterExample` filter. → Ch. VIII

Theme — Visual profile of the dashboard and/or extension. Light/dark, plus per-platform custom themes (Profiles tab in the extension). Derived from the `--izi-*` design tokens defined in `src/static/styles/base.css`. → Ch. XXXVI, Ch. XXXIII

Trash / Soft delete — Move a prompt to the trash : it drops out of default listings but remains restorable. Endpoints `/trash/list`, `/trash/count`, `/restore/{id}`. Purge is permanent deletion. → Ch. VII

UI level — Old 3-level setting (Simple/Standard/Advanced) that drove UI complexity. Replaced by the system of 44 features and 4 presets. Automatic migration on first load. → Ch. XXXVII

Version — Frozen state of a prompt at one point in time. Any change is versioned (deduplicated by checksum). Read via `GET /api/versions/{id}` ; compared via `POST /api/diff/compare` (side-by-side, Line/Word toggle) ; restored by copying an old version back with `POST /api/versions/{id}/copy`. → Ch. XXVIII

Virtual section — Section declared `@@@section virtual <name>` : abstract, no body. Every concrete descendant in the inheritance chain must override it with a body. Errors : VIRTUAL_SECTION_HAS_BODY, UNIMPLEMENTED_VIR-
TUAL_SECTION, INCLUDE_OF_VIRTUAL_SECTION. → Ch. XXI

Webhook — The single outbound alert webhook : a best-effort HTTP POST in Slack-attachment format, sent when the alert system fires and `webhook` is in `[alerts] channels`, to the one `[alerts] webhook_url`. No `[webhooks]` section, no per-event routing, no HMAC signing, no retries. → Ch. LX

WIP — Marker tagging a prompt as Work In Progress — not ready for general use. Surfaced through the `filterWip` filter. → Ch. VIII

Index thématique

Administration

Admin dashboard → Ch. XLII
Alerts → Ch. XLVI
Analytics → Ch. XLV
Audit → Ch. XLVI
Diagnostics → Ch. XLVII
License → Ch. XLIV
User management → Ch. XLIII

AI platforms

ChatGPT → Ch. XXXIV, Ch. LVII
Claude → Ch. XXXIV, Ch. LVII
DeepSeek → Ch. XXXIV, Ch. LVII
Gemini → Ch. XXXIV, Ch. LVII
Grok → Ch. XXXIV, Ch. LVII

Concepts

Architecture (overview) → Ch. I, Ch. II
Deployment → Ch. III, Ch. LII
Hosting (self-hosted vs cloud) → Ch. III
i18n (French / English) → Ch. XL
Lifecycle of a prompt → Ch. VII, Ch. VIII
Mental model of IziBuddy → Ch. II
Multi-tenant → Ch. XXV, Ch. XXVII
Overview → Ch. I
Permissions and roles → Ch. XXV

Privacy → Ch. XLV
Quick start (5 minutes) → Ch. V
Rendering a prompt → Ch. XXII, Ch. XLI
Security (CSP, cross-origin) → Ch. IV, Ch. LII
Sharing vs duplicating → Ch. XXIV

Connectors

Connector — AI platforms → Ch. LVII
Connector — databases → Ch. LV
Connector — Git remote → Ch. LIX
Connector — object storage → Ch. LVI
Connector — SMTP / e-mail → Ch. LVIII
Connector — webhooks → Ch. LX
Filesystem (DB + storage) → Ch. LV, Ch. LVI
PostgreSQL → Ch. LV
S3 (AWS) → Ch. LVI

Contextual tips

cut-paste-burst-split → Ch. XXXIX
duplicate-burst-pool2l → Ch. XXXIX
expert-discovery → Ch. XXXIX
first-deletion-undo → Ch. XXXIX
first-version-saved → Ch. XXXIX
no-tags-30-templates → Ch. XXXIX
poolevel-max-reminder → Ch. XXXIX
promote-collections-3-folders → Ch. XXXIX
promote-folders-50 → Ch. XXXIX
simple-preset-many-templates → Ch. XXXIX
standard-using-collections → Ch. XXXIX

Domain objects

Category → Ch. IX
Collection → Ch. XI
Container → Ch. XI
Folder → Ch. XI
Group → Ch. XXV
Tag → Ch. X
Template → Ch. VI
Version → Ch. XXVIII

Filters

filterClearAll → Ch. XII
filterCollection → Ch. XII
filterDisabled → Ch. XII, Ch. VIII
filterExample → Ch. XII, Ch. VIII
filterFolder → Ch. XII
filterHidden → Ch. XII, Ch. VIII
filterImportant → Ch. XII, Ch. VIII
filterLevel → Ch. XII
filterLocked → Ch. XII, Ch. VIII
filterNoScoring → Ch. XII, Ch. XXIX
filtersExclude → Ch. XII
filterTemplate → Ch. XII
filterWip → Ch. XII, Ch. VIII

Keyboard shortcuts

Ctrl+C / Ctrl+V (copy/paste templates) → Ch. XIV, Ch. XL
Ctrl+Click (multi-select) → Ch. XIV, Ch. XL
Ctrl+X / Ctrl+V (cut/paste) → Ch. XIV, Ch. XL
Ctrl+Z / Ctrl+Y (undo/redo) → Ch. XIV, Ch. XL
F5 / F6 (cross-pane) → Ch. XIII, Ch. XIV, Ch. XL
Shift+Click (range select) → Ch. XIV, Ch. XL

Markers

Disabled → Ch. VIII
Hidden → Ch. VIII
Important (⭐) → Ch. VIII
Locked (🔒) → Ch. VIII
Template → Ch. VIII
WIP → Ch. VIII

Operations

Backups → Ch. L
izibuddy.ini → Ch. XLVIII
Logging → Ch. XLIX
REST API → Ch. LI
Reverse proxy → Ch. LII
Upgrading → Ch. LIII

REST API endpoints

/api/auth/users → Ch. XLIII, Ch. LI
/api/groups → Ch. XXV, Ch. LI
/api/invitations → Ch. XXVI, Ch. LI
/api/tags → Ch. X, Ch. LI
/api/templates (CRUD) → Ch. VII, Ch. LI
/api/templates/update/bulk → Ch. XIV, Ch. LI
/api/versions → Ch. XXVIII, Ch. LI

System variables

@@@=DATE → Ch. XLI
@@@=DATETIME → Ch. XLI
@@@=DAY → Ch. XLI
@@@=GROUPS → Ch. XLI

@@@=GROUPS_ADMIN → Ch. XLI
@@@=LANG → Ch. XLI
@@@=MONTH → Ch. XLI
@@@=TIME → Ch. XLI
@@@=USER → Ch. XLI
@@@=WEEKDAY → Ch. XLI
@@@=YEAR → Ch. XLI

UI and features

Container colors → Ch. XI
Contextual tips → Ch. XXXIX
Features panel → Ch. XXXVIII
Filters → Ch. XII, Ch. VIII
Floating panel (extension) → Ch. XXX
Notes (private) → Ch. XXI
Presets (Simple / Standard / Advanced / Expert) → Ch. XXXVII
Search → Ch. XII
Smart editor → Ch. XVI
Split pane (shared view) → Ch. XIII
Stash (extension) → Ch. XXXV
Themes → Ch. XXXVI, Ch. XXXIII
Views (grid / details) → Ch. XIII

Index alphabétique

/

/api/auth/users → Ch. XLIII, Ch. LI
/api/groups → Ch. XXV, Ch. LI
/api/invitations → Ch. XXVI, Ch. LI
/api/tags → Ch. X, Ch. LI
/api/templates (CRUD) → Ch. VII, Ch. LI
/api/templates/update/bulk → Ch. XIV, Ch. LI
/api/versions → Ch. XXVIII, Ch. LI

@

@@@=DATE → Ch. XLI
@@@=DATETIME → Ch. XLI
@@@=DAY → Ch. XLI
@@@=GROUPS → Ch. XLI
@@@=GROUPS_ADMIN → Ch. XLI
@@@=LANG → Ch. XLI
@@@=MONTH → Ch. XLI
@@@=TIME → Ch. XLI
@@@=USER → Ch. XLI
@@@=WEEKDAY → Ch. XLI
@@@=YEAR → Ch. XLI

A

Admin dashboard → Ch. XLII
Alerts → Ch. XLVI
Analytics → Ch. XLV
Architecture (overview) → Ch. I, Ch. II
Audit → Ch. XLVI

B

Backups → Ch. L

C

Category → Ch. IX
ChatGPT → Ch. XXXIV, Ch. LVII
Claude → Ch. XXXIV, Ch. LVII
Collection → Ch. XI
Connector — AI platforms → Ch. LVII
Connector — databases → Ch. LV
Connector — Git remote → Ch. LIX
Connector — object storage → Ch. LVI
Connector — SMTP / e-mail → Ch. LVIII
Connector — webhooks → Ch. LX
Container → Ch. XI
Container colors → Ch. XI
Contextual tips → Ch. XXXIX
Ctrl+C / Ctrl+V (copy/paste templates) → Ch. XIV, Ch. XL
Ctrl+Click (multi-select) → Ch. XIV, Ch. XL
Ctrl+X / Ctrl+V (cut/paste) → Ch. XIV, Ch. XL
Ctrl+Z / Ctrl+Y (undo/redo) → Ch. XIV, Ch. XL
cut-paste-burst-split → Ch. XXXIX

D

DeepSeek → Ch. XXXIV, Ch. LVII
Deployment → Ch. III, Ch. LII
Diagnostics → Ch. XLVII
Disabled → Ch. VIII
duplicate-burst-pool2l → Ch. XXXIX

E

expert-discovery → Ch. XXXIX

F

F5 / F6 (cross-pane) → Ch. XIII, Ch. XIV, Ch. XL
Features panel → Ch. XXXVIII
Filesystem (DB + storage) → Ch. LV, Ch. LVI
filterClearAll → Ch. XII
filterCollection → Ch. XII
filterDisabled → Ch. XII, Ch. VIII

filterExample → Ch. XII, Ch. VIII
filterFolder → Ch. XII
filterHidden → Ch. XII, Ch. VIII
filterImportant → Ch. XII, Ch. VIII
filterLevel → Ch. XII
filterLocked → Ch. XII, Ch. VIII
filterNoScoring → Ch. XII, Ch. XXIX
Filters → Ch. XII, Ch. VIII
filtersExclude → Ch. XII
filterTemplate → Ch. XII
filterWip → Ch. XII, Ch. VIII
first-deletion-undo → Ch. XXXIX
first-version-saved → Ch. XXXIX
Floating panel (extension) → Ch. XXX
Folder → Ch. XI

G

Gemini → Ch. XXXIV, Ch. LVII
Grok → Ch. XXXIV, Ch. LVII
Group → Ch. XXV

H

Hidden → Ch. VIII
Hosting (self-hosted vs cloud) → Ch. III

I

i18n (French / English) → Ch. XL
Important (⭐) → Ch. VIII
izibuddy.ini → Ch. XLVIII

L

License → Ch. XLIV
Lifecycle of a prompt → Ch. VII, Ch. VIII
Locked (🔒) → Ch. VIII
Logging → Ch. XLIX

M

Mental model of IziBuddy → Ch. II
Multi-tenant → Ch. XXV, Ch. XXVII

N

no-tags-30-templates → Ch. XXXIX
Notes (private) → Ch. XXI

O

Overview → Ch. I

P

Permissions and roles → Ch. XXV
poolevel-max-reminder → Ch. XXXIX
PostgreSQL → Ch. LV
Presets (Simple / Standard / Advanced / Expert) → Ch. XXXVII
Privacy → Ch. XLV
promote-collections-3-folders → Ch. XXXIX
promote-folders-50 → Ch. XXXIX

Q

Quick start (5 minutes) → Ch. V

R

Rendering a prompt → Ch. XXII, Ch. XLI
REST API → Ch. LI
Reverse proxy → Ch. LII

S

S3 (AWS) → Ch. LVI
Search → Ch. XII
Security (CSP, cross-origin) → Ch. IV, Ch. LII
Sharing vs duplicating → Ch. XXIV
Shift+Click (range select) → Ch. XIV, Ch. XL

simple-preset-many-templates → Ch. XXXIX
Smart editor → Ch. XVI
Split pane (shared view) → Ch. XIII
standard-using-collections → Ch. XXXIX
Stash (extension) → Ch. XXXV

T

Tag → Ch. X
Template → Ch. VI
Template → Ch. VIII
Themes → Ch. XXXVI, Ch. XXXIII

U

Upgrading → Ch. LIII
User management → Ch. XLIII

V

Version → Ch. XXVIII
Views (grid / details) → Ch. XIII

W

WIP → Ch. VIII
