Kit de documentation

Project — Common Structure — EN

1 Introduction

This document is a translation in substance. The reference is the original document in French; extensions and changes are always made there.

This document defines the mandatory structure of every project that uses the Kit de documentation. It is authoritative for the files a project must hold, for the normative form of each data file, and for the lifecycle rules governing their evolution.

Two situations fall under the same contract: setting up a new project, and bringing an existing project back into conformity after a Kit update. In both cases the *project files* are created or regenerated from the skeletons given below.

The audience is the active Kit user managing one or more projects. The *AI* reads this document at the start of a project session to check conformity before any *generation*.

2 Inventory of project files

Normative table of the files an active project must hold. The Flag column gives the name of the switch in [Prefix] - Registry.requires; the Default column gives the value used when a project is set up. A file without a flag is mandatory without exception.

Project file	Role	Flag	Default
[Prefix] - Projet - *Prompt* (TS).docx	Technical instructions specific to the project	—	Mandatory
[Prefix] - *Registry.js*	Project source of truth — versions, flags, timestamps	—	Mandatory
[Prefix] - Settings.js	Project-owned settings — outside the Kit perimeter	—	If needed
[Prefix] - Projet - Todos (TS).docx	Open entries, debts and dated register of decisions	—	Mandatory
[Prefix] - Glossary - Terms (TS).js	Glossary module — source of the glossary and of *pipeline* pass 2	glossary	true
[Prefix] - Glossaire - Termes - LANG (TS).docx	Project glossary — couplet with Terms.js. One document per published language if the glossary is translated. Naming: Naming Convention §2.1 and §2.4	glossary	true

Project file	Role	Flag	Default
[Prefix] - Brands (TS).js	Brands module — *pipeline* pass 1	brands	false
[Prefix] - Text Highlights (TS).js	Highlighted fragments — *pipeline*, between brands and glossary	textHighlights	false
[Prefix] - Variable Friendly Names (TS).js	Variables module — *pipeline* pass 3	variableNames	false
[Prefix] - HASS Entities (TS).js	HASS entities module — *pipeline* pass 4	hassEntities	false
[Prefix] - Colors (TS).js	Project colours — replace those of the Kit, §6.5	colors	false
[Prefix] - Document Titles (TS).js	Display labels of documents by language, §6.8	documentTitles	false
[Prefix] - Documentation - *Cover Sheet* (TS).docx	*PCL* constants — couplet with the PNG	coverSheet	true
[Prefix] - Documentation - *Cover Sheet* (TS).png	*Cover page* of the *paper binder*	coverSheet	true
[Prefix] - Documentation - Reading Guide (TS).docx	Reading guide of the *paper binder*	readingGuide	true
[Prefix] - Documentation - *Pipeline* HTML (TS).docx	Specification of the project HTML publication	—	Mandatory

Note: *A file whose flag is false is absent from the project. The **Node.js** generators read Registry.requires at start-up and skip the matching require() calls and setters — see §6.6. Every file carries a timestamp in its name, with the single exception of [Prefix] - Registry.js, which is itself the source of the timestamps. A document published in several languages also carries a language tag before its timestamp — Naming Convention §2.4.*

3 Shared Kit dependencies

The Kit stylesheets — General, Glossary, YAML, HTML — are technical libraries shared by every project. They are never copied into a project. Project generators reference them directly with require() at their Kit location.

The file Kit - *Registry.js* is the single source of the active *stylesheet* versions. Projects read it to detect a Kit update and to know whether a project couplet must be regenerated. No project stores a *stylesheet* version locally — at *generation* time the truth is always in Kit - *Registry.js*.

The Kit also has its own glossary, Kit - Glossaire.docx, distinct from the project glossary. A project carries its own terms, in its own terms module: the generators load the project's, never the Kit's.

A third group may appear in the working directory: files prefixed Kit-X. These are the extensions written by the Kit user — additional stylesheets, a colour palette, the *registry* of those stylesheets. They are neither Kit files, replaced at every update, nor *project files*, belonging to one project: they belong to their author and follow them from project to project.

- **A project reads them without owning them:** a generator can import a Kit-X *stylesheet* as it imports a Kit one. The project *Registry* does not declare their versions — those live in Kit-X - *Registry.js*. Naming: Naming Convention §3.3. How to write one: Extending the Kit.

4 Project Prompt

The file [Prefix] - Projet - *Prompt* (TS).docx holds the technical instructions specific to the project. It completes the *Prompt* — it never duplicates it. The *AI* reads it at the start of every working session on the project.

Values to customise: the project prefix, project.documentAuthor and project.webAuthor (the names shown in the footer of the document and of the page), project.docLanguage (FR, EN, DE or LU — default EN for projects, single source of truth in the project section of [Prefix] - *Registry.js*), the editorial conventions of the project's field, and any narrative rules of its own.

The technical rules common to every project — formatting, session protocols, naming convention, validation chain — live in the *Prompt* and are not reproduced in the project *prompt*. The project *prompt* holds only what differs from one project to another.

5 Project Registry

Every project has its own [Prefix] - *Registry.js*. It is the source of truth for identity, the flags of required files, family domains, the timestamps of generated documents, rendering flags, the web compilation notice, and the deployment metadata of the HTML site.

5.1 Role and content

The project *Registry* holds the mandatory sections below and one optional section. Each has its own update rule — see §5.3.

- **project:** identity and language of the project. The field project.docLanguage (FR, EN, DE, LU) is the single source of truth for language, read by the stylesheets for *L10N* selection and by the *Cover Sheet renderer* for the localised *PCL* strings. projectShareable, false by default, says whether the project is offered for reuse: at true the publication pass produces the project archive and the download icon appears. allowNonKitTemplates, false by default, opens the right to create *templates outside the Kit*'s forms — Extending the Kit §9. Three boolean flags join it as well, false by default: autoAddGlossary, autoAddTextHighlights and autoAddBrands. At false, the *AI* reports a candidate and waits for confirmation; at true, it adds it to the module concerned and reports it at the end of the section — *Kit Prompt* §13.2.
- **requires:** flags for the presence of optional files — coverSheet, readingGuide, glossary, textHighlights, documentTitles, brands, variableNames, hassEntities, colors. Read by the generators to condition require() calls and *setters*.
- **familyDomains:** a list of hostnames without scheme. Family domains whose links, and those of their sub-domains, stay captured in the *Android* app *WebView*. Any other domain is routed to the system browser. Read by style.setFamilyDomains() at the head of gen-[project]-site.js. A stable section — changed only when the family perimeter changes.
- **documents:** mapping from slug to *timestamp* for each generated document of the project. Updated at every document delivery — the *timestamp* reflects the last regeneration. A document published in several languages has one entry per variant, its slug carrying the language suffix: philosophy-fr, philosophy-de. Incremental publishing compares by slug, so each language republishes independently. The optional key piece names the file of an appendix piece: it lives at the root of the project, the pass copies it under the site's assets/pieces/, under its original name, and the page's download button serves it in place of the document's PDF — HTML *Pipeline* §6.3. Two optional keys join them, written by the site generator: contentTs, the *timestamp* of the last change of text, and contentHash, the *fingerprint* of the text that produced it. A *Registry* without them works; the generator sets them on the first pass.
- **rendering:** flags driving the *generation* of outputs. The sub-sections coverSheet.sectionHMode (FIXED or DYNAMIC), consumed by kit_render_cover_sheet.py, and toc.generate / toc.hidden, driving the

side HTML table of contents through renderDocument and getCSS. The section is optional — in its absence the *renderer* falls back to FIXED. Declaring it explicitly stays recommended, so that the choice is legible. languageSelector is set here too — §15 —, and indexIcon declares the landing illustration, the project's fourth icon beside favicon.svg, favicon.ico and apple-touch-icon.png, all four at its root. homeHref gives the target of the home icon on the *landing page* alone, which leads out of the sub-site; the home icon of a document page always returns to the index of its language, with no setting — along with indexIcon: a project that has placed index-icon.svg at its root declares it here, and the *landing page* shows the image to the left of the subtitle. Absent, nothing changes.

- **compiledWith:** compilation notice shown in the footer of the sub-site pages. An object indexed by language code, passed to the HTML *stylesheet* by the site generator through config.compiledWith. An absent block, null, or an empty string for a language: no notice is rendered. This field drives the web only — no.docx produced by the Kit carries a compilation notice, the General *stylesheet* having removed the matching *L10N* key.
- **deploy:** metadata of the HTML sub-site — typically lastDeploy. Written by gen-[project]-site.js during a full or incremental deployment.
- **stylesheets:** OPTIONAL. Versions and timestamps of the stylesheets written by the project, on the *model* of the section of the same name in Kit - *Registry.js*. Absent if the project writes none. It lists only the project's stylesheets: those of the Kit stay declared in the Kit *Registry*, the single source of their versions — §3. How to write one: Extending the Kit.

5.2 Normative structure

Skeleton to instantiate when setting up a project. The comments recall the meaning of each section. The file carries no *timestamp* in its name — it is itself the source of the timestamps.

```
'use strict';
// ==============================
// [Préfixe] - Registry.js — Source de vérité du projet
//
// Versions stylesheet Kit utilisées : lues depuis Kit - Registry.js.
// Ce fichier projet ne duplique pas les versions Kit.
// ==============================

module.exports = {

  // ---- Identité et langue ---------------------------------
  // project.docLanguage pilote la sélection L10N des stylesheets
  // General/Glossary/HTML et les constantes PCL du Cover Sheet.
```

```
  project: {
    docLanguage: 'EN',  // 'EN' (défaut consommateurs) | 'FR' | 'DE' | 'LU'
    documentAuthor:   'Prénom Nom',          // pied de page des .docx et PDF
    webAuthor:      'Prénom Nom',        // pied des pages web — '' : site non
signé
    documentSiteBase:  'https://[sous-site]',  //  racine  des  renvois  entre
documents
    projectShareable:      false,  // projet offert en reprise
    allowNonKitTemplates:  false,  // Étendre le Kit §9
    autoAddGlossary:       false,  // Kit Prompt §13.2
    autoAddTextHighlights: false,
    autoAddBrands:         false,
  },

  // ---- Fichiers requis — flags ----------------------------
  // Règle : si flag = false, le fichier correspondant est absent
  // du projet et le générateur saute le require() + setter().
  requires: {
    coverSheet:    true,   // défaut true
    readingGuide:  true,   // défaut true
    glossary:      true,   // défaut true
    textHighlights: false, // défaut false
    documentTitles: false, // défaut false
    brands:        false,  // défaut false
    variableNames: false,  // défaut false
    hassEntities:  false,  // défaut false
    colors:        false,  // défaut false
  },

  // ---- Domaines famille — routage hyperliens HTML ---------
  // Liens vers ces domaines (et sous-domaines) restent dans la
  // WebView de l'app Android famille. Liens externes ouvrent
  // dans le navigateur via target="_blank" rel="noopener".
  // Lu par style.setFamilyDomains() en tête de gen-[projet]-site.js.
  familyDomains: ['sliver.lu', 'hexi.lu'],

  // ---- Documents projet — timestamp dernière génération ---
  documents: {
    // 'slug-document': { ts: 'AAAA-MM-JJ - HHhMM' },
  },

  // ---- Rendering — flags pilotant la génération -----------
  // Lus par les renderers avec fallback gracieux si la section
  // est absente — voir Cover Sheet §3 ligne SECTION_H_MODE.
  rendering: {
    homeHref: null,           // maison de la page d'accueil seule — Structure
commune §15
    languageSelector: 'document', // 'document' (défaut) | 'site' — Structure
commune §15
    indexIcon: false,          // true : index-icon.svg s'affiche sur la page
d'accueil
    coverSheet: {
      sectionHMode: 'FIXED',  // FIXED (défaut) | DYNAMIC
    },
    toc: {
      generate: true,    // false -> bloc <nav class="toc"> absent du HTML
      hidden:   false,   // true  -> CSS .toc { display: none; } injecté
```

```
    },
  },

  // ---- Mention de compilation — pied de page HTML ---------
  // Rendue dans le pied de page des pages du sous-site, transmise
  // par gen-[projet]-site.js via config.compiledWith.
  // Objet indexé par code langue. null, bloc absent ou chaîne vide
  // pour une langue : aucune mention rendue.
  // Ne concerne JAMAIS les .docx — voir §5.1.
  compiledWith: null,
  // Exemple si le projet souhaite l'afficher :
  //   compiledWith: {
  //     FR: 'Compilé avec Claude AI',
  //     EN: 'Compiled with Claude AI',
  //     DE: 'Kompiliert mit Claude AI',
  //     LU: 'Kompiléiert mat Claude AI',
  //   },


  // ---- Archive de telechargement --------------------------
  // Present si le projet publie son archive ; absent sinon.
  projectZip: {
    version:  '0.01',            // monte avec la structure ou les regles
    filename: '[sous-site]-0.01.zip',
    updated:  'AAAA-MM-JJ',      // suit le numero de version
  },
   // ---- Déploiement site HTML ------------------------------
  deploy: {
    siteName: '[sous-site]',       // nomme l'archive et les pages d'index
    siteInfrastructure: 'parent',  // icônes, robots.txt et .htaccess du site
parent
    lastDeploy: null,  // ISO 'AAAA-MM-JJThh:mm:ss' — écrit par le générateur
  },

};
```

Note: *The skeleton was widened to include the project, familyDomains, rendering and compiledWith sections — historically absent although the stylesheets and the Cover Sheet renderer read them. A project conforming to the old skeleton could crash kit_render_cover_sheet.py with a KeyError. Protection is now twofold: a complete skeleton here, and a graceful fallback on the renderer side.*

5.3 Lifecycle

The project section is stable — changed only if the project changes its main language, a rare and explicit case documented in the Update Guide §6.

The requires section is stable over the life of the project — changed only when a functional dependency is added or removed. A project that starts documenting *Home Assistant* switches hassEntities from false to true and creates the matching module.

The familyDomains section is stable — changed only when the family perimeter changes.

The documents section is updated at every document delivery — the document's generator, or the user on receipt, writes in the new *timestamp*.

The rendering section is stable — changed to switch the *Cover Sheet* between FIXED and DYNAMIC, a rare case, or to toggle the HTML table of contents. The user never edits *Registry.js* by hand: they ask the *AI* in the project concerned, which performs the switch.

The compiledWith section is stable — it is an editorial decision of the project, not a technical state. Changing it forces no document regeneration: the value is read when the site is generated.

The deploy.lastDeploy section is written exclusively by gen-[project]-site.js during a deployment. No other script, and not the user, edits it by hand.

5.4 Project-owned settings — Settings.js

The project *Registry* carries only the blocks the Kit specifies: project, requires, stylesheets, familyDomains, documents, rendering, compiledWith, projectZip, deploy. A project fills in values there; it adds neither key nor block. What belongs to it lives in a separate file, [Prefix] - Settings.js, which the Kit neither specifies nor reads.

- **A file boundary, not a matter of discipline.** A block of one's own lodged in the *Registry* forces a case-by-case judgement on whether its place is legitimate. In a separate file, a key unknown to the *Registry* is wrong by construction, and a check can say so without arbitrating.
- **Documented in the project Prompt.** Every heading of Settings.js carries there its name, its keys, the effect of each, and the effect of its absence. An undocumented key is a key lost by the next session.
- **Read with a loud stop.** A generator reading a setting fails, naming it, if it is missing or has the wrong type, rather than falling back on a silent default.

Note: *kit_check_registry.py rejects any first-level key absent from the Kit's list. It runs before any generation — Quality Control §3. The separation holds only if something verifies it: without a check it would be a second convention, as optional as the first.*

Note: *A Kit update touches neither the project Registry nor its Settings.js — Update Guide. No mechanism protects them: it is the procedure that does not replace them, which is why it enumerates what it does replace.*

Note: *Kit-X extensions are something else: they belong to their author, follow them from project to project, and declare their versions in Kit-X - Registry.js — §3.*

6 Data modules

A project may hold *data modules* — each conditionally enabled by a flag in Registry.requires. The structural rules of each module are normative and not open to interpretation. Any divergence between an existing file and the structure below is an anomaly to fix before any *generation*.

This section is the single home of the data-module contract. The other Kit documents refer to it without restating it.

6.1 Glossary Terms

File: [Prefix] - Glossary - Terms (TS).js. Flag Registry.requires.glossary, default true. Couplet with the glossary document or documents — all members share the same *timestamp* at every regeneration. Naming of the documents: Naming Convention §2.1 and §2.4.

The module is loaded at the head of every generator and feeds pass 2 of the *pipeline* — terms are rendered automatically in teal italics throughout the running text.

Mandatory fields per entry: each element of glossaryEntries carries four fields, three of which are required.

Field	Type	Required	Description
balise	string	Yes	Stable HTML *anchor* — _slugify of the term in its original language, unique within the file. Language-independent and never changed once published: it identifies the concept, not the word
term	string or object	Yes	Canonical term. A string if the glossary is monolingual, an object indexed by language code otherwise
aliases	array or object	No	Alternative forms pointing at the same tag. An array if monolingual, an object indexed by language code otherwise

Field	Type	Required	Description
definition	string or object	Yes	Definition in plain language, never *Markdown*. A string if monolingual, an object indexed by language code otherwise

A project publishing its glossary in several languages indexes term, aliases and definition by language code. The tag itself never changes: it is derived from the term in its original language and serves as a public *anchor*. That is what lets the language selector move from one page to another while keeping the reader's position — the German variant of a document points at the same *anchor* as the French one.

The two forms coexist without migration. A string means monolingual, an object means multilingual. A project switches term by term, at the moment it translates.

- **Sorting by language:** the alphabetical order of entries is computed on the term in the rendered language. Two languages' glossaries therefore do not share the same order, which is correct for a glossary.
- **Missing field — loud failure:** if a published language has no value for term or definition, the generator stops and names the offending tag. No silent fallback on the *Registry* language: a glossary truncated without warning is exactly the kind of damage the Kit fights.
- **Translated aliases:** the aliases of a language feed the *pipeline*'s detection in the documents of that language. Without German aliases, no term turns teal in a German document — the *pipeline* falls silent exactly where it is expected.
- **Glossary only:** the optional field detection: false keeps the entry in the glossary, on paper and on the site, without feeding the detection passes — for a common word one wants to define without colouring it everywhere. buildSearchTerms leaves it out.
- **Hyphenated compounds:** a compound whose second part starts with a lower-case letter — zone-based, Tasmota-flashed — is recognised only if declared as an alias of the term. Otherwise the boundary treats it as a file or package name, which it protects: python-docx. A second part with a capital stays a word boundary — HTML-*Stylesheet*.

Mandatory exports: glossaryEntries, an array of objects, and buildSearchTerms(LANG), a function returning the surface and *anchor* pairs sorted longest-first for the requested language. A monolingual glossary may keep the historical export glossarySearchTerms.

Skeleton to instantiate at set-up, or when absorbing a Kit rule that widens the contract.

```
'use strict';
// ==============================
// [Préfixe] - Glossary - Terms (TS).js
// Source de vérité du glossaire projet.
// Couplet avec [Préfixe] - Glossaire (TS).docx — même timestamp.
// Projet multilingue : un Glossaire par langue publiée, tous au
// même timestamp que ce fichier.
// ==============================

const glossaryEntries = [

  // --- Forme monolingue : term, aliases et definition sont
  //     des chaînes. Forme historique, toujours valide.
  // {
  //   term:       'Terme canonique',
  //   aliases:    ['alias 1', 'alias 2'],
  //   balise:     'terme-canonique',       // _slugify(term), unique
  //   definition: 'Définition en langage courant, sans Markdown.',
  // },

  // --- Forme multilingue : term, aliases et definition sont
  //     indexés par code langue. La balise reste hors langue.
  // {
  //   balise: 'hallucination',             // ancre publique, invariante
  //   term: {
  //     FR: 'Hallucination',
  //     DE: 'Halluzination',
  //   },
  //   aliases: {
  //     FR: [],
  //     DE: ['Konfabulation'],
  //   },
  //   definition: {
  //     FR: 'Phénomène par lequel un modèle génère...',
  //     DE: 'Phänomen, bei dem ein Modell...',
  //   },
  // },

];

// Construction longest-first de la liste de recherche.
// LANG est la langue du document en cours de génération.
// Une chaine et un tableau valent pour toutes les langues — forme
// monolingue, Structure commune §6.1. Sans ce test, v[LANG] rend
// undefined sur un tableau d'alias et les alias disparaissent sans
// message. 2026-09-18.
const pick = (v, LANG) =>
  (v === undefined || v === null) ? undefined
  : (typeof v === 'string' || Array.isArray(v)) ? v
  : v[LANG];

function buildSearchTerms(LANG) {
  const out = [];
  glossaryEntries.forEach(e => {
```

```
    if (e.detection === false) return;  // terme de glossaire seul
    const term = pick(e.term, LANG);
    if (term === undefined) {
      throw new Error(`Glossaire : terme absent en ${LANG} pour la balise $
{e.balise}`);
    }
    out.push({ surface: term, anchor: e.balise });
    const al = e.aliases ? pick(e.aliases, LANG) : [];
    (al || []).forEach(a => out.push({ surface: a, anchor: e.balise }));
  });
  out.sort((a, b) => b.surface.length - a.surface.length);
  return out;
}

module.exports = { glossaryEntries, buildSearchTerms };
```

- **glossaryRubriques.** The headings of the glossary: number, title and lead-in per language. Each term's rubrique field points at them. A heading title is glossary content, not a rendering decision — writing it into a generator would put it out of the author's reach and let paper and web diverge.
- **glossaryDocument.** The identity of the document and its framing texts: project title, author, authoritative language, file-name segments per project language, and per rendered language the lead-in, the opening note, the introduction, the closing note and the translation note. The artefact producing the glossary therefore knows neither the project name nor its language: it reads them here.

Note: *The nomBase field follows Naming Convention §2.4: category and subject stay in the project language, only the final tag marks the language of the content. An English-language project produces “AI - Glossary - Terms - EN”, not “AI - Glossaire - Termes - EN”.*

6.2 Brands

File: [Prefix] - Brands (TS).js. Flag Registry.requires.brands, default false. File absent if the project documents no brands.

Mandatory export: brandEntries, a flat array of brand names. Rendered small caps bold through pass 1 of the *pipeline* throughout the running text. Recommended order: longest-first, to avoid overlaps between nested brands.

```
'use strict';
// ==============================
// [Préfixe] - Brands (TS).js
// Noms de marques du projet — rendus small caps bold (passe 1).
// ==============================

const brandEntries = [
  // 'Nom de marque 1',
  // 'Nom de marque 2',
];
```

```
module.exports = { brandEntries };
```

6.3 Variable Friendly Names

File: [Prefix] - Variable Friendly Names (TS).js. Flag Registry.requires.variableNames, default false. File absent if the project documents no technical variables.

Mandatory export: variableNameEntries. Names of variables and technical parameters. Rendered in dark green italics, VARIABLE_NAME_COLOR, through pass 3 of the *pipeline*.

```
'use strict';
// ==============================
// [Préfixe] - Variable Friendly Names (TS).js
// Noms de variables techniques — italique dark green (passe 3).
// ==============================

const variableNameEntries = [
  // 'MA_VARIABLE',
  // 'autreNomVariable',
];

module.exports = { variableNameEntries };
```

6.4 HASS Entities

File: [Prefix] - HASS Entities (TS).js. Flag Registry.requires.hassEntities, default false. File absent if the project does not document *Home Assistant*.

Mandatory export: hassEntityEntries. Names of *Home Assistant* entities. Rendered in purple italics, HASS_ENTITY_COLOR, through pass 4 of the *pipeline*.

```
'use strict';
// ==============================
// [Préfixe] - HASS Entities (TS).js
// Entités Home Assistant — rendues italique violet (passe 4).
// ==============================

const hassEntityEntries = [
  // 'light.salon',
  // 'sensor.temperature_salon',
];

module.exports = { hassEntityEntries };
```

6.5 Colours

File: [Prefix] - Colors (TS).js, for the colours of this project. A second level exists outside the project — Kit-X - Colors (TS).js — for those that follow their author from project to project; it is driven by no flag, it is there or it is not. Flag Registry.requires.colors, default false. File absent if the project keeps the Kit colours.

Mandatory export: colorEntries, an object with two tables — docx for documents, html for web pages. Each table carries only the replaced keys; an absent key keeps the *stylesheet* default. The key names and their defaults are tabulated in the Reference of both stylesheets.

```
'use strict';
// ==============================
// [Prefixe] - Colors.js
// Couleurs du projet — remplacent celles du Kit.
// ==============================

// Deux tables distinctes : le contraste sur fond blanc imprime ne se
// regle pas comme le contraste sur ecran.
const colorEntries = {
  docx: { HEADING_COLOR: '8B0000' },
  html: { HEADING_COLOR: '#8B0000', CODE_BG: '#FAFAFA' },
};

module.exports = { colorEntries };
```

Note: *Three levels stack, from the most general to the most particular: the stylesheet defaults, then Kit-X - Colors.js if it exists, then the project's own. Each level redefines only what it wants to change. A palette placed in Kit-X follows the user from project to project; a project palette holds for that project alone.*

6.6 Conditional call at the head of a generator

Every project generator loads the *Registry* at start-up and calls on each *data module* only if its flag is set. This pattern is the direct technical counterpart of the Registry.requires flags — it replaces the earlier rule under which the *setters* were called unconditionally.

```
// ---- Tête de générateur projet — chargement conditionnel ----
const style    = require('./Kit - Stylesheet - General - Code.js');
const Registry = require('./[Préfixe] - Registry.js');

// Glossaire — défaut true
if (Registry.requires.glossary) {
  const { glossarySearchTerms } = require('./[Préfixe] - Glossary - Terms
(TS).js');
  style.setGlossaryTerms(glossarySearchTerms);
}

// Marques — défaut false
if (Registry.requires.brands) {
  const { brandEntries } = require('./[Préfixe] - Brands (TS).js');
  style.setBrands(brandEntries);
}

// Variable Friendly Names — défaut false
if (Registry.requires.variableNames) {
  const { variableNameEntries } = require('./[Préfixe] - Variable Friendly
Names (TS).js');
  style.setVariableNames(variableNameEntries);
```

```
}

// HASS Entities — défaut false
if (Registry.requires.hassEntities) {
  const { hassEntityEntries } = require('./[Préfixe] - HASS Entities
(TS).js');
  style.setHassEntities(hassEntityEntries);
}
```

Note: *If a flag is false, the matching file is absent from the project and neither a require() nor a setter call is made for that module. The inactive pipeline passes stay silent — the text functions keep working without error for the other active passes. The discipline is audited by kit_check_setters.py, see Quality Control §3.4.*

6.7 Checking protocol after a Kit injection

When the user reports a Kit injection, the *AI* applies this protocol to the *data modules* before any other work. It completes the general divergence diagnosis of the Update Guide §4.4, of which it is the module-specific form.

- **Read §6 in full:** the normative structure may have changed with the injected Kit.
- **Check every module present:** verify that all mandatory fields are there and that the exports conform to the contract of §6.1 to §6.4.
- **Check Glossary Terms in particular:** the presence of the tag field on every entry, and the surface and *anchor* format on glossarySearchTerms. A project coming from an earlier Kit may carry an old format in which the tag is absent and glossarySearchTerms exposes termKey instead of *anchor*. The migration adds the tag to each entry through _slugify(term), collision-safe, and updates the export.
- **Report every divergence:** in the form “Divergence found in [file]: [description]. Migration proposed: [action].”
- **Wait for confirmation:** no migration is applied without explicit agreement, migration by migration.

6.8 Document labels module

A file name is an identifier: it does not change from one language to another. The displayed label, by contrast, follows the reader's language — on the site *landing page*, in a reference from one document to another, and in the subtitle of a *cover page*. Without this module a German reader reads French titles.

File: [Prefix] - Document Titles (TS).js, driven by the requires.documentTitles flag. It exports the following tables.

- **titleEntries.** One entry per document, key = file name without *timestamp* or language suffix. Values: the subject in FR, DE, EN and LU; alias, short forms used in the corpus; aliasSection, single-word forms

that count only before a section reference; slug, identifier of the HTML page or null if the document is not published; multilingue, true if the slug carries a language suffix.

- **categoryLabels.** The translated file-name categories — the table of Naming Convention §5.1 made executable. It serves to compose the full label of a document and the subtitle of its *cover page*.
- **sectionTitles.** The *tab* titles of the *binder*, by number and language. The cover-page index and the site's per-language index read them here. They used to live in the site generator, out of reach of the cover *renderer*, which therefore displayed French titles in a German *binder*.
- **coverHeader.** The two lines of the cover banner, by language. They come from the rendering constants of the .docx, which carries only one version: a German rendering therefore showed a French banner.
- **The lecture field of titleEntries.** The name of a document as one writes it in a sentence — The update guide. Entered by hand, language by language: an article, and sometimes a preposition, cannot be derived from the subject. The *binder* index and the site index use it; the file name stays the technical form.

An empty value makes both the label AND the address fall back on the project language: serving a French label while pointing at a page that does not exist would be worse than not translating. The population rule: every document in the project language appears there.

Note: *The generator passes titleEntries and categoryLabels to the stylesheet through setDocumentTitles, as it does for language, glossary and colours: those two are all it needs, since it composes document labels. sectionTitles and coverHeader serve only the cover renderer and the site generator, which read the module directly. The stylesheet imports no data file.*

6.9 Text Highlights

[Prefix] - Text Highlights (TS).js exports highlightEntries, a flat list of fragments. Each declared fragment is rendered in italics at every occurrence, in every document of the project, without marking. Flag textHighlights, default false.

- **What goes in.** An entity that is one at every occurrence and is neither a brand nor a glossary term: a newspaper, an organisation, the name of an artificial-intelligence *model*, a third-party piece of software.
- **Precedence.** The pass sits after brands and before the glossary. A fragment already carried by one of those tables keeps its rendering: a duplicate entry adds nothing and would deprive a term of its link.

Note: *The pass marks every occurrence. An emphasis whose scope depends on the sentence therefore has no place in this table: it is written at the intended spot —*

General Reference §5.5 and §5.6.

7 Project glossary

The glossary document is the public face of the project glossary. It forms a couplet with [Prefix] - Glossary - Terms (TS).js — both files share the same *timestamp* and are always regenerated together.

The glossary is always regenerated from Terms.js, never the other way round. The .docx file is a visual rendering of the .js module — any change of content, adding, removing or altering a term, goes through the .js first, and the regeneration of the .docx follows mechanically with a fresh shared *timestamp*.

A project publishing its glossary in several languages produces one document per language. The couplet then becomes one terms module and n documents, all carrying the same *timestamp* and regenerated together. Adding a term, or correcting a single language, therefore forces regeneration of the whole. The naming of these documents belongs to Naming Convention §2.1 and §2.4: category and subject in the project language, language tag after the subject. This document does not restate it — a rule has a single home.

Each variant has its own entry in Registry.documents with its *timestamp*, so that HTML deployment republishes only the pages actually changed.

The files are absent from the project if Registry.requires.glossary is false. The formatting *pipeline* works without them — no term is italicised in teal in that case.

8 Project Cover Sheet

Couplet [Prefix] - Documentation - *Cover Sheet* (TS).docx and [Prefix] - Documentation - *Cover Sheet* (TS).png — *PCL* constants and generated *cover page*. Flag Registry.requires.coverSheet, default true. Both files share the same *timestamp*.

The PNG is produced by the *Python renderer* kit_render_cover_sheet.py — stateless and deterministic. Any visual change goes exclusively through the *PCL* constants of the .docx, never through the *renderer* code. The *renderer*'s full contract and the *inventory* of every *PCL* constant are documented in the *Cover Sheet*.

8.1 Skeleton of the .docx

The file [Prefix] - Documentation - *Cover Sheet* (TS).docx holds the normative sections of the Kit *model*. Every project duplicates the Kit Cover Sheet.docx and then customises §2 and §3 only — the other sections stay identical to the Kit *model*.

Section	Content	Project customisation
§1 Purpose	Description of the *Cover Sheet* role — *cover page* of the project *paper binder*	None — faithful copy of the Kit *model*

Section	Content	Project customisation
§2 *Binder* organisation	Table of the project headings — number, *tab* title, attached documents, *Tab*	Full customisation — see §8.2
§3 *PCL* — project-specific constants	Table of the *PCL* constants customised by the project	See §8.3 — typically HEADER_TEXT alone
§4 Updating this document	Rules for updating the couplet — fresh *timestamp* shared by .docx and .png	None — faithful copy of the Kit *model*

8.2 Customising §2 — the project headings

Table §2 of the project *Cover Sheet* lists the project's headings, not the Kit's. The number of numbered headings varies freely from project to project, within TAB_COUNT, which is 12 tabs.

TAB_NUMBERED_MAX is not a limit to respect but a derived value: the *renderer* computes it automatically by counting the active headings of §2. The matching *PCL* line is useful only to reserve extra tabs explicitly — five active headings but eight coloured tabs, say, to prepare three future ones.

For each heading: *tab* number, title, list of documents attached; the colour follows from the number — §11. The documents listed are the project's published documents — a document appears in one heading only.

Note: *The list of headings and their attached documents is settled interactively when the project is set up — see Initialisation Guide §5.*

8.3 Customising §3 — project-specific rendering constants

Table §3 of the project *Cover Sheet* lists only the *PCL* constants the project customises. All the others — A4 at 300 dpi, colours, margins, fonts, *tab* positions — are inherited from Kit *Cover Sheet* §3 without change or duplication.

In practice the only constants systematically customised are HEADER_TEXT_1 and HEADER_TEXT_2: the two lines shown in the orange banner of the *cover page*. The break between them is decided when writing, not computed when rendering — the subject on the first line, its qualification on the second. A short text may fill HEADER_TEXT_1 only and leave the second empty.

Their value follows the project's typographic convention — typically sentence case or Title Case, for *instance* Immobilier — patrimoine et acquisitions, Photogear, Roodt-Hass — domotique. All-caps is discouraged: it reads as a typographic shout.

Kit exception: the Kit shows “KIT de documentation” on the first line and “assistée par intelligence artificielle” on the second. The acronym KIT is capitalised as a deliberate signal — the Kit is the canonical reference for every project, not a project among others. That exception does not transfer to projects.

If the project needs to customise another *PCL* constant, the new value appears in §3 of the project *Cover Sheet* with an explicit “override Kit” mention.

8.4 Duplication recipe

When setting up a new project, or adding the *Cover Sheet* to an existing one, the procedure follows five steps.

- Duplicate the Kit Cover Sheet.docx to [Prefix] - Documentation - *Cover Sheet* (TS).docx with a fresh *timestamp*.
- Keep §1 Purpose and §4 Updating as they are — no change.
- Rewrite §2 *Binder* organisation with the project's headings, following §8.2.
- Reduce §3 *PCL* to the customised constants alone, typically HEADER_TEXT. The other Kit constants are inherited implicitly.
- Generate the PNG through kit_render_cover_sheet.py, passing the project.docx and the project *Registry* as arguments. The couplet is delivered together.

Note: *Any later change to the project Cover Sheet — a new heading, a changed HEADER_TEXT, an overridden PCL — regenerates the whole couplet,.docx and.png, with a new shared timestamp.*

8.5 When not required

If Registry.requires.coverSheet is false, the project has no *paper binder*. Rare in practice — every project documented so far holds its *Cover Sheet*. When the flag is false, the *Cover Sheet* couplet is absent from the project and the *renderer* kit_render_cover_sheet.py is not called.

8.6 Relation to the HTML landing page

The structure of the paper *Cover Sheet* and that of the project's HTML *landing page* are declared separately. §2 of the Cover Sheet.docx is the canonical source of the *paper binder*'s organisation; the SECTIONS constant of gen-[prefix]-site.js is the canonical source of the web landing. No script reads one to drive the other. Whether they match is the project's choice: the Kit keeps them identical, a project may put together a lighter *binder*. That the two indexes differ is intended by design, not a fault to correct: the paper carries a *binder* in the base language, the site publishes what is read online, in every language. The look — colours, spirit and reading key — is preserved in every case. See *Pipeline HTML* §2.5 for the full rule. Any silent

synchronisation of the two structures on the assumption of drift is catalogued as an *anti-pattern* in *Quality Control* §6.

9 Project HTML pipeline

The file [Prefix] - Documentation - *Pipeline* HTML (TS).docx belongs to each project — it documents the publishing decisions specific to the project's sub-site. It does not replace *Pipeline HTML*, which stays the source of truth for the format and the shared technical rules.

Mandatory minimum content: seven publishing decisions must appear in it.

- **Inventory of documents:** the list of the project's documents eligible for HTML publication — one line per document.
- **Greyed or not greyed flag:** for each document, the decision to show the line in grey italics — listed but not linked on the *landing page* — or as a clickable link to its HTML page.
- **With or without PDF flag:** for each published document, the decision to generate a PDF version and to show the matching icon in the document bar and on the *landing page*.
- **Project sub-domain:** root URL of the sub-site — used by renderLandingPage.homeHref and by the back links of every document page.
- **Landing quote:** the quotation shown on the sub-site *landing page*. Defined verbatim in §2.4 of the project *Pipeline HTML*, at the same place as in Kit *Pipeline HTML* §2.4 for the Kit site. The project's site generator copies this text as-is into its LANDING_QUOTE constant.
- **Annexes:** the list of PDFs in the Annexe category, placed by hand in the *working tree*, each coupled with a short Kit.docx describing its content. The piece is declared by the piece key in the *Registry*; the generator copies it under assets/pieces/, under its original name, and generates the .html from the coupled .docx, whose button serves the piece. See *Pipeline HTML* §6.3.
- **Manuals:** the list of PDFs in the Manual category — original documents from the manufacturer or producer — each coupled with a short Kit.docx describing the equipment and its use. Same *pipeline* mechanism as annexes.

Pipeline HTML supplies the format *model* for these sections and the shared technical rules — structure of the html folder, mandatory external CSS, assetsBase, HTML glossary conventions. The project *Pipeline HTML* completes those rules with the project's own decisions.

10 Dependency chain and lifecycle

10.1 Main chain

The *project files* hang together in a strict chain of dependencies. Breaking it produces documents that are technically valid and visually wrong.

- **Data modules to pipeline:** Brands.js feeds setBrands and gives small caps bold in pass 1; Glossary Terms.js feeds setGlossaryTerms and gives teal italics in pass 2; Variable Friendly Names.js feeds setVariableNames and gives dark green italics in pass 3; HASS Entities.js feeds setHassEntities and gives purple italics in pass 4. URL detection in pass 5 is automatic, with no module.
- **Kit stylesheet to produced document:** every .docx is generated by a **Node.js** script that require()s the active Kit *stylesheet*.
- **PCL to PNG:** the visual rendering of the *cover page* depends exclusively on the *PCL* constants of the Cover Sheet.docx, interpreted by the *Python renderer*.
- **PCL to landing page:** the same *PCL* constants feed HEADER_TEXT and TAB_COLORS on the HTML *landing page*.
- **HTML stylesheet to CSS:** getCSS() produces the file assets/kit-style.css, referenced externally by every HTML page of the sub-site.

10.2 Absorbing a Kit update

When the *AI* reports a Kit update, absorption follows three steps, ordered by dependency. No shortcut allowed.

- Read Kit - *Registry.js* to obtain the active versions of General, Glossary, YAML and HTML, and the timestamps of the Kit documents.
- Compare each active version with the one used at the last *generation* of a project couplet — the reference is the StylesheetVersion injected as a *fingerprint* in the project.docx.
- For each project couplet affected by a version gap, regenerate the couplet with the matching fresh *timestamp*. Regeneration follows the chain of §10.1.

Note: *Project files were historically updated only after the Kit files. That model is dropped. The active rule is: a project absorbs a Kit update as soon as it is called upon, following the procedure above. The detailed divergence-diagnosis protocol is in Update Guide §4.4; its data-module form is in §6.7.*

11 Governance

The rules below govern the evolution of every project using the Kit. No exception is granted without an explicit *instruction*.

- **Kit prefix reserved:** the *AI* neither creates nor modifies any file with the Kit prefix unless the user is explicitly working on an evolution of the Kit itself.
- **Inviolable couplets:** Kit stylesheet.js and Reference.docx; Kit Cover Sheet.docx and.png; project Cover Sheet.docx and.png; project Glossary Terms.js and Glossaire.docx. Regenerating one member entails regenerating the other with the same fresh *timestamp*. Full *inventory*: *Prompt* §6.

- **Generator from scratch:** no *ad hoc generator* gen-*.js is reused from one session to the next. Each session starts from the active Kit *stylesheet* and its Reference.docx, the script rewritten in full. Stable artifacts — §12 and *Quality Control* §3 — are not subject to this rule: shipped with the Kit so that a project has everything it needs from the start, they change only when their logic does.
- **PCL renderer only:** any visual change to the *Cover Sheet* goes through the *PCL* constants of the .docx, never through the *renderer* code. The *renderer* is stateless and deterministic: the same *PCL* constants give the same PNG.
- **Registry flags — single source:** the project *Prompt* may reference the flags present in Registry.requires but never duplicates their value. Any other document mentioning a flag must point at the *Registry* as the reference.
- **Sub-site infrastructure:** the .htaccess and the robots.txt of a sub-site belong to the **sliver.lu** project. No Kit or project generator produces them, and they never pass through a deployment ZIP. *Inventory* of expected and forbidden files: *Prompt* §16.
- **Delivery in one archive:** the files produced during a session are delivered in a single archive carrying the complete state of the project, never file by file. The recipient replaces a folder instead of piecing parts together, and their backup cannot diverge from the working version.
- **Composition of a delivery:** the following archives, handed over in the same *exchange*. The complete project archive, always — it carries the whole state and replaces the previous folder. The site publication archive, as soon as a pass has been made, full or incremental. One more if the project carries a Kit-X extension: it is delivered separately, because it does not belong in the same place — it follows its author, not the project — §3.

Note: *The order of deposit is stated where it matters. The download archive goes into assets/downloads/ before the site ZIP: deposited after, the download icon leads to a missing page until the archive arrives.*

- **Project archive:** the archive of the whole tree — tools, stylesheets, modules, documents, *Registry* — under the name declared in the *Registry* in projectZip.filename. It is produced by the publication pass when projectShareable is true, placed under the site's assets/downloads/, and it is this archive that is delivered: there is no separate working archive any more.
- **Self-contained complete project:** the archive carries everything the project needs to run for whoever receives it, without fetching anything from a session.
- **npm modules:** a package.json at the root declares each module and its minimum version. Every installation is done fresh, with *npm* install, never by copying a node_modules.
- **Version number:** it goes up when the structure of the project or the instructions that govern its behaviour change: organisation of files and

sections, naming conventions, *Prompt*, check rules, stylesheets. The second figure follows these changes; the first goes up only if a project using the Kit has to be migrated to follow it. A text correction, a translation or a republication of the site does not raise it. Each increase is recorded in the *Registry* changelog, with its reason. The date projectZip.updated changes with it. Every delivery that changes a file raises the number, text corrections included: two archives bearing the same number with different content leave the sub-projects unable to tell them apart.

- **Site regeneration:** a site *generation* pass is likewise delivered in an archive, whether the pass is full or incremental. The distinction governs what the archive contains, not the form in which it is handed over.
- **Naming of archives:** a delivered archive carries a fresh *timestamp*, and its name follows the pattern of its family — Naming convention §4.5, which tabulates the cases. The project archive is the exception: its name comes from Registry.projectZip.filename and carries a version number, because it is a published address — the site's link changes only with that number.
- **Identical index structure:** a project's documentation presents itself like the Kit's — index by headings, each document in HTML and PDF, language icons on documents that have variants, download icon for the project archive. The only deviation admitted concerns the theme colours — bar, headings, links — through the colours module. The six *tab* colours, derived in a cycle from the section number, belong to paper: there they tell the *binder*'s *dividers* apart. The *landing page* does not carry them — a section number is written there in COVER_NUM_COLOR, a single colour for the whole site, settable per project like the bar's.
- **Project journal:** every project keeps a journal of open entries, debts and dated decisions. Structure and content in §14. It is the only document of the project allowed to carry a history.

12 Project stable executable artefacts

A project may produce its own stable executable artefacts: scripts reused session after session, living in the project's *working tree*, distinct from the *ad hoc generators* recreated every session. Typical example: trading_generate_history_report.py.

12.1 Naming convention

Unix snake_case nomenclature. Project prefix in lower case followed by an underscore, name in snake_case, no *timestamp* in the name. A formal exception to Naming Convention §2, documented in that same document at §3.3. Project examples: trading_generate_history_report.py, hexi_validate_recipe.py. Criterion: a stable executable artefact — *Python*, **Node.js** CLI, shell. Any other file follows the main pattern.

12.2 Role — validator or renderer

Two possible categories. A *validator* returns exit code 0 if everything conforms, non-zero otherwise, and blocks delivery on failure. A *renderer* produces a deterministic artefact: the same input and the same *context* give the same output. If the project need falls into neither category, it is probably an *ad hoc generator* rather than a stable artefact — use the gen- prefix and Naming Convention §3.6. Matches the Kit split documented in *Quality Control* §3.

12.3 Lifecycle

The project's stable artefact lives permanently in the project's *working tree*, not the Kit's. It is not regenerated every session — only when its logic changes. Its evolution is tracked in a changelog block at the head of the file itself, as for the Kit stylesheets. No external tracking document is imposed — the standard Unix convention applies. The artefact carries no *timestamp* in its name.

12.4 Creation recipe

When a project identifies a recurring need that would warrant a stable artefact, the session creating it follows seven steps.

- Confirm with the user that the need is genuinely recurring and not a one-off.
- Name the file according to §12.1.
- Write the self-contained script, keeping external dependencies to a minimum.
- Document the dependencies in the file header.
- Document the usage contract in the docstring: inputs, outputs, return codes, example calls.
- Initialise the changelog block at the head of the file.
- Deliver through present_files so that the user places the file in the project's *working tree*.

12.5 Later evolution

When a stable artefact must evolve, whether for a bug or an extension, the session is dedicated to it and not mixed with other deliverables. The source is read directly from the *working tree* — they are read there directly. The change increments the version in the changelog block at the head of the file, with a description. Then the standard *sequential delivery*: validation, copy to outputs, present_files.

Note: *A stable artefact belongs to the complete project delivery, on the same footing as the documents and the stylesheets. An archive omitting it would not allow a restart from nothing, and the gap would show only when restoring.*

13 Kit bug report

A project may meet a Kit behaviour that deserves reporting: unexpected rendering of a *stylesheet* function, divergence between the .js code and its Reference.docx, an *anti-pattern* not yet catalogued, a *validator* false positive. The structured reporting channel is a short dedicated document, produced by the project and delivered to the Kit project *chat* for analysis and correction. It replaces the informal briefs used historically — same content, unified format.

13.1 Purpose and triggers

The *Kit bug report* is produced when the project detects a Kit behaviour that cannot be resolved locally in the project. *Prompt* §5.3 applies: a local workaround never replaces a fix at the source. If the root cause is in the Kit, it is the Kit that must fix it.

Three typical triggers.

- **Stylesheet bug:** a *stylesheet* function does not render as documented in its Reference.docx — divergence between runtime behaviour and contract. Example: the badly named bgColor parameter of sectionBanner, corrected to accentColor after a report from the Sorso project.
- **Pipeline bug:** a *pipeline* step produces an inconsistent or silently truncated result. Example: gen-kit-site.js was not calling the data *setters* of the HTML *stylesheet* — the Kit site historically had neither brand markup nor glossary links.
- **Uncatalogued anti-pattern:** a recurring error found in the project that would warrant an entry in *Quality Control* §6 and, ideally, an automatic check in *Layer 2*.

13.2 Nomenclature

The *bug report* follows the standard pattern of Naming Convention §2.1: [Prefix] - Project - *Kit bug report* (TS).docx. Category Project, per §7, open to both prefixes. The subject "*Kit bug report*" stays in English in every language — a stable technical name in the sense of §5.2 of the Convention.

The Project category is translated according to the reporting project's project.docLanguage.

The file is generated by the project through a gen-kit-bug-report.js script written from scratch, like any project document — formatting engine the Kit General *stylesheet* active at the time of writing, kit_validate_docx as the output check.

13.3 Structure of the document

Mandatory sections, in this order. Short by construction — a *bug report* is not a thesis: it describes, reproduces, and proposes. Detailed validation and correction follow in the Kit session.

The *bug report* is a document of record — *Kit Prompt* §12 — whose structure is fixed here: it reports on an existing defect, smooths nothing over, and its last section carries what remains to be decided.

- **§1 Identification:** project prefix, date of observation, project.docLanguage, the Kit *stylesheet* versions active when the bug appeared — read from Kit - *Registry.js*, since the project *Registry* does not carry them, and copied as-is — and the documents or pipelines involved.
- **§2 Observation:** observed behaviour and expected behaviour, in two short paragraphs. Describes factually what happened, without interpretation.
- **§3 Reproduction:** minimal extract from the gen-*.js, or the command that triggers the bug, as a rawBlock, plus the exact sequence executed. Precise enough for the *AI* on the Kit side to reproduce it in session without further questions.
- **§4 Impact and suggested diagnosis:** which documents or pipelines are affected and how severely. The project's hypothesis on the root cause, not binding: the *AI* on the Kit side redoes the full analysis. State whether a local workaround exists or not.

13.4 Reporting workflow

Once the bug report.docx is generated and validated on the project side, reporting happens by depositing several binaries at once in the Kit project *chat*. The Kit project has access only to what is attached in the *chat* — not to the *working tree* of other projects.

- **The bug report .docx:** the main file, binary in the *chat*. Not the *working tree*: extracted text is insufficient for the canonical content of a .docx.
- **The offending gen-*.js:** the project generator that triggers the bug, or a minimal extract — as text in the *chat*, or as an attached.js file.
- **The resulting .docx:** if the bug produces a badly rendered.docx, attach the binary for inspection — table count, w:tbl, *custom properties*.
- **Any screenshots:** Word, *LibreOffice* or browser captures if the bug is visual — attached as images.

Note: *The bug report is read at the start of the Kit session as a brief. The AI on the Kit side matches it against the active rules, identifies the layer concerned, proposes a correction plan, and waits for the explicit go before any generation. The standard sequential delivery cycle then follows.*

13.5 Non-permanence

The *bug report* is an ephemeral file: produced, delivered, archived once resolved. It does not appear in the *inventory* of §2 — that lists the permanent *project files* which survive between sessions, which a *bug report* does not. The trace of the resolution lives elsewhere: an entry in the Todos on the Kit side, possibly an update to a Kit document.

Note: *Retention on the project side: the user's free choice. No mandatory storage in the working tree — the bug report has done its work as soon as the Kit fix is delivered and the project has absorbed the new version of the couplet concerned, following §10.2.*

14 Project journal

Every project keeps a journal. It carries what remains to be done and the dated trace of what was decided — that is what makes it possible, months later, to find out why something was done this way rather than another.

It is modelled on *Kit - Projet - Todos*, which is the reference copy. Its sections, in this order.

- **§1 Open bugs.** What is broken and unrepaired. One entry per defect, with what it produces and what it prevents. The section carries the number of open entries and its date, so that “no entries” reads as a dated observation.
- **§2 Work in progress.** What needs a decision before it can be done, or work whose scope exceeds one session. An entry names what is blocking, not what one wishes for.
- **§3 Technical debts.** What works and is known to be imperfect. A debt that bothers no one today is still a debt: it is recorded with the reason for not clearing it now.
- **§4 Register of resolutions.** A four-column table — number, title, date, content — in continuous numbering that is never reused. Each entry says what was decided and why, with the versions delivered.

Note: *What is recorded: structural decisions. A change of form, a rule set or withdrawn, a defect fixed, an arbitration settled. What is not recorded: generation passes, publications, data updates — they leave their trace in the timestamps, not in the journal.*

Note: *The journal is the only document of the project allowed to carry a history. The others describe a state; if they also carried their own past, a reader would no longer know which of the two is authoritative.*

15 Language selection

A fully multilingual site and a site of which only some documents are translated do not have the same need. The Kit offers both mechanisms; the project declares which it uses, under rendering.languageSelector.

Mode	What it assumes	Where the selector appears
site	Every published page exists in every declared language	Menu at the end of the bar, identical on every page
document	Nothing; only some documents are translated	Icons at document level, where variants exist

Note: *The default is document: it assumes nothing about the project. Missing it breaks nothing and shows on the first page opened — the generator also announces the mode at every pass. A loud stop is reserved for defects that do not show.*

Note: *In site mode, an untranslated document would make the menu appear and disappear from page to page, and it would stop being a landmark. A project that publishes monolingual documents uses document mode.*

- **What the project provides.** The list of each page's variants — language code and address. The Kit provides the two mechanisms, their localised labels and their flags.

16 Base language and translations

The project declares its base language under project.docLanguage. Documents written in that language are the reference: a rule, a correction or a decision is written there first, and work is done from them, by people and *AI* alike.

Translations are a service to the reader. They render the sense so as to read like a text written in their own language, never word for word. Nothing is lost or added; file names, code and section numbers stay as they are.

Each variant carries, at the top of its §1, the fixed note from *Kit Prompt* §17, with the project's base language. A translation is updated on request, from the map of the reference document, and its skeleton is checked with kit_check_ossature.py.

Full doctrine: *Kit Prompt* §17.