The online edition, which follows the packages as they change:
https://compmd.hakanserce.com/guides/developers/

CompMD Developer's Guide

How to build with CompMD: documents people write as text, rendered as pages in your application, with components of your own and from packs, and safe to open from anyone. The spec says what CompMD is; the package READMEs say what each package has; this guide shows how to build with them, from a first page to a published pack. Writing documents themselves is the Author's Guide.

It is written for the `@compmd` packages at version 0.3.0, and follows them as they change. On the website it is at compmd.hakanserce.com/guides/developers/.

Who it is for

Developers who put CompMD in an application, or write packs for it. You know JavaScript or TypeScript and a bundler such as Vite. You want to know about the packages, the sandbox and its trust levels, components in React, Svelte or JavaScript, and publishing. You don't need to know CompMD's syntax first: each chapter explains what it uses, and links to the Author's Guide for the rest.

If you write documents rather than code, the Author's Guide is the book for you. The two meet at components: a component written in CompMD is in the Author's Guide, and one written in code is in this guide.

Contents

Part I. Getting started

1. Introduction
2. Quick start
3. How it fits together

Part II. Documents in your app

4. Processing documents
5. Live editing

6. Data and inputs
7. Rendering and styling

Part III. Components

8. Components and trust
9. Packs from npm
10. Your app's own components
11. The sandbox, closer
12. The Developer Tools

Part IV. Writing packs

13. Your first pack
14. Signatures and params
15. React, Svelte and libraries
16. Code renderers
17. Testing and publishing

Part V. Beyond Vite

18. Other hosts
19. The Rust crate

Appendices

20. The configuration's reference
21. The diagnostics
22. A glossary

The examples

Each chapter's code is a file in `examples/`, which the guides' tests type-check, build and run in Chrome and WebKit, and compare with the chapter's text, so the code you read is code that works.

1. Introduction

CompMD is Markdown with components. A writer puts a box, a chart or an input in a document with a line of its own, and the document still reads well as text, on GitHub or in any Markdown viewer. Your application renders it as a page.

```
---
read: 12
pages: 3480
md:
  inputs:
    goal: number = 20
---

::: grid {min=12rem gap=m}
::: box {padding=m border=subtle border-radius=m}
**{{md.doc.read}}** books read this year
::: /box
::: box {padding=m border=subtle border-radius=m}
**:value[3,480]{of=(md.number(md.doc.pages, 0))}** pages
::: /box
::: /grid

:: number-input[Goal: 20 books] {bind=goal}
```

This guide is for developers who put CompMD in an application: a notes app, a documentation site, a dashboard whose pages people write, an editor with a live preview. It shows how to render documents, give them components of your own and from packs, and write packs others can use. How to write the documents themselves is the Author's Guide.

What CompMD gives your application

- **Documents your users write as text.** Markdown, which they know, with components, data, inputs and layout, which they can learn a line at a time. A document is a file they own: it diffs, it merges, it reads in any editor.
- **Pages you control.** The look of a page is yours: its stylesheet, its tokens, its components. The document says what is on the page; your application says how it looks.

- **Documents from anyone, safe to open.** Components written in CompMD run no code. Components written in JavaScript, React or Svelte run in a sandbox that sees only its own params, with no network, no storage and no way out of its box. A document from a stranger can't reach your page, your session or your data.
- **Diagnostics with fixes.** Every mistake has a code, a place and, where one is clear, the edit that fixes it; an editor can show them as the writer types, with hints at the caret.

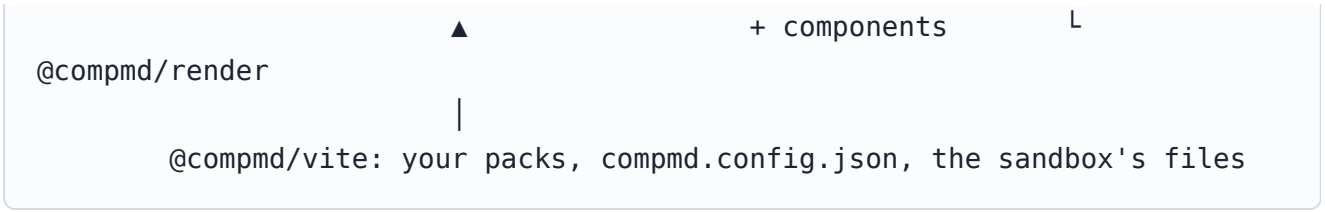
The packages

Package	What it does
`@compmd/processor`	Reads a document, with everything it needs, and gives its trees, its diagnostics and its components. The processor in Rust, as WebAssembly.
`@compmd/render`	Renders the expanded tree as a live page, and updates it in place as the document changes.
`@compmd/sandbox`	Runs components at the trust their pack is given: in frames with no network, or your own in your page.
`@compmd/vite`	Reads `compmd.config.json`, gives the processor your packs, and serves the sandbox and the packs' files from your app.
`@compmd/registry`	Gets the packs a document imports that your app doesn't install, from npm or another registry, checked against their integrity.
`@compmd/devtools`	The Developer Tools: a document's text, its page and what the processor and the sandbox make of it, as a part a page embeds.
`@compmd/pack`	`compmd-pack`, which builds a pack, checks it before it is published, runs its tests and serves its test page.
`@compmd/create-pack`	Writes a new pack: a component as one built module, its signature, and a test page.

An app that shows documents uses the first four, and `@compmd/registry` when it fetches packs it doesn't install (chapter 9). The quick start installs `@compmd/processor`, `@compmd/sandbox` and `@compmd/vite`; `@compmd/render` comes with the sandbox. The last two are for writing packs, in Part IV.

A document goes through them like this:

```
  notes.comp.md ──► @compmd/processor ──► expanded tree ──► @compmd/sandbox
──► the page
```



The processor does no I/O of its own: your application gives it the document, the files it reads, its packs and its data, and gets back everything the page needs. That's what lets the same processor run in a browser, in a worker, in a build, or natively in Rust.

Words this guide uses

Word	Means
document	a CompMD file, `.comp.md`, or any Markdown file with an `md` key in its frontmatter
frontmatter	the YAML between `---` lines at the top: CompMD's settings under `md`, the author's own values beside them
component	a named piece of a page: a container (`::: box` … `::: /box`), a leaf (`::chart[…] {…}`) or an inline one (`:value[…]{…}`)
params	a component's settings, in braces: `{padding=m}`
fallback	what shows where a component can't: a leaf's or inline component's text in brackets, a container's body
signature	the params and slots a component takes, which the processor checks every use against
pack	a document that exports components, published on npm or kept in a folder of your app
expanded tree	the processor's result: the document's content, with every value computed, ready to render
diagnostic	a mistake the processor found, with a code (`F…` an error, `W…` a warning), a place and a message
host	your application: what gives the processor its request and shows the page
trust level	how far a pack's code is trusted: strict, permissive or trusted, set by your configuration (chapter 3)

The glossary has the rest, linked to the spec.

How to read this guide

Part I gets you from nothing to a page in a few minutes, then explains what runs where. The parts after it each take one side of building with CompMD, and can be read in any order. Each chapter's code is a working example in `examples/`, and a document with a **Try it** link opens in the playground, where you can change it and watch the page.

Next: the quick start.

2. Quick start

A Vite app that renders a CompMD document, with an input the reader changes. Four files, about forty lines.

Install

```
npm create vite@latest my-pages -- --template vanilla-ts
cd my-pages
npm install @compmd/vite @compmd/processor @compmd/sandbox
```

The plugin

`@compmd/vite` reads your configuration, gives the processor your packs, and serves the sandbox's frame documents from your app:

```
import { compmd } from "@compmd/vite";
import { defineConfig } from "vite";

export default defineConfig({
  plugins: [compmd()],
});
```

A document

Save this as `notes.comp.md`. Its frontmatter has a list of tasks and an input, `done`, which the number input changes:

```
---
tasks: [Write the outline, Draft the first part, Ask for a review, Publish]
md:
  inputs:
    done: number = 1
---
```

```
# The guide, step by step

::: box {padding=m border=subtle border-radius=m}
**:value[1 of 4 done]{of=(string(md.input.done) + " of " +
string(size(md.doc.tasks)) + " done")}**

:: number-input[Done: 1] {bind=done}
::: /box

::: each {in=(md.doc.tasks) as=task join=list}
1. {{task}}
::: /each
```

Try it in the playground.

The page

A place for the page in `index.html`:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>My first CompMD page</title>
  </head>
  <body>
    <main id="page" style="height: 24rem"></main>
    <script type="module" src="./main.ts"></script>
  </body>
</html>
```

And `main.ts`, which opens the document and renders it:

```
import { init, OpenDocument } from "@compmd/processor";
import { createSandbox } from "@compmd/sandbox";
import { request, sandbox } from "virtual:compmd";
import text from "./notes.comp.md?raw";

await init();

// The document, open with what the app gives it: its packs, from
compmd.config.json.
```

```
let inputs: Record<string, unknown> = {};
const open = () =>
  new OpenDocument({ ...request, path: "notes.comp.md", files: {
"notes.comp.md": text }, now: new Date().toISOString(), inputs });
let opened = open();

// Its page, in a sandboxed frame.
const page = document.getElementById("page") as HTMLElement;
const view = createSandbox(sandbox).open(page, { components:
opened.components });
view.render(opened.expanded);

// A reader changed an input: open the document again with its new value.
view.on("set", ({ input, value }) => {
  inputs = { ...inputs, [input]: value };
  opened.free();
  opened = open();
  view.render(opened.expanded);
});
```

For `virtual:compmd`'s types, add `"@compmd/vite/client"` to `compilerOptions.types` in `tsconfig.json`.

Run it

```
npm run dev
```

The page shows the heading, a box with *1 of 4 done* and the number input, and the four tasks. Set the input to 3, and the box says *3 of 4 done*: the frame tells your page the input changed, and your page opens the document again with the new value.

What just happened

- `init()` loaded the processor, as WebAssembly.
- `OpenDocument` read the document with what `request` holds (your packs, none yet), and gave its expanded tree and the components it uses. The fields `main.ts` adds are the document's own: `path`, the document's name in `files`, which holds its text by path; `now`, the time the page is made, which `md.today()` reads, since the processor has no clock of its own; and `inputs`, the inputs' values by name.

- The frontmatter's `md.inputs` declared the input, `done: number = 1`: its type and the value it has until a reader changes it. `md.input.done` reads it in an expression, and `bind=done` ties the number input to it. A value in `inputs` that doesn't fit the input's type is ignored, and the default is used.
- `createSandbox(sandbox)` made a strict frame for the page, from the frame documents the plugin serves under `compmd/`, and rendered the tree in it.
- The number input is a built-in component; when the reader changed it, the frame sent `set`, and nothing else can: the page's code runs in a frame with no network and no access to your page.
- Opening again on each change is the simplest way to keep the page in step, and fast enough for a document this size. Live editing does it with less work, for an editor.

When something is wrong

Nothing in the document is ever an exception: a mistake in it is a diagnostic, and the page still renders, with fallbacks where it must. `opened.diagnostics` lists them, each with a code, a place and a message. Log them while you try things:

```
for (const { code, range, message } of opened.diagnostics)
console.warn(`${code} ${range.start.line}:${range.start.column}
${message}`);
```

Misspell `bind=done` as `bind=don`, and the page shows the input's fallback, *Done: 1*, while the list says `F030`: the input it names isn't declared. Only a request that isn't valid throws, such as a `now` that isn't an RFC 3339 time, or a `path` that isn't in `files`. Processing documents has the diagnostics in full.

Next: how it fits together, what runs where, and why.

3. How it fits together

The quick start used four packages in a few lines. This chapter is what each does, where it runs, and why it is split that way, so the parts after it make sense.

The processor: text in, trees out

`@compmd/processor` takes a **request**, a document and everything it needs, and gives back what a page needs. It runs in stages, each taking only what it uses:

Stage	Needs	Gives
Parse	the document's text	the parse tree; syntax diagnostics
Resolve	the files it imports, its packs	every component name bound to a definition
Validate	signatures	params checked and typed
Load	its data, the inputs' values	the data
Evaluate	all of the above, and the time	the expanded tree: content only, ready to render

The processor does no I/O. Your application puts everything in the request. That's why it runs the same in a browser, a worker, a build or natively, and why opening a document never makes it fetch anything on its own. Only `path` and `now` are required:

Field	Holds	The stage that reads it
`path`	the document's path in `files`	all
`files`	the files of the document's folder by path: the document, what it imports, its data files	parse, resolve, load
`packs`	the packs by name, each with its releases and their files	resolve
`ambient`	packs every document gets without importing them	resolve

Field	Holds	The stage that reads it
`policy`	your packs' trust, which packs a document may import, which are denied	resolve
`allowedOrigins`	the origins remote data may come from	validate
`remote`	remote data, fetched beforehand, by URL	load
`host`	your app's own data, for `host:` sources	load
`inputs`	the inputs' values, by name	load, evaluate
`now`, `tz`	the time the page is made, RFC 3339, and the time zone of a document without one	evaluate
`origins`	whether expanded nodes say where in the text they come from, as a live preview needs	evaluate

`virtual:compmd`'s `request` fills `packs`, `ambient` and `policy` from your configuration; the rest is the document's. `sources(request)` lists what a render needs that the request lacks, remote URLs, `host:` sources and packs, so you can get them first. Processing documents and data and inputs go through them.

Parsing needs the text alone, so an editor or a linter can parse any document, even one whose packs and data it doesn't have.

`process(request)` runs every stage once. `OpenDocument` keeps the document open: each edit processes only the blocks it touches, and gives a *splice*, what changed in the expanded tree, for an editor's live preview.

Diagnostics

Every stage reports what it finds wrong with the document as diagnostics, and none stops the others: a broken component renders its fallback, and the rest of the page is there. A request that isn't valid is the only thing that throws. Each diagnostic is the spec's machine form:

Field	Is
`code`	`F…` for an error, which changes what renders; `W…` for a warning, which doesn't
`severity`	`error` or `warning`, from the code

Field	Is
`message`	a sentence for the writer; not stable, so match on `code`
`range`	where: 1-based lines and columns, in code points
`file`	the file it is in, when it isn't the processed document, such as a pack's document
`related`	other places it involves, such as the opener of a container never closed
`stage`	the stage that found it: `parse`, `resolve`, `validate`, `load` or `evaluate`
`fix`	when there is one, the edit that fixes it, for a quick fix

The processor gives them in stage order, with its result. Two come later, from the sandbox, since only running a component's code finds them: `W118` for a component that threw or failed, and `W121` for code that can't load (the processor reports `W121` itself only for a pack whose `peerDependencies` your frames can't meet). They arrive through the view's `error` event, `view.on("error", ({ component, code, message }) => …)`. The diagnostics appendix lists every code.

The expanded tree, and the components

The expanded tree is content: headings, paragraphs, lists, boxes, input components, and the components written in code, by name and params. It holds no code. The code of each component comes beside it, in `components`: a fence's or a file's text to compile, or a built module's place in its pack, with the trust its pack is given.

The renderer and the sandbox: trees in, a page out

`@compmd/render` turns a tree into a live page and updates it in place: a new tree, or a splice, changes only what changed, and components keep their state.

`@compmd/sandbox` decides **where** each component's code runs, by the trust its pack is given:

Trust	Where it runs	What it can reach	Who
strict (the default)	a frame with an opaque origin and no network	its params, its slots, its box	any pack, any document
permissive	a frame of its own	that, plus the network to origins you list, and `host` if you give it	packs your configuration lists

Trust	Where it runs	What it can reach	Who
trusted	your page	what your page can	your app's own components, built with it

A document can never raise a pack's trust: only your configuration can. So a document from anyone, and any pack it imports, can't reach your page, your session or your data.

The sandbox renders a document in one of two modes:

- **Frame mode** (the default, as in the quick start): the whole page in one strict frame. Simple, and right for documents from anyone.
- **Host mode**: the page in your page, your trusted components inline, every other component in a frame of its own. Right for an app with components of its own.

	Frame mode, `page: "frame"`	**Host mode,** `page: "host"`
The page	one strict frame per document	your page
Strict components	in the page's frame	a frame per use
Permissive components	their fallbacks, with an error `not-here`	a frame per use, with `host` if your configuration gives it
Trusted components	their fallbacks, with an error `not-here`	inline, in your page

Frames only go down in trust: your page holds permissive and strict frames, a permissive frame holds strict ones, and a strict frame holds none. So a permissive or trusted component in the slot of a strict container shows its fallback too. Which level each component gets comes with it from the processor, in `components`, set by your configuration; the sandbox only carries it out. Components and trust and the sandbox, closer say more.

The plugin: your app's side

`@compmd/vite` does at build time what every app would otherwise write:

- it reads `compmd.config.json`: which packs get more than strict trust, which are ambient (every document gets their names), which are denied;
- it finds your **packs**, in `node_modules` and in folders of your app, and gives them to the processor as `request.packs`;

- it serves the sandbox's **frame documents** and the packs' **built modules** from your app, under `compmd/`, with the headers the frames need;
- it builds your **trusted components** with your app, and gives them to the sandbox as `sandbox.trusted`.

`virtual:compmd` hands both to your code, `request` for the processor and `sandbox` for the sandbox, so you spread them in and write no policy, path or import map yourself.

Why it is split this way

- **The processor knows nothing of the browser**, so it can check documents in CI, render them at build time, or run in a native app.
- **The renderer knows nothing of trust**, so a host that renders documents it trusts completely can use it alone.
- **The sandbox carries the security**, in one place, tested against every threat it guards from, so no application writes a frame or a content security policy itself.
- **The plugin is the only part that knows your build**, and other hosts can do its job by hand, as other hosts shows.

Next: Part II, processing documents.

4. Processing documents

`process` takes a document with everything it needs and gives back its trees and its diagnostics. This chapter goes through its request field by field, the files a document reads, what comes back, and what to do with a diagnostic and its fix. It builds a script that checks a document as a build would, run with Node.

Loading the processor

`@compmd/processor` is WebAssembly. Load it once, before the first call:

- **In a browser**, `await init()` fetches the module next to the package's JavaScript, as the quick start does. A bundler such as Vite serves it.
- **In Node**, give the module's bytes to `initSync`: `initSync({ module: readFileSync(createRequire(import.meta.url).resolve("@compmd/processor/compmd_wasm_bg.wasm")) })`. The package exports the file by that name.

After that, every call is synchronous: `process`, `sources`, `frontmatterOf`, `componentsOf`, `plainHtml`, and the open documents of chapter 5.

The request

A request is a plain object. Only `path` and `now` are required:

Field	What it is
`path`	the processed document's path in `files`
`files`	the files of the document's root folder, by `/`-separated path: the document, the documents it imports, its data
`packs`	the packs a document may import, by name, each a list of releases: `{version, integrity?, broken?, files}`
`ambient`	packs every document gets without importing them, at the lowest priority
`reportAmbient`	report each use of a name from an ambient pack, W113

Field	What it is
`policy`	your policy for packs: their trust, whether a document may import packs you don't list, those you deny or left out
`allowedOrigins`	the origins remote data may come from, such as `https://status.example.org`
`remote`	remote data you fetched beforehand, by URL: `{text, contentType?}`
`host`	your app's own data, by source without `host:`
`now`	the render time, RFC 3339: what `md.now()` and `md.today()` give
`tz`	the time zone for a document without a `tz` key: an IANA name, or `UTC` (the default)
`inputs`	the values given for the document's inputs, by name
`origins`	whether expanded nodes carry their origins, as a live preview needs

In a Vite app, `virtual:compmd`'s `request` holds `packs`, `ambient` and `policy` from your configuration (chapter 9); you spread it in and add the rest. The processor reads nothing else: no file system, no network, no clock. The same request always gives the same result, which is what lets you check a document in CI and get what a reader will see.

`process` throws only when the request itself is wrong: `now` isn't RFC 3339, or `path` isn't one of `files`. Everything wrong with the document is a diagnostic.

The files a document reads

A document reads files by relative path: the documents it imports (`from: ./parts/notice.comp.md`) and its data (`from: ./releases.csv`). Each path resolves against the file it is written in, inside the root folder `files` stands for, and a path that leaves it (`../`) is F026. So you decide what a document may read by what you put in `files`.

The example is a folder with a document, the data it reads beside it, and a document it imports in a subfolder:

```
---
title: Releases this quarter
md:
  import:
    - from: ./parts/notice.comp.md
      names: [notice]
  data:
```

```
    - from: ./releases.csv
      as: releases
      schema: {version: string, date: string, changes: number}
---

# {{md.doc.title}}

::: notice {title="Three releases" border=subtle}
:value[24 changes]{of=(md.number(md.sum(md.data.releases.map(r,
r.changes)), 0) + " changes")} in all, each listed below.
::: /notice

::: each {in=(md.data.releases) as=r join=list}
- **{{r.version}}**, {{r.date}}: {{r.changes}} changes
::: /each
```

Try it in the playground.

Its data, `releases.csv`:

```
version,date,changes
2.0,2026-07-14,9
2.1,2026-08-25,11
2.2,2026-10-06,4
```

And `parts/notice.comp.md`, which defines the `notice` component in CompMD and exports it:

```
---
md:
  export:
    names: [notice]
---

A box with a title, for what a reader should see first.

```compmd component=notice {padding=m border-radius=m background=subtle}
params:
  title: string
slots: [default]
---
**{{title}}**
```
```

````
{{md.slot}}
```
````

Packs are not files of the folder: a document imports them by name (`from: "@kit/charts@1"`), and a pack is in the request only if `packs` holds a release of it. Each release's `files` are the pack's own: its `package.json`, its document, what that imports. A built module's text can be left empty, since the processor only names it for the renderer. Chapter 9 has where packs come from.

The script

`check.ts` reads a folder's files, asks what the document needs fetched, processes it, and prints each diagnostic with its fix:

```
// Checks a CompMD document as a build would: what it needs fetched, every
diagnostic with its fix, and whether any is
// an error. Run it with Node: `node check.ts <folder> <document>`, such as
`node check.ts . quarter.comp.md`.
import { readdirSync, readFileSync } from "node:fs";
import { createRequire } from "node:module";
import { join, relative, sep } from "node:path";
import { argv, exit } from "node:process";
import { initSync, process as processDocument, sources, type Diagnostic,
type Request } from "@compmd/processor";

// In Node the module is loaded from its file; in a browser, `await init()`
fetches it.
initSync({ module:
readFileSync(createRequire(import.meta.url).resolve("@compmd/processor/comp
md_wasm_bg.wasm")) });

/** The text files under `folder` a document may read, by their `/`-
separated paths in it. */
export function filesOf(folder: string): Record<string, string> {
  return Object.fromEntries(
    readdirSync(folder, { recursive: true, withFileTypes: true })
      .filter((entry) => entry.isFile() && /\.(comp\.md|csv|tsv|json|ya?
ml)$/.test(entry.name))
      .map((entry) => {
        const path = join(entry.parentPath, entry.name);
        return [relative(folder, path).split(sep).join("/"),
readFileSync(path, "utf8")];
```
```

```
      }),
  );
}

/** A diagnostic as a line: where it is, its code and message, and the edit
that fixes it, when it has one. */
export function lineOf(document: string, { file, range, code, message, fix
}: Diagnostic): string {
  const at = `${file ??
document}:${String(range.start.line)}:${String(range.start.column)}`;
  const fixed = fix ? `\n  fix: write \`${fix.text}\` at
${String(fix.range.start.line)}:${String(fix.range.start.column)}-${String(
fix.range.end.column)}` : "";
  return `${at} ${code} ${message}${fixed}`;
}

/** Checks `document`, one of `folder`'s files, rendered at `now`: the
lines to print, and whether it has an error. */
export function check(folder: string, document: string, now: string): {
lines: string[]; failed: boolean } {
  const request: Request = { path: document, files: filesOf(folder), now };
  const { remote, host } = sources(request);
  const needed = [...remote, ...host.map((source) =>
`host:${source}`)].map((source) => `${document}: needs ${source}, which a
check doesn't fetch`);
  const { diagnostics } = processDocument(request);
  return { lines: [...needed, ...diagnostics.map((diagnostic) =>
lineOf(document, diagnostic))], failed: diagnostics.some(({ severity }) =>
severity === "error") };
}

if (import.meta.main) {
  const [folder = ".", document = "index.comp.md"] = argv.slice(2);
  const { lines, failed } = check(folder, document, new
Date().toISOString());
  console.log(lines.length === 0 ? `${document}: no diagnostics` :
lines.join("\n"));
  if (failed) exit(1);
}
```

`process` is imported as `processDocument`, since Node has a `process` of its own. Install `@compmd/processor`, and run it with Node 24.2 or later, which runs TypeScript as it is and has `import.meta.main`:

```
npm install @compmd/processor
node check.ts . quarter.comp.md
```

It prints `quarter.comp.md: no diagnostics`.

The result

`process(request)` gives three things:

- **`parse`**, the parse tree: the document as written, mdast with CompMD's own nodes (spec §12). Tools that read or rewrite documents use it.
- **`expanded`**, the expanded tree: content only, with definitions, `if`, `each` and interpolations replaced by what they give. It is what a page renders (chapter 7), with the inputs and their values on its root.
- **`diagnostics`**, of every stage, in stage order.

`process` doesn't give the components the tree names with their code; an open document does (`OpenDocument.components`), which is why the quick start uses one even for a page that never changes.

Diagnostics and their fixes

Change `border=subtle` to `border=subtl` in `quarter.comp.md`, and run the script again:

```
quarter.comp.md:15:43 F009 `border` takes `none`, `subtle`, `strong` or a
CSS border, such as `1px solid #d0d7de`, so the param is treated as absent;
did you mean `subtle`?
  fix: write `subtle` at 15:43-48
```

It exits with 1, so a CI job that runs it fails. A diagnostic has:

Field	What it is
`code`	`F…` for an error, `W…` for a warning (spec §10.2)
`severity`	`error` or `warning`
`message`	what is wrong, in words, for people
`file`	the file it is in, when it isn't the processed document: an imported document, or `@scope/pack@version/path` for a pack's

Field	What it is
`range`	where: `start` and, mostly, `end`, each a 1-based `line` and `column`
`related`	other places that matter, each with its `file`, `range` and `message`, innermost first: the invocations that led to an error in a definition, say
`stage`	`parse`, `resolve`, `validate`, `load` or `evaluate`
`fix`	when the message suggests an edit: the text at `fix.range`, on one line, becomes `fix.text`

Columns count code points, not UTF-16 units: in the line `😀 Hi`, `Hi` is at column 3, while JavaScript counts the emoji as two units and puts `Hi` at index 3, which would be column 4. To make a fix in a JavaScript string, count the line's characters as code points; chapter 5's editor does, in `offsetOf`.

The processor always recovers. With an error, it drops the smallest broken piece, a param, a line, a record, and goes on, so there is always a page: here the box has no border. It never hides author content (spec §10.3). What an error means is yours to decide: a build fails on any `F`, as the script does, and a preview shows the page with the diagnostics beside it.

What a document needs fetched

The processor fetches nothing. `sources(request)` says what a render of the document needs that only you can get:

- `remote`: the URLs of its remote data, from origins in `allowedOrigins`;
- `host`: its `host:` sources, without `host:`;
- `packs`: the packs it imports that `packs` has no matching release of, each `{pack, range, integrity?}`.

Each comes once, in the order the document names it, and it lists what the render needs, whether or not the request already holds it: compare with what you have. You fetch, put what you got in `remote`, `host` or `packs`, and process. A source you couldn't get stays out of the request, and the document shows its `fallback`, or F026. Chapter 6 does this for data, and `@compmd/registry` for packs (chapter 9). The script only reports what it finds, since a check fetches nothing.

Other questions to ask

Two more functions take the same request:

- `frontmatterOf(request)`: what the frontmatter gives the render: the author's own fields (`doc`), the settings under `md`, each data import with what was loaded (`null` when it failed), and the inputs with their values. An app's inputs panel or a document's properties come from it.
- `componentsOf(request)`: the components the document can name, grouped by the file that defines them, each with its kind, its signature and where the document uses it; and the names it uses that stand for no component. A list of a document's components, or a "go to definition", comes from it.

The processor's README has every type.

The processor's limits

The processor bounds what a document can make it do, whoever wrote it:

- **An expanded tree of 100,000 nodes.** A document whose `each` or template slot would make more gets F034 at the one that crossed the limit, which shows its fallback; what came before stays (spec §8.5). Four nested lists of 100 records are 100 million nodes, so the limit is what keeps such a document from hanging a page.
- **100,000 steps per expression**, a step for each node evaluated and each character or element it builds. An expression that takes more is F024, "the expression takes too much work". A filter takes about six steps a record.

Neither can be changed in `@compmd/processor`.

How long processing takes is measured in a worker in Chrome and WebKit, against budgets: a 4 KB document in 14 ms, a 52 KB one in 48 ms, an edit to an open document in 1 ms (the README's budgets). On an Apple M4, `process` takes 2 to 10 ms for these, and an edit 0.1 to 0.3 ms.

In a worker

Processing a long document takes milliseconds that a page's main thread may not have while a reader types or scrolls. `process` runs the same in a worker, and its request and result are plain data, which `postMessage` copies:

```
// processing.worker.ts
import { init, process, type Request } from "@compmd/processor";

const ready = init();
addEventListener("message", async ({ data }: MessageEvent<Request>) => {
  await ready;
```

```
  postMessage(process(data));
});
```

The page makes it with `new Worker(new URL("./processing.worker.ts", import.meta.url), { type: "module" })`, posts each request, and renders what comes back. For an editor, which processes on every key press, a worker keeps an open document instead, and posts only what changed: chapter 5 shows how the playground does it.

Next: live editing, a page that follows its text as it is typed.

5. Live editing

An editor processes on every key press, so it can't process the whole document each time. This chapter builds a text area with a live page beside it: it keeps the document open, sends each edit, and gives the page only what changed, so the page follows the text as it is typed, and components keep their state. It lists each diagnostic with a button for its fix, and says what is at the caret.

An open document

`new OpenDocument(request)` processes a document once and keeps it. Each `edit(start, end, text)` then processes only the blocks the edit touches, and gives what it did:

```
const opened = new OpenDocument({ ...request, path: "notes.comp.md", files:
{ "notes.comp.md": text }, now, origins: true });
page.render(opened.expanded);
// The reader selected `world` and typed `there`.
const { splice, diagnostics, components } = opened.edit(6, 11, "there");
```

- **An edit** is the range of the old text that changed and what replaced it, in UTF-16 units, as a JavaScript string counts them: a text area's `selectionStart`, CodeMirror's `from` and `to`. One edit at a time, each in the text the one before left.
- **A change** gives `splice`, what the edit did to the expanded tree; every `diagnostics` of the document; and `components`, only when the components the tree names changed.
- **The open document keeps the tree.** `opened.expanded` is always the whole tree, `opened.diagnostics` and `opened.components` the latest.
- `free()` releases the document's memory in the module. Call it when you open the document again, or close it.

Give the open document the editor's own text, `textarea.value` or CodeMirror's `state.doc.toString()`, not the file it was read from: a text area turns line endings into LF, and both must count the same text.

For an edit inside one block, a keystroke takes about 0.1 ms in a 52 KB document and 0.3 ms in a 500 KB one (incremental processing has how, and what an edit to the frontmatter or a definition redoes).

Splices

A splice says which of the tree's top-level nodes changed: the nodes `from` to `to` became `nodes`, and every node after them moved by `lines` lines and `offset` code points. It also has the root's `inputs` and the `edit` that made it, in code points.

```
interface Splice {
  from: number;
  to: number;
  nodes: ExpandedNode[];
  lines: number;
  offset: number;
  inputs?: Record<string, ExpandedInput>;
  edit?: TextEdit;
}
```

A page applies it with `page.splice([splice], change)`, and a sandbox's view with `view.splice`: only the replaced blocks render again. `change` is `{text, diagnostics}`, the text after the edit and its diagnostics, each `{code, line, column}`. A component in a block the splice didn't touch keeps its instance; one in a block it did is matched to its old view by its key.

Keys come from origins. With `origins: true`, every expanded node carries its origin, the range of the text that made it. The page gives each origin an id, follows it through each edit as an editor follows positions, and gives a node whose range lands where an old one went the old one's view. So typing inside a card keeps the card, and its component's state, and typing above it only moves it. Without origins, nodes are matched by place.

Text being typed is often invalid. Given the diagnostics, the page keeps what it showed where the text is broken mid-typing:

- frontmatter that doesn't parse (F001, F033) keeps the last good page, until it parses again;
- a top-level container not closed yet (F004) shows its opener as a paragraph, and what follows stays as it was;
- any other error in a top-level block keeps that block's last good render.

So pass the diagnostics of the processed document itself (those without `file`), as `forPage` does below.

The editor

A text area, a line for the hint, a list for the diagnostics, and the page:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>A live CompMD editor</title>
  </head>
  <body style="display: grid; grid-template-columns: 1fr 1fr; gap: 1rem">
    <div>
      <textarea id="text" spellcheck="false" style="width: 100%; height:
20rem; font-family: monospace"></textarea>
      <p id="hint"></p>
      <ul id="problems"></ul>
    </div>
    <main id="page" style="height: 24rem"></main>
    <script type="module" src="./main.ts"></script>
  </body>
</html>
```

The document it starts with, `dinner.comp.md`:

```
---
md:
  inputs:
    guests: number = 4
---

# Dinner for :value[4]{of=(string(md.input.guests))}

:: number-input[Guests: 4] {bind=guests}

::: box {padding=m border=subtle border-radius=m}
**To buy:** :value[600 g]{of=(string(md.input.guests * 150) + " g")} of
rice, and a lemon.
::: /box

Edit this text, and the page follows it as you type.
```

Try it in the playground.

And `main.ts`. The app is the quick start's, with the same `vite.config.ts`:

```
import { init, OpenDocument, type Diagnostic, type TextEdit } from
"@compmd/processor";
import { codePointOffset, utf16Offset } from "@compmd/render";
import { createSandbox } from "@compmd/sandbox";
import { request, sandbox } from "virtual:compmd";
import text from "./dinner.comp.md?raw";

await init();

const editor = document.getElementById("text") as HTMLTextAreaElement;
const hint = document.getElementById("hint") as HTMLElement;
const problems = document.getElementById("problems") as HTMLElement;
editor.value = text;

// The document, open with origins, which the page's keys come from;
`shown` is the text it has.
let inputs: Record<string, unknown> = {};
let shown = editor.value;
const open = () =>
  new OpenDocument({ ...request, path: "dinner.comp.md", files: {
"dinner.comp.md": shown }, now: new Date().toISOString(), inputs, origins:
true });
let opened = open();

const view = createSandbox(sandbox).open(document.getElementById("page") as
HTMLElement, { components: opened.components });
view.render(opened.expanded);
showProblems(opened.diagnostics);

/** The processor's diagnostics of the text itself, as the page's rules for
invalid text read them. */
const forPage = (diagnostics: readonly Diagnostic[]) =>
  diagnostics.filter(({ file }) => file === undefined).map(({ code, range
}) => ({ code, line: range.start.line, column: range.start.column }));

/** The edit that turned `before` into `after`: what lies between the text
they start and end with in common. */
function editOf(before: string, after: string): TextEdit {
  let start = 0;
  while (start < before.length && before[start] === after[start]) start++;
  let kept = 0;
  while (kept < before.length - start && kept < after.length - start &&
before[before.length - 1 - kept] === after[after.length - 1 - kept])
kept++;
```

```
  // Whole characters: 😀 and 😁 share their first UTF-16 unit.
  const second = (at: number) => /[\uDC00-\uDFFF]/.test(before.charAt(at));
  if (second(start)) start--;
  if (second(before.length - kept)) kept--;
  return { start, end: before.length - kept, text: after.slice(start,
after.length - kept) };
}

/** The text area holds the text after `edit`: the open document takes it,
and the page gets its splice. */
function edited(edit: TextEdit) {
  shown = editor.value;
  const { splice, diagnostics, components } = opened.edit(edit.start,
edit.end, edit.text);
  if (components) void view.components(components);
  view.splice([splice], { text: shown, diagnostics: forPage(diagnostics)
});
  showProblems(diagnostics);
  showHint();
}

editor.addEventListener("input", () => {
  edited(editOf(shown, editor.value));
});

/** Where a line and column, counted in code points as diagnostics count
them, are in `text`, in UTF-16 units. */
function offsetOf(text: string, { line, column }: { line: number; column:
number }): number {
  let at = text.split("\n").slice(0, line - 1).reduce((sum, before) => sum
+ before.length + 1, 0);
  for (let counted = 1; counted < column; counted++) at +=
(text.codePointAt(at) ?? 0) > 0xffff ? 2 : 1;
  return at;
}

/** Each diagnostic, with a button for its fix, which edits the text as
typing does. */
function showProblems(diagnostics: readonly Diagnostic[]) {
  problems.replaceChildren(
    ...diagnostics.map(({ code, message, range, fix }) => {
      const item = document.createElement("li");
      item.textContent =
`${String(range.start.line)}:${String(range.start.column)} ${code}
```

```
${message} `;
      if (fix) {
        const button = document.createElement("button");
        button.textContent = `Write ${fix.text}`;
        button.addEventListener("click", () => {
          const edit = { start: offsetOf(shown, fix.range.start), end:
offsetOf(shown, fix.range.end), text: fix.text };
          editor.setRangeText(edit.text, edit.start, edit.end, "end");
          edited(edit);
        });
        item.append(button);
      }
      return item;
    }),
  );
}

/** What is at the caret: a param and what it takes, a component, or the
diagnostic there. */
function showHint() {
  const at = opened.hintAt(editor.selectionStart);
  const subject = at?.subject;
  if (subject?.kind === "param") hint.textContent = `${subject.param.name}
takes ${subject.param.takes.description}`;
  else if (subject?.kind === "component") hint.textContent =
`${subject.name}, a ${subject.component?.kind ?? "unknown"} component`;
  else hint.textContent = at?.diagnostics[0]?.message ?? "";
}

editor.addEventListener("keyup", showHint);
editor.addEventListener("click", showHint);

// The text and the page, both ways: a click on the page puts the caret
where what was clicked came from, and a
// click in the text shows on the page what the caret is in.
view.on("pointed", ({ source }) => {
  if (source.start.offset === undefined) return;
  const caret = utf16Offset(shown, source.start.offset);
  editor.setSelectionRange(caret, caret);
  showHint();
});
editor.addEventListener("click", () => {
  view.reveal(codePointOffset(shown, editor.selectionStart));
});
```

```
// A reader changed an input: the same text, opened again with the new
value.
view.on("set", ({ input, value }) => {
  inputs = { ...inputs, [input]: value };
  opened.free();
  opened = open();
  view.render(opened.expanded, { text: shown, diagnostics:
forPage(opened.diagnostics) });
});
```

Run it, and type in the text area: a line typed at the end shows on the page as you type it. Change the guests to 6 on the page, and the heading says *Dinner for 6* and the box *900 g*, while the text stays as you wrote it. Write `border=subtl`: the list shows F009 with a *Write subtle* button, which mends the text as typing would. Put the caret in `border`, and the line under the text says what it takes. Click *To buy* on the page, and the caret goes to that line of the text; click in the heading's line of the text, and the page outlines the heading for a moment.

The text and the page, both ways

The page knows which text made each thing it shows, from the origins (`origins: true`), so an editor can go from one to the other. A sandbox's view does it the same in both page modes, though in frame mode the page is in a frame the host can't reach:

- **`view.on("pointed", ({ source }) => …)`**: the reader clicked on the page, on what came from `source`, a range of the text: `start` and `end`, each with its `line`, `column` and `offset`. The page reports clicks only once you listen. The click goes on as usual, so a link still comes as `link`.
- **`view.reveal(offset)`** scrolls the page to what the text at `offset` drew, as little as it takes, and outlines it for a moment in the theme's accent (the class `compmd-revealed`).
- **`view.rectsAt(offset)`** resolves with the boxes of what it drew, in the page's coordinates, for an editor that draws its own marks beside or over the page.
- **`view.lines(from)`** gives each line of the text that draws something, with the boxes it shows in, from the container's top left corner: a paragraph's later lines where their text wraps in, each line of a code block, a component on the line that uses it. It is an async generator that measures each line as your loop reaches it, so a gutter of line numbers beside the page stops at the bottom of what shows; `view.on("layout", …)` says when to draw it again, after a render or when something on the page changed size.

```
view.on("layout", async () => {
  const numbers = [];
  for await (const { line, rects: [first] } of view.lines()) {
    if (first.y > page.scrollTop + page.clientHeight) break;
    numbers.push(Object.assign(document.createElement("span"), {
textContent: String(line), style: `position: absolute; top: ${first.y}px`
}));
  }
  gutter.replaceChildren(...numbers);
});
```

Offsets count code points, as the processor's do, and a text area's count UTF-16 units: an emoji is one code point and two units. `codePointOffset(text, utf16)` and `utf16Offset(text, codePoint)`, from `@compmd/render`, convert one to the other in the text the page shows (`shown` here).

What a click gives follows the origins: a word in `*em*` gives the emphasis, text a `value` made the node holding it, anything a `compmd` component's body drew the line that uses the component, and a host-language component's own drawing the component. An offset gives the innermost thing that came from there, all of them for an `each`, whose items share their template's text. `@compmd/render`'s README has the whole table, and `page.sourceOf(node)` and `page.elementsAt(offset)`, the same for a page you render yourself; where each line is has the rules for lines, and `page.lines()` with `onLayout`.

The example reveals on a click in the text only. The playground reveals on every move of the caret the reader makes, by keys too, but not while they type, where the outline would follow every key press.

Edits from a text area

`editOf` finds the edit by comparing the text before and after an input: what lies between what both start and end with. That is always a right edit, whatever the input was: typing, deleting, pasting, undo. It keeps a character that takes two UTF-16 units, such as an emoji, whole.

It can place an edit where the reader didn't make it: cutting `:: notes[Notes]` from in front of `::: card` leaves `::` in front of `card`, and a comparison alone puts the edit inside the card's opener. The page's keys follow the edit, so the card may lose its state. The playground also reads the selection before each input: when the text outside it didn't change, the selection is what was replaced. An editor with a model of its own, such as CodeMirror, has the edits already, in each transaction's changes, in order.

A diagnostic's fix is an edit too. Its range is in lines and code point columns, so `offsetOf` turns it into UTF-16 positions in the text; the fix then goes through `edited` like a key press. A fix is of the text its diagnostic was found in: here every edit is processed before the next, so they are always the same text.

Hints at the caret

`hintAt(offset)` says what is at a UTF-16 position of the text as the last change left it (positional hints), or `undefined` in plain prose:

Field	What it is
`range`	the text the hint is about, a name, a key, a value, in lines and code point columns; a completion replaces it
`subject`	what it is: a `component`, the `params` of one, a `param` with what it `takes`, a `slot`, a `path` in an interpolation or an expression, a `function`, a `frontmatter` key, or a `type`
`completions`	what may be written at `range`, each `{label, text, detail?}`
`diagnostics`	the diagnostics whose range holds the position

A hint is a lookup, not a pass, so it is cheap enough for every caret move. The example shows a param's `takes.description`, the words its F009 uses, so a hint and a diagnostic never say different things. An editor with completion as you type lists `completions`, and writes the one chosen over `range`, as a fix is written.

Keeping a page in step with a text

The example keeps three things in step: the text area's text, the open document's (`shown`), and the page's. Each input changes the first, `edited` the other two, in that order, before the next input. Two things open the document again instead of editing it:

- **An input changed.** The text is the same, so the new document's tree has the same origins, and `view.render(tree, change)` with the same text keeps every block's keys.
- **Anything else in the request changed**, a file the document imports, a pack, data you fetched: an open document keeps what it was opened with, so open it again with the new request. After an edit to the frontmatter that names a URL or a `host:` source the request doesn't hold, the document shows its fallback (or F026, without one) until you fetch it, and `opened.sources()` lists it, with `opened.reload({remote, host})` to give it (chapter 6).

The playground's way: a worker

The example processes in the text area's input handler, which is fine for a short document. The playground never does: a long document's frontmatter edit takes milliseconds, and the key press would be painted that much later. Its editor sends each edit to a worker, which keeps the document open, and the page gets the splices from it.

The worker holds an `OpenDocumentJson`, which is `OpenDocument` without the tree: each change's splice comes as the module's JSON text, which the worker posts and the page passes to its view unread. Posted text is copied in one piece, and the renderer parses it once, in the page's frame.

```
// editor.worker.ts
import { init, OpenDocumentJson, type Request, type TextEdit } from
"@compmd/processor";

const ready = init();
let opened: OpenDocumentJson | undefined;
addEventListener("message", async ({ data }: MessageEvent<{ request:
Request } | { edits: TextEdit[] }>) => {
  await ready;
  if ("request" in data) {
    opened?.free();
    opened = new OpenDocumentJson(data.request);
    postMessage({ opened: true, changes: [opened.changes()] });
  } else if (opened) {
    const document = opened;
    postMessage({ opened: false, changes: data.edits.map(({ start, end,
text }) => document.edit(start, end, text)) });
  }
});
```

`changes()` gives what the last change did; after opening, the whole tree as one splice, of an empty tree. Each change is `{splice, diagnostics, inputs, components?}`, with the inputs apart, since the page doesn't read the splice. The page gives them all to its view at once: `view.splice(changes.map(({ splice }) => splice), { text, diagnostics }, { replace: opened, inputs: last.inputs })`, where `replace` says the first splice is of an empty tree, as after opening.

While the worker works, the reader types on. The playground's editor doesn't queue a message per key press: edits made while the worker is busy wait, and go together in the next message, each with positions in the text the one before left, and the worker applies them in order. A host may also merge them into one edit that covers them all. Each message

carries the caret, so each answer brings the hint at it, of the text just processed, and a caret that moves while nothing is typed asks for the hint alone.

Next: data and inputs, where a document's data comes from, and the values a reader changes.

6. Data and inputs

A document reads data from files beside it, from URLs, and from your app itself, and its inputs are values a reader changes. The processor fetches none of it: your app gives it everything, so it decides what a document may reach. This chapter builds a page of orders that reads your app's own records, a file beside the document, and a status the reader picks, kept in the page's address.

How a document declares its data and inputs is the Author's Guide's subject (data from files, inputs the reader changes); here is what your app does with them.

Data files

A data import's `from` is a relative path, a URL, or a `host:` source:

`from`	What it reads	What your app gives
`./delivery.csv`, `./team.yaml`, `./site.comp.md`	a file of the document's folder	the file, in `files`
`https://api.example.com/status.json`	a URL, from an origin you allow	its text, in `remote`, by URL
`host:orders?status=open`	your app's own data	the value, in `host`, by source

Files are the simplest: they are in `files` like the document (chapter 4), and loaded by extension, `.csv` and `.tsv` as lists of records, `.yaml`, `.yml` and `.json` as their values, a `.comp.md` as its author's frontmatter fields. A file that isn't there, or doesn't parse, is F026; a record that doesn't match the import's `schema` is F027 and is dropped.

The example

`orders.comp.md` reads the orders of one status from the app, and the delivery times from a file:

```
---
md:
```

```
  inputs:
    status: open | shipped = open
  data:
    - from: host:orders?status={{md.input.status}}
      as: orders
      schema: {id: string, customer: string, total: number}
      fallback: ./sample-orders.json
    - from: ./delivery.csv
      as: delivery
      schema: {zone: string, days: number}
---

# Orders

:: select-input[Status: open] {bind=status}

::: each {in=(md.data.orders) as=o join=list}
- **{{o.id}}**, {{o.customer}}: {{o.total}} EUR

:: #empty
No orders.
::: /each

:value[2 orders]{of=(string(size(md.data.orders)) + " orders")} in all.

## Delivery

::: each {in=(md.data.delivery) as=d join=list}
- {{d.zone}}: {{d.days}} days
::: /each
```

Try it in the playground, which has no orders of yours: it shows the fallback.

`delivery.csv`:

```
zone,days
Europe,2
Elsewhere,6
```

`sample-orders.json`, the fallback, for a host that doesn't provide `host:orders`:

```
[
  { "id": "S-1", "customer": "A sample customer", "total": 10 }
```

```
]
```

The page, as in the quick start:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Orders</title>
  </head>
  <body>
    <main id="page" style="height: 28rem"></main>
    <script type="module" src="./main.ts"></script>
  </body>
</html>
```

And `main.ts`, with the same `vite.config.ts` as the quick start:

```
import { init, OpenDocument, sources, type Request } from
"@compmd/processor";
import { createSandbox } from "@compmd/sandbox";
import { request, sandbox } from "virtual:compmd";
import delivery from "./delivery.csv?raw";
import text from "./orders.comp.md?raw";
import samples from "./sample-orders.json?raw";

await init();

/** The app's own records, which its documents read as `host:orders`. */
const ORDERS = [
  { id: "A-1041", customer: "Ada", total: 42.5, status: "open" },
  { id: "A-1042", customer: "Grace", total: 18, status: "open" },
  { id: "A-1039", customer: "Edsger", total: 7.25, status: "shipped" },
];

/** What the app gives a `host:` source, without `host:`; `undefined` for
one it doesn't provide. */
function hostData(source: string): unknown {
  const [name, query = ""] = source.split("?");
  if (name !== "orders") return undefined;
  const status = new URLSearchParams(query).get("status");
  return ORDERS.filter((order) => order.status === status).map(({ id,
customer, total }) => ({ id, customer, total }));
```

```
}

// The inputs' values, kept in the page's address: anyone can write one,
and the processor checks each.
let inputs: Record<string, unknown> = Object.fromEntries(new
URLSearchParams(location.search));

/** The document with the inputs' values, and the host data that render
needs, which depends on them. */
function open(): OpenDocument {
  const files = { "orders.comp.md": text, "delivery.csv": delivery,
"sample-orders.json": samples };
  const asked: Request = { ...request, path: "orders.comp.md", files, now:
new Date().toISOString(), inputs };
  const host: Record<string, unknown> = {};
  for (const source of sources(asked).host) {
    const data = hostData(source);
    if (data !== undefined) host[source] = data;
  }
  return new OpenDocument({ ...asked, host });
}

let opened = open();
const view = createSandbox(sandbox).open(document.getElementById("page") as
HTMLElement, { components: opened.components });
view.render(opened.expanded);

// The reader picked another status: keep it, and open the document again,
with the data it now needs.
view.on("set", ({ input, value }) => {
  inputs = { ...inputs, [input]: value };
  history.replaceState(null, "", `?${new
URLSearchParams(Object.entries(inputs).map(([name, given]) => [name,
String(given)])).toString()}`);
  opened.free();
  opened = open();
  view.render(opened.expanded);
});
```

Run it: the page lists Ada's and Grace's orders, *2 orders in all*, and the two delivery zones. Pick *shipped*, and it lists Edsger's, and the address ends in `?status=shipped`, so a reload, or a link, opens the same view.

The host's data

A `host:` source names data your app provides: the processor gives it nothing but the text after `host:`, here `orders?status=open`, and you decide what that means. It is the one way a document reads your app's own records, and it reads only what you answer.

- **`sources(request).host`** lists the sources a render needs, without `host:`, after interpolating the inputs. Since `from` may hold an input, the sources change with the inputs: that is why `open` asks again each time.
- **You answer in `request.host`**, by source, with any JSON value. It lands in `md.data` like a file's, checked against the import's `schema` (F027 drops a record that doesn't match), so give what the schema says, and only the fields it names.
- **A source you don't provide** is left out. The document then gets its `fallback`, with no diagnostic, or, without a fallback, `null` and F026: an `each` over it shows its `empty` slot.
- **Access is yours.** The document carries no credentials, and a reader sees only what your app answers for that reader. A shared document can ask for any source by name, so answer each as you would any request from that reader.

When your data changes, open the document again with the new data: rendering is a function of what the request holds (spec §8.5).

Data from URLs

A document may read an `https://` URL, but only from an origin in `allowedOrigins`. A document can't grant itself access: any other origin is F028, and the import gets its fallback.

```
md:
  data:
    - from: https://status.example.org/status.json?region=
{{md.input.region}}
      as: status
      fallback: ./status.json
```

What the spec asks of a host that fetches while a reader views a document (spec §8.6 and §15), which is your app, since the processor fetches nothing:

- **The reader decides.** The first time a document needs an origin the reader hasn't decided on, ask, and remember the answer. Until the reader allows it, it isn't in `allowedOrigins`. Origins your app configures need no question.
- **No credentials:** no cookies or other credentials with a document's fetches, except those you configured for that origin, and no `Referer`.

- **The origin learns** the reader's address, the time, and the input values in the URL. A shared cache, by URL, tells it less.

`loadData`, from `@compmd/sandbox`, does all three in your page. `sources(request)` lists the URLs a render needs, with the inputs' values in them: `remote` from the allowed origins, `disallowed` from the others. Give both to `loadData`, which asks the reader about new origins, fetches from the allowed ones without credentials or a `Referer`, and gives `{allowedOrigins, remote}` for the request:

```
import { loadData } from "@compmd/sandbox";

const configured = ["https://status.example.org"];
const asked = { ...request, path, files, now, allowedOrigins: configured };
let opened = new OpenDocument(asked);            // meanwhile, a new origin
is F028 and shows its fallback
view.render(opened.expanded);
const { remote, disallowed } = sources(asked);
const loaded = await loadData([...remote, ...disallowed], { configured,
document: path });
opened.free();
opened = new OpenDocument({ ...asked, ...loaded }); // origins are checked
when a document is opened
view.render(opened.expanded);
```

- **Configured origins** are never asked about. Every other origin is asked about once, however many documents need it, and the answer, allow or deny, is kept per origin for every document.
- **The question** is a plain dialog in your page, never in a frame, whose code could draw a question of its own: *This document wants to load data from api.example.com. Allow?*, saying what the site learns. Give `ask: (origin, document) => Promise<"allow" | "deny">` to ask in your own UI; `originQuestion(origin)` gives the words.
- **The answers** are kept in `localStorage` (`localAnswers()`), or in a store you give (`answers`), such as one on your server for signed-in readers. `localAnswers().list()` and `forget(origin)` let a reader see and change them, on a settings page of yours.
- **A URL that can't be fetched** is left out: the document shows its fallback, or F026 without one.
- **Your own fetching:** a host that fetches through its server, or caches by URL, calls `allowOrigins(urls, options)`, which asks and gives the allowed origins alone. Fetch only from those.

The playground does this: try it with a document that reads a pack's latest release from npm's registry, which the playground doesn't list.

A document rendered once, at build time or on a server, has no reader to ask: give `loadData` an `ask` that denies, and it fetches only from your configured origins, so nothing reaches another origin.

Inputs and their values

An input is a typed value the reader changes, with a default, so a document always renders. Its value comes from `request.inputs`, by name:

- **A value that doesn't fit is the default.** `?status=lost` opens the open orders, since `lost` isn't one of the enum's members; a number input given `"x"` gets its default. No diagnostic says so. So `inputs` can come from anywhere, a URL, storage, a reader's link, without checking it first.
- **The root of the expanded tree lists the inputs**, `expanded.inputs`, each with its `type`, an enum's `members`, and the `value` the render used. An inputs panel of your own, as the playground's, reads it; so does `frontmatterOf(request).inputs`.
- **Where values are kept between renders is yours.** The example keeps them in the page's address; another app keeps them per reader, or not at all.

The components that set them

Two kinds of component change an input:

- **The input components**, built in: `date-input`, `number-input`, `text-input`, `select-input` for an enum, and `toggle-input` for a boolean. Each binds one input, `{bind=status}`, and renders a control.
- **A component in code** whose signature declares a param of type `input<…>`, which the document binds to an input (`:: map[Map] {selected=region}`). Its code calls `context.set(param, value)` (chapter 10). An input such a component can change can't appear in a data import's `from` (F031), so component code can't send data out through a URL.

Either way, the sandbox tells your page with the view's `set` event, `{input, value}`, named by the input, not the param. It passes on only a `set` of one of the document's inputs; a frame that asks to set anything else is refused. In host mode, a trusted component's `set` comes the same way.

Processing again with new values

A new value is a new render: open the document again with it, as the example does in its `set` listener. Rendering is a function of the request, so the page then shows what a fresh

render with that value gives, the data included: here a new status is a new `host:` source, and `open` answers it before it processes.

The text is the same, so the new tree has the same origins, and `view.render` brings the page up to date in place: a component keeps its state across a new value. An editor that keeps its document open for edits does the same (chapter 5). An open document that only needs data it fetched again, with the same inputs, takes `opened.reload({remote, host})` instead, which loads and expands again and gives a splice of the whole tree.

Next: rendering and styling, the page itself: its look, light and dark, and its links.

7. Rendering and styling

A document's page should look like your app: its fonts, its colours, light or dark as your app is. This chapter builds a page styled with your app's stylesheet and theme tokens, with a switch between light and dark, that shows a link's address before it opens it, and shows beside it what a plain Markdown renderer makes of the same text. It ends with `@compmd/render` alone, for pages that need no sandbox.

Where the page is drawn

`@compmd/render` draws an expanded tree as HTML and CSS, the reference rendering of the built-ins (its README): a `box` is a `<div class="compmd-box">` with its layout params as CSS, an input component a `<label>` with its control, an alert a `<div role="note">`. The sandbox uses it too: in frame mode the page is drawn by `@compmd/render` inside the strict frame, in host mode in your page.

So there are two places to style:

	Frame mode (the default)	**Host mode**
The page is in	a strict frame, a document of its own	your page
Your page's CSS	doesn't reach it	applies, as to any of your elements
Your stylesheets reach it by	`view.styles(urls)`	your page's own `<link>`s
Theme tokens	`view.theme(tokens)`	`view.theme(tokens)`
Its colour scheme	the container's `color-scheme`, sent to the frame	the container's `color-scheme`

Components in frames of their own, in either mode, get the same: the sandbox gives every frame your stylesheets, the tokens and the scheme.

What's still loading. A use of a pack your app is still getting (chapter 9) is its fallback in `<div class="compmd-pending" aria-busy="true">`, a `<span>` in a line of text, a little faded. Screen readers say it is busy, and that it changed when the pack arrives. Restyle `.compmd-`

`pending` in your stylesheet to show loading your way, a shimmer, say; the fallback stays readable whatever you do.

The example

A document with boxes that use the theme's tokens, and a link:

```
---
team: the docs team
---

# Welcome aboard

::: box {padding=m border-radius=m background=accent}
Your first week, from :value[the docs team]{of=(md.doc.team)}.
::: /box

::: row {gap=m}
::: box {padding=m border=subtle border-radius=m}
**Monday:** read [the spec](https://compmd.hakanserce.com/spec/).
::: /box
::: box {padding=m border=subtle border-radius=m}
**Tuesday:** write a first document.
::: /box
::: /row
```

Try it in the playground.

The app's stylesheet for its documents, `public/page.css`, which Vite serves as `/page.css`:

```
/* The app's look for its documents, which the page's frame loads: served
from the app, as the frames' policy allows. */
body {
  font: 16px/1.6 Georgia, serif;
}

h1 {
  color: var(--compmd-accent);
}
```

The page: a switch, the document's page, a frame for the plain rendering, and a dialog for links:

```
<!doctype html>
<html lang="en" style="color-scheme: light">
  <head>
    <meta charset="utf-8" />
    <title>A styled CompMD page</title>
  </head>
  <body style="background: Canvas; color: CanvasText">
    <button id="scheme">Dark</button>
    <main id="page" style="height: 20rem"></main>
    <h2>What a plain renderer shows</h2>
    <iframe id="plain" sandbox="" title="The plain rendering" style="width:
100%; height: 14rem; border: 0"></iframe>
    <dialog id="link">
      <p>This link goes to <output></output></p>
      <form method="dialog">
        <button value="cancel">Stay here</button>
        <button value="open">Open it</button>
      </form>
    </dialog>
    <script type="module" src="./main.ts"></script>
  </body>
</html>
```

And `main.ts`, with the same `vite.config.ts` as the quick start:

```
import { init, OpenDocument, plainHtml } from "@compmd/processor";
import { createSandbox } from "@compmd/sandbox";
import { request, sandbox } from "virtual:compmd";
import text from "./welcome.comp.md?raw";

await init();

const opened = new OpenDocument({ ...request, path: "welcome.comp.md",
files: { "welcome.comp.md": text }, now: new Date().toISOString() });
const view = createSandbox(sandbox).open(document.getElementById("page") as
HTMLElement, { components: opened.components });

// The app's look: its stylesheet in the page's frame, and the theme tokens
the built-ins' styles read.
view.styles([new URL("page.css", location.href).href]);
view.theme({ "--compmd-accent": "#0b7285", "--compmd-space-m": "1.25rem",
"--compmd-radius-m": "12px" });
view.render(opened.expanded);
```

```
// Light or dark is the page's `color-scheme`; after the app switches it,
the view reads it again.
const scheme = document.getElementById("scheme") as HTMLButtonElement;
scheme.addEventListener("click", () => {
  const dark = document.documentElement.style.colorScheme !== "dark";
  document.documentElement.style.colorScheme = dark ? "dark" : "light";
  scheme.textContent = dark ? "Light" : "Dark";
  view.scheme();
});

// A link the reader clicked in the page: show where it goes, and open it
only when the reader says so.
const dialog = document.getElementById("link") as HTMLDialogElement;
view.on("link", ({ url }) => {
  const target = URL.parse(url, location.href);
  if (!target || !["https:", "http:", "mailto:"].includes(target.protocol))
return;
  (dialog.querySelector("output") as HTMLOutputElement).value =
target.href;
  dialog.onclose = () => {
    if (dialog.returnValue === "open") window.open(target.href, "_blank",
"noopener");
  };
  dialog.showModal();
});

// What a plain Markdown renderer shows of the same text, in a frame that
runs nothing.
(document.getElementById("plain") as HTMLIFrameElement).srcdoc =
plainHtml(text);
```

Run it: the heading is in Georgia and teal, the first box has the accent's background with rounder corners. *Dark* turns the page and its frame dark. A click on *the spec* opens the dialog with the link's address, and nothing opens until the reader says so. Under the page is the same text as GitHub would show it.

Theme tokens

The built-ins' tokens are CSS custom properties (spec §9.4). `padding=m` is `var(--compmd-space-m, 1rem)`, `background=accent` a mix of `--compmd-accent`, so a theme changes them without a document changing:

Token	Custom property	Default
`s`, `m`, `l` spaces (`padding`, `gap`)	`--compmd-space-s`, `-m`, `-l`	`0.5rem`, `1rem`, `2rem`
`background=subtle`, `strong`	`--compmd-surface-subtle`, `-strong`	the text colour at 6%, 12%
`background=accent`	`--compmd-surface-accent`	`--compmd-accent` (`#0969da`) at 12%
`border=subtle`, `strong`	`--compmd-border-subtle`, `-strong`	a 1px line of the text colour at 20%, 45%
`border-radius=s`, `m`, `l`, `full`	`--compmd-radius-s`, `-m`, `-l`, `-full`	`4px`, `8px`, `16px`, `9999px`

The defaults are made from the text colour, so they suit light and dark text with no theme at all. `view.theme(tokens)` sets the `--compmd-*` properties you give on the page's root and in every frame, in place of the last ones; other names are ignored. A frame's document doesn't inherit your page's custom properties, which is why the tokens go through the view and not your stylesheet.

The host's stylesheets

`view.styles(urls)` gives the page's frame, and every component's frame, your stylesheets, in place of the last ones (`createSandbox`'s `stylesheets` option gives the first). The frames' policy loads stylesheets only from the origin that serves the frame documents, your app's with `@compmd/vite`, and a strict frame's fonts too, so serve them there; a stylesheet from a CDN doesn't load.

A stylesheet in a frame styles what the renderer writes: elements (`h1`, `table`), and the built-ins' classes, `compmd-box`, `compmd-input`, `compmd-alert`, `compmd-footnotes` and the others in the reference rendering. Each styled block also gets a numbered class, `compmd-b3`, which carries its own params' CSS: don't style those. A document can't style anything itself: its params are a closed vocabulary, and no param passes CSS through.

In host mode the page is in your page, and your page's CSS reaches it like any of your elements. `view.styles` then only reaches the frames of components that aren't trusted.

Light and dark

The page takes its colour scheme from where you put it: the container's `color-scheme`, and the reader's preference when it allows both (`light dark`). The view reads it when it opens, at

each render, when the reader's preference changes, and when printing starts and ends, and sends it to every frame, which sets it on its document. So a frame stays transparent on your page, its text in your page's colours.

An app that switches its own theme, with a class or a style as the example does, calls `view.scheme()` after it: the view reads the scheme again, and the frames and components follow. A component that draws with CSS colours needs nothing more; one that takes its colours when it draws, a diagram or a canvas, reads `context.scheme`, and gets an update when it changes (chapter 10).

Code blocks

A code block shows as code when no renderer draws it: a language nothing renders, or a code renderer's block whose component is missing or fails, which shows its code block as its fallback. The sandbox highlights it, in both page modes, with Shiki in GitHub's light and dark themes, in about 60 popular languages, CompMD's own syntax in `md` and `compmd` blocks among them; any other language stays plain (the sandbox's README lists them). Each language loads the first time a block of it shows, so a page without code blocks loads nothing of it. The colours follow the page's light or dark as frames do, with nothing more to call.

To style it, a stylesheet styles `pre` and `code` as it would any page's: the tokens are spans in the `<code>`, each with its colour.

Links and the `link` event

A component in a sandbox can't navigate or open a window (spec §15, rule 4). When a reader clicks a link in a frame, the frame doesn't follow it: it sends the view `link`, `{url}`, with the `href` as written, and your app decides. Only a reader's real click sends it; a link to a place in the page, a footnote, stays in the frame.

- **In frame mode** every link of the page is in the frame, links written in Markdown too, so every click on one comes as `link`.
- **In host mode** the page's own links are your page's, and open as your other links do; `link` comes from the frames of components.

The spec's rule is to show the reader the full address before opening it: a document from anyone can write `[your bank](https://bank.example.evil/)`. The example resolves the address against the page, opens only `https:`, `http:` and `mailto:`, shows it in a dialog, and opens it in a new tab with `noopener` only when the reader asks. Never assign the URL to `location` unchecked: a component may send a `javascript:` URL.

What a plain renderer shows

A CompMD document is a Markdown document, and GitHub, an editor's preview or any CommonMark renderer shows it without CompMD: the component lines as text, the content inside containers as Markdown, a leaf's fallback in its line (spec §14). `plainHtml(text)` gives that rendering, CommonMark and GFM, as HTML, with raw HTML escaped. Writers use it to see how their document degrades; the example shows it under the page.

The example puts it in a frame with `sandbox=""`, which runs no script and opens no link in your page, rather than in your page: what it shows is the document's, which may be anyone's.

Without the sandbox: pages and regions

`@compmd/render` draws a tree without `@compmd/sandbox`, in your page:

```
import { createPage } from "@compmd/render";

const page = createPage(container, {
  components: { stars: mountStars },                    // your own
components, by name
  onSet: (input, value) => openAgain({ [input]: value }),
  tokens: { "--compmd-accent": "#0b7285" },
});
page.render(opened.expanded);                           // or
page.splice(splices, change), as in chapter 5
```

It runs only the components you give it in `components`; every other component the document names shows its fallback, and `onError` hears of it as `missing`. Without a sandbox, though, the page is in your page: its links are your page's, and its images load from wherever they point. So use it alone for documents you trust as your own code, and the sandbox for any other.

Its options, beside `components`, `onSet`, `onError` and `tokens`: `code` renders the code blocks of a language (`{ mermaid: (source, target) => … }`), `math` renders TeX (`(tex, { display }, target) => …`), `highlight` highlights the code blocks that show as code (`(code, language, target) => …`, or the sandbox's `shikiHighlight()`), `wikiLink` says where a wiki link points, and `host` is what components get as `context.host`. The page has `theme(tokens)` and `scheme()` as a view has, and `destroy()`.

A **region** draws any list of nodes, for an editor that shows blocks in its own widgets: `createRegion(target, { components, stylesheet: page.stylesheet }).render(nodes, { inputs, headingLevel })`. Breakpoints are container queries, so a region outside a page

needs a container to follow: the class `compmd-container` on its target, or on an element around it. Without one, `onError` reports `no-container` once. A region in a sandbox, one use per widget, is `HostRuntime`'s `mountUse` (chapter 11).

Next: components and trust, what the components in a document may reach, and who decides.

8. Components and trust

A document can bring code: a component it defines in a fence, or one from a pack it imports. This chapter is what that code may do in your app, at each of the three trust levels, who decides a level, and what the sandbox guards against. The chapters after it put it to work: packs from npm, your app's own components, and the sandbox in depth.

Three levels

Every component written in code (JavaScript, TypeScript, JSX or Svelte) runs at one of three levels. Components written in CompMD run no code, so they need none.

Level	Where it runs	What it can reach	For
strict (the default)	a frame with an opaque origin and no network	its params, its slots, its box, and `set` for the inputs bound to it	code from documents, and every pack your configuration doesn't list for more
permissive	a frame of its own, one per use	the same, plus the network to the origins you list for its pack, and `host` if you give it	packs you list, such as a map that loads its tiles
trusted	your page	what your page's own code can, and `host` if you give one	your app's own components, in a folder of your app, built with it

A level belongs to a pack, never to a component or a use: every component of a pack runs at the pack's level.

Strict: the default

Strict code gets the host contract (spec §7.4) and nothing else. In plain words, a strict component:

1. **sees only what it is given**: its params, its slots and its box, not the rest of the document, its data or your page;

2. **has no network**: no requests, images, fonts, stylesheets or sockets, to anywhere; its code comes from its pack, and its libraries from the frames;
3. **can't reach anyone's session**: no cookies, storage or credentials, yours or the reader's;
4. **can't navigate or open windows**: a link it shows opens only when the reader clicks it, and your page hears of the click first (`view.on("link", …)`);
5. **can be stopped**: a frame that stops answering is removed, and its use shows its fallback (W118).

That is why any document can import any pack: a pack nobody reviewed still can't send what it is given anywhere, because it has nowhere to send it. Private data you pass a strict component as a param stays in its frame.

Permissive: what you list, and nothing more

A permissive pack keeps rules 3 to 5, and gets only what your configuration lists for it:

- **the network to its origins**, each `https://host[:port]`, added to its frame's policy before any of its code loads;
- **`host`**, the calls your app offers it, when you give it `"host": true` (chapter 10).

Each permissive use gets a frame of its own, so the network one pack is given reaches no other pack's code. A permissive frame can hold strict frames, for strict components in its slots, and nothing else.

Trusted: your own code

A trusted component is your app's: built, reviewed and shipped with it, so a sandbox would only protect your app from itself. It mounts in your page, as any of your code does, and gets `host` as a permissive one does, when your app gives one: the same calls and events, so a component works at either level (chapter 10). Only a folder of your app can be trusted, never a pack from a registry: `@compmd/vite` stops the build if the configuration says otherwise.

Its params still come from documents, which may be anyone's. A trusted component treats them as untrusted input: no HTML from a param, URLs checked, and nothing done outside the document unless the reader asks for it (chapter 10).

Who decides

Your configuration, and only it. `compmd.config.json` lists the packs that get more than strict:

```
{
  "packs": {
    "@acme/maps": { "trust": "permissive", "network":
["https://tiles.example.com"] },
    "@acme/reviews": { "trust": "permissive", "host": true },
    "./components": { "trust": "trusted", "ambient": true }
  }
}
```

The plugin turns it into the processor's policy, the processor marks each component the document uses with its pack's level (`trust` in `components`), and the sandbox runs each where its level says. The sandbox never reads the configuration itself, so a level is decided in one place.

Never a document. Nothing in a document raises a level: not its frontmatter, not an import, not a param. A component a document defines itself is always strict, in any app. This one runs in the strict sandbox wherever it is opened:

```
```compmd component=shout signature
params:
  text: string
```

```jsx component=shout runtime=react@19
export default function Shout({ text }) {
  return <strong>{text.toUpperCase()}</strong>;
}
```

:: shout[Hello] {text="Hello"}
```

Never a pack. A pack can't name its own level, its own origins or a registry. Listing a pack gives it more wherever your app gets it, installed or fetched.

Where the page renders

The sandbox renders a document in one of two modes, and your app chooses, with `createSandbox`'s `page` option:

	`page: "frame"` **(the default)**	`page: "host"`
The page	one strict frame per document	your page
Strict components	in the page's frame, sharing it	a frame per use
Permissive components	their fallbacks	a frame per use
Trusted components	their fallbacks	inline, in your page
For	documents from anyone, with strict packs	an app with components of its own, or permissive packs

A permissive or trusted component that can't run where it is shows its fallback, and the view's `error` event says why (`not-here`). Chapter 11 has both modes in depth.

The threat model, in plain words

Who might misbehave, and what they must not reach:

Code	Comes from	Must not reach
Your page, the sandbox's host side, trusted components	your app	(trusted)
Permissive code	a pack you list	origins other than its pack's; your page and session; other frames; `host`, unless you give it
Strict code	documents, unlisted packs	any network; your page and session; other documents' frames; code of a higher level
The document	anyone	a higher level for any of its code

The sandbox stops each threat with something the browser enforces: the frames' opaque origin, their content security policies, which a frame can only narrow, frames that only nest lower levels, a hash check for every built module, and a ping for every frame that runs components. Each one is a browser test in Chrome and WebKit. The design has the full list, T1 to T10 (running components in the sandbox).

What a browser can't stop, and so the sandbox can't either:

- **WebRTC**, which no content security policy covers.
- **Requests to the files the frames are served from.** A frame loads its scripts and styles from where you serve `compmd/`, so a request there can carry data in its query string to your own server. Strict code can reach nobody else this way. If you mind, serve `compmd/` from an origin of its own, with no logs.
- **A frame that spins in WebKit.** Chrome stops it and removes it; WebKit runs these frames in the page's process, so a frame that never lets go freezes the page with it.
- **What a component draws in its own box**, such as a fake password prompt. What is typed into it has no way out, and the frames are placed as the document's content, never over your interface.

Code that acts with someone's access, a trusted component or one given `host`, has one more rule (spec §15): the document that uses it may be someone else's, so it acts outside the document (sends, writes, shares, pays) only when the reader asks, after showing what it will do.

Next: packs from npm, what your app installs or fetches, and what your configuration gives them.

9. Packs from npm

Your documents can use packs published on npm, in two ways: you install a pack with your app, or your app fetches the packs a document imports from a registry while it runs. Either way, your configuration says what each pack may do, and a document can never ask for more. This chapter builds a page that draws a diagram with a pack it never installed.

Installing a pack

```
npm install @comp-markdown/mermaid
```

That's all. `@compmd/vite` finds every installed scoped package that is a CompMD pack (its `package.json` names its document in a `compmd` field, or it has an `index.comp.md`) and gives it to the processor in `virtual:compmd`'s `request.packs`, with its integrity from your lock file, for documents that pin it. It serves the pack's built modules from your app, under `compmd/packs/`, and tells the sandbox where each one is, with its file's hash. A document that imports `@comp-markdown/mermaid@0` now gets it.

Every installed pack is available to every document, in the strict sandbox, with nothing to configure. The configuration is for what goes beyond that.

The configuration

`compmd.config.json`, next to your app's `package.json`, is your app's policy for packs: never a document's, and never a pack's. Its keys:

```
{
  "$schema": "./node_modules/@compmd/vite/compmd.config.schema.json",
  "packs": {
    "@kit/charts": { "version": "^2.0.0" },
    "@acme/maps": { "trust": "permissive", "network":
["https://tiles.example.com"] },
    "@acme/reviews": { "trust": "permissive", "host": true },
    "./components": { "trust": "trusted", "ambient": true }
  },
  "imports": "any",
  "deny": [{ "pack": "@bad/pack", "version": "<2", "reason": "a flaw found
```

```
in 1.x" }],
  "maxPackSize": "10 MB",
  "libraries": ["d3"],
  "registries": { "*": "https://registry.npmjs.org/" },
  "fetch": "host"
}
```

Key	What it is
`packs`	the packs that get more than the defaults, by name, or by folder for a pack in your app (chapter 10)
`packs.*.trust`	`strict` (the default); `permissive`, a frame of its own with the network to the origins you list, and `host` if you give it; `trusted`, in your page, for a folder of your app only (chapter 8)
`packs.*.network`	the origins a permissive pack may reach, each `https://host[:port]`
`packs.*.host`	whether a permissive pack's components get `host`, the calls your app offers them
`packs.*.ambient`	every document gets the pack's names without importing it
`packs.*.version`	a range the installed version must satisfy, checked when your app builds
`imports`	`any` (the default): a document may import any pack; `listed`: only those in `packs`
`deny`	packs your app won't serve, all versions or a range, with the reason readers see
`maxPackSize`	the largest pack a document gets, in bytes or as `"10 MB"`; for a pack from a registry, checked before it is downloaded (10 MB when not given)
`libraries`	npm packages you install that the frames offer components by name, besides React and Svelte
`registries`	where packs your app doesn't install come from, by scope (`"@kit"`), with `"*"` for every other scope
`fetch`	who reaches the registries: `host` (the default), through your app's proxy; `direct`, the reader's browser

`deny` is your own policy, for content you won't show or packs you won't serve; it isn't what keeps readers safe. The sandbox does that for every pack, denied or not, known or not.

The plugin checks the configuration when your app builds, and in development when it changes: every problem at once, each listed pack installed at the version it asks for, and each listed pack's documents through the processor.

Packs your app doesn't install

With `registries`, a document may import any pack published on npm, and your app fetches it the first time a document asks. Three files more than the quick start's, with the same `vite.config.ts`. The configuration names npm, reached through your app's proxy, and keeps the default size cap:

```
{
  "registries": { "*": "https://registry.npmjs.org/" },
  "fetch": "host",
  "maxPackSize": "10 MB"
}
```

A document that imports the Mermaid pack, which the app doesn't install, and uses it to draw a code block:

````
---
md:
  import:
    - from: "@comp-markdown/mermaid@0"
      names: [mermaid]
---

# How a change ships

Every change goes the same way, and nobody merges their own:

```mermaid {alt="A change is written, reviewed, merged, then released"}
graph LR
  Written   Written --> Reviewed  Reviewed --> Merged  Merged --> Released
```
````

Try it in the playground, which fetches its packs from npm the same way.

The page, as in the quick start:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>A pack from npm</title>
  </head>
  <body>
    <main id="page" style="height: 24rem"></main>
    <script type="module" src="./main.ts"></script>
  </body>
</html>
```

And `main.ts`, which resolves the document's packs before it opens the document:

```
import { init, OpenDocument } from "@compmd/processor";
import { defaultCache, resolvePacks } from "@compmd/registry";
import { createSandbox } from "@compmd/sandbox";
import { registry, request, sandbox } from "virtual:compmd";
import text from "./plan.comp.md?raw";

await init();

// The document with what the app gives it, then the packs it imports that
the app doesn't install, from the registry.
const asked = { ...request, path: "plan.comp.md", files: { "plan.comp.md":
text }, now: new Date().toISOString() };
const resolved = registry && (await resolvePacks(asked, { ...registry,
cache: defaultCache() }));
for (const { pack, message } of resolved?.problems ?? [])
console.warn(`${pack}: ${message}`);
const opened = new OpenDocument(resolved?.request ?? asked);

// A fetched pack's built module reaches its frame as code, already
checked; an installed one's by its URL.
const page = document.getElementById("page") as HTMLElement;
const modules = (place: Parameters<typeof sandbox.modules>[0]) =>
resolved?.modules(place) ?? sandbox.modules(place);
const view = createSandbox({ ...sandbox, modules }).open(page, {
components: opened.components });
view.render(opened.expanded);
```

Install `@compmd/registry` beside the packages of the quick start, and run it: the page shows the heading, then the diagram, drawn by a pack the app has never seen before.

What just happened

- `virtual:compmd`'s `registry` is what `resolvePacks` takes from your configuration: the registries, who fetches, and the size cap. Without `registries`, it is `undefined`, and the document opens with what your app installs.
- `resolvePacks` asked the processor which packs the document imports that the request doesn't hold (`sources`), resolved each range to the newest matching version, and applied your policy before downloading anything: `deny`, `imports`, and the size cap, from the registry's metadata. Then it fetched the pack's tarball, checked it against the hash npm publishes (and the document's `integrity`, when it pins one), and asked again for the packs that pack imports, until none was missing.
- It gave back the request with the packs added, and `modules`: each fetched pack's built modules, as checked files. The sandbox sends a frame such a module as it sends compiled source, and the strict frame still loads nothing from the network.
- What couldn't be had is in `problems`, and the document says it too: an import no registry could give is `F026`, its uses showing their fallbacks.
- `defaultCache()` keeps what was fetched in the browser's Cache Storage, by integrity: a version never changes, so a later visit fetches nothing but the versions list, once it is older than npm says it may be.

Who fetches, and what the registry learns

With `"fetch": "host"`, the page reaches the registries only through your app's proxy, at `compmd/registry/`, which passes each request on to its scope's registry: npm learns nothing about your readers. The dev server and `vite preview` serve the proxy. A deployed app serves it from its server or a worker, with the same registries; `registryProxy` is a fetch handler, here a worker's:

```
import { NpmRegistry, registryProxy } from "@compmd/registry";

export default { fetch: registryProxy({ registries: { "*": new
NpmRegistry({ url: "https://registry.npmjs.org/" }) }, base:
"/compmd/registry/" }) };
```

It caches nothing itself, so put an HTTP cache or a CDN in front of it; its README has the same for Node. A private registry's token stays on your server, in its `NpmRegistry`'s `headers`, never in the reader's browser.

An app with no server, such as a static site, can't proxy. With `"fetch": "direct"`, the reader's browser fetches from the registries you name, with no cookies or other credentials, and nothing from the document but the names and versions of its imports. The registry then learns the reader's address and which packs, at which versions, the document imports; never the document. Tell your readers.

Neither a document nor a pack can name a registry: imports name packs, and your configuration says where packs come from. A pack's own imports go through the same registries and policy as the document's.

Rendering before the packs arrive

The example waits for its packs before showing anything. An editor can't: it processes on every keystroke. Open the document at once with what you have, with the packs you are getting named in the request's `pending`, resolve them in the background, and open it again when they arrive:

```
const missing = sources(asked).packs.map(({ pack }) => pack);
const first = new OpenDocument({ ...asked, pending: { packs: missing } });
// their uses show as loading, with no F026
view.render(first.expanded);
const resolved = await resolvePacks(asked, { ...registry, timeout: 30_000
});
const opened = new OpenDocument(resolved.request);
// drawn; or F026, for what couldn't be had
```

Until then, each use of a pending pack is its fallback marked as loading (spec §4.5): `@compmd/render` draws it faded, in a box with `aria-busy`, and the diagnostics have nothing to say. The processor never gives up on a pending pack; you do. `resolvePacks` stops waiting for a pack after `timeout` (30 s unless you say), and lists it in `problems`; opened again without it pending, the document says `F026`, as for any pack that can't be had. The playground works this way: a document's packs are pending from its first render, and an import you type is asked for once typing pauses, so a name typed letter by letter isn't fetched letter by letter.

What a pack from a registry can do

The same as an installed one: a pack nobody listed runs strict, whoever published it, wherever it came from. Only your configuration gives a pack more, by name, and it applies to the pack wherever your app gets it. So a document from anyone can import any pack, and still can't reach your page, your session, or your readers' data.

Next: your app's own components.

10. Your app's own components

Your app can give its documents components of its own: a rating, a link to a record, a chart of its data. This chapter builds a page with three. `stars`, in React, and `spoiler`, in Svelte, are trusted: built with the app, they mount in your page. `reviews` is permissive and given `host`: it runs in a frame of its own, and asks your app for what it needs.

A folder is a pack

A pack in your app is a folder with a `package.json` and a document, as a pack on npm is. The `package.json` names it:

```
{
  "name": "@you/app",
  "version": "1.0.0",
  "private": true
}
```

Its document, `components/index.comp.md`, imports each component's file, exports the components under the pack's name and version, and declares their signatures, which every use is checked against before anything runs:

```
---
md:
  import:
    - from: ./Stars.tsx
      names: [{ name: Stars, as: stars }]
    - from: ./Spoiler.svelte
      names: [{ name: Spoiler, as: spoiler }]
  export:
    pack: "@you/app"
    version: "1.0.0"
    names: [stars, spoiler]
---

The app's own components, built with it and trusted: they mount in its
page.
```

````
```compmd component=stars signature
params:
  value: input<number>
  of: number = 5
```

```compmd component=spoiler signature
params:
  label: string
slots: [default]
```
````

A component file exports one component, its default export, named after its file (`Stars.tsx` exports `Stars`); `as` gives it the name documents use. `value: input<number>` makes `value` a param a document binds to one of its inputs, which the component may set. `slots: [default]` gives `spoiler` a body: it is a container, and what a document writes in it is its default slot.

The components

The file's extension says how it is mounted:

Extension	Binding	Built with
`.ts`, `.js`	plain: the default export is the host contract's `mount`	Vite alone
`.tsx`, `.jsx`	React 19: params are props, the default slot is `children`, the rest is the `compmd` prop	Vite alone, with `react` and `react-dom` installed
`.svelte`	Svelte 5: params are props, the default slot is the `children` snippet, the rest is the `compmd` prop	Svelte's plugin, `@sveltejs/vite-plugin-svelte`, with `svelte` installed

A param is a prop of the same name, in camelCase. The `compmd` prop holds the rest of the host contract (spec §7.4): `set`, `fallback`, `scheme`, `fail` and the others.

`Stars.tsx` is a React component:

```
import type { ReactElement } from "react";

interface Props {
  value: number;
```

```
  of: number;
  compmd: { set(param: string, value: unknown): void; fallback: { text:
string } };
}

/** The most stars a document may ask for. */
const MOST = 10;

/** `value` as a whole number from 0 to `max`, whatever a document gave. */
function wholeUpTo(value: unknown, max: number): number {
  const number = Math.round(Number(value));
  return Number.isFinite(number) ? Math.min(Math.max(number, 0), max) : 0;
}

// A rating of `of` stars, `value` of them filled; a click sets the input
`value` is bound to.
export default function Stars({ value, of, compmd }: Props): ReactElement {
  const count = Math.max(1, wholeUpTo(of, MOST));
  const filled = wholeUpTo(value, count);
  return (
    <span role="radiogroup" aria-label={compmd.fallback.text}>
      {Array.from({ length: count }, (_, i) => (
        <button
          key={i}
          type="button"
          role="radio"
          aria-checked={i + 1 === filled}
          aria-label={`${String(i + 1)} of ${String(count)}`}
          onClick={() => {
            compmd.set("value", i + 1);
          }}
        >
          {i < filled ? "★" : "☆"}
        </button>
      ))}
    </span>
  );
}
```

The binding renders it into a React root in its box, and on an update renders it again into the same root, so React keeps its state. `compmd.fallback.text` is the text a reader without it sees, *3 of 5 stars*, which labels the group; `compmd.set("value", …)` changes the input `value` is bound to. It is `@compmd/vite`'s example's `Stars`, with its numbers kept in bounds (below).

`Spoiler.svelte` is a Svelte component, which hides its body until the reader asks:

```
<!-- Its content, hidden until the reader asks: a button shows it, and
hides it again. <!-- Its content, hidden until the reader asks: a button
shows it, and hides it again. -->
<script lang="ts">
  import type { Snippet } from "svelte";

  interface Props {
    label: string;
    children?: Snippet;
  }

  let { label, children }: Props = $props();
  let shown = $state(false);
</script>

<button type="button" aria-expanded={shown} onclick={() => (shown =
!shown)}>
  {shown ? "Hide" : "Show"} {label}
</button>
<div hidden={!shown}>
  {@render children?.()}
</div>
```

`children` is its body: the binding places in it what the renderer made of the Markdown a document writes in `spoiler`. `shown` is the component's own state: a click shows the body, the next hides it again, and `aria-expanded` tells a screen reader which it is. `spoiler` never sets an input, so it doesn't use the `compmd` prop. The website's own `stars` is a Svelte one too.

Vite compiles JSX itself, so React needs no plugin: `"jsx": "react-jsx"` in your `tsconfig.json` gives it React's automatic runtime. A `.svelte` file needs Svelte's compiler, which its plugin brings, so this app's `vite.config.ts` is the quick start's with `svelte()` added:

```
import { compmd } from "@compmd/vite";
import { svelte } from "@sveltejs/vite-plugin-svelte";
import { defineConfig } from "vite";

export default defineConfig({
  plugins: [svelte(), compmd()],
});
```

Params are untrusted input

Your component is trusted, but the documents that use it may be anyone's, and so are its params. The signature checks each param's type, and the values an enum allows; it can't check that a number is in a range, or that two params agree. So a trusted component:

- **never makes HTML from a param**: in React and Svelte, text and attributes in the template, as `Stars` and `Spoiler` do, never `dangerouslySetInnerHTML` or `{@html}`; in a plain component, `textContent` and attributes, never `innerHTML`;
- **keeps numbers in bounds**: here a document that asks for a million stars gets ten, and a value past `of` fills them all;
- **checks URLs** before it shows a link from a param, as the processor's interpolation does: `https:` and the like, never `javascript:`;
- **acts outside the document only when the reader asks**, after showing what it will do ("Send *notes.csv* to …?"): a document that isn't yours can aim it at what it shouldn't touch (spec §15).

The same holds for anything the component reads through `set`'s bound input: it came from the reader or the document.

Trusting it

`compmd.config.json` lists the folder, trusted, and ambient, so every document gets `stars` and `spoiler` without importing them. It lists the second pack of this chapter too, `reviews`, which comes below:

```
{
  "packs": {
    "./components": { "trust": "trusted", "ambient": true },
    "./reviews": { "trust": "permissive", "host": true }
  }
}
```

A folder pack is listed by its path, from your app's root; documents can also import it by its name, `@you/app`. `@compmd/vite` reads it when your app builds: it checks the pack's document through the processor, imports `Stars.tsx` and `Spoiler.svelte` into your app's bundle, each wrapped in its binding (a plain file stays as it is), and gives them to the sandbox in `virtual:compmd`'s `sandbox.trusted`. Their code is never sent to a frame.

The document

`notes.comp.md` uses `stars`, bound to its input `rating`, and `spoiler` around a note on the ending, and imports `reviews` by its pack's name:

```
---
md:
  import:
    - from: "@you/reviews@1"
      names: [reviews]
  inputs:
    rating: number = 3
---

# Reading notes: *The Left Hand of Darkness*

My rating: :value[3]{of=(string(md.input.rating))} of 5.

:: stars[3 of 5 stars] {value=rating}

:: reviews[What other readers thought] {book="The Left Hand of Darkness"}

::: spoiler {label="the ending"}

Estraven is shot at the border, and Genly finishes the mission without him.

::: /spoiler
```

It needs this app's components, so the playground shows their fallbacks: *3 of 5 stars*, *What other readers thought*, and the note on the ending, since a container's fallback is its body. Any other app shows the same fallbacks, as a plain Markdown renderer does.

Permissive components with `host`

`reviews` shows what other readers thought of a book. The reviews are your app's data, so the component asks your app for them, through `host`. It is a pack folder too:

```
{
  "name": "@you/reviews",
  "version": "1.0.0",
  "private": true
}
```

```
---
md:
  import:
    - from: ./reviews.ts
      names: [reviews]
  export:
    pack: "@you/reviews"
    version: "1.0.0"
    names: [reviews]
---

What other readers thought of a book, from the app's data: permissive, and
given `host`.

```compmd component=reviews signature
params:
  book: string
```
```

Listed as `"trust": "permissive", "host": true`, with no `network`, it gets a frame of its own with no network, and `host` in its context. Its source is compiled in the sandbox, as a document's own components are, so it imports nothing but what the frames provide:

```
/** What a permissive component given `host` gets: the app's functions,
called by name, answered by its page. */
interface Host {
  call(name: string, ...args: unknown[]): Promise<unknown>;
}

/** What the renderer gives a plain component (spec §7.4), as far as
`reviews` uses it. */
interface Context {
  params: Record<string, unknown>;
  host?: unknown;
  busy(work: Promise<unknown>): void;
}

/** What the app answers for a book it has reviews of. */
interface Summary {
  count: number;
  average: number;
}
```

```ts
const isSummary = (value: unknown): value is Summary =>
  typeof value === "object" && value !== null && typeof (value as
Summary).count === "number" && typeof (value as Summary).average ===
"number";

// A summary of a book's reviews, which it asks the app for: the reviews
are the app's data, never the document's.
export default function reviews(target: Element, { params, host, busy }:
Context) {
  const line = target.ownerDocument.createElement("p");
  target.append(line);
  /** How many times it asked: an answer to an older question is dropped.
*/
  let asked = 0;
  const show = async (next: Record<string, unknown>) => {
    if (host === undefined) {
      line.textContent = "Reviews aren't available here.";
      return;
    }
    const question = ++asked;
    line.textContent = "Asking for the reviews…";
    let summary: unknown;
    try {
      summary = await (host as Host).call("reviewsOf", String(next.book));
    } catch {
      summary = undefined;
    }
    if (question !== asked) return;
    line.textContent = isSummary(summary) ? `${String(summary.count)}
readers gave it ${summary.average.toFixed(1)} of 5 on average.` : "No
reviews yet.";
  };
  // It draws once the app answers: `busy` tells the page, which waits for
it to settle.
  busy(show(params));
  return {
    update: (next: Record<string, unknown>) => {
      busy(show(next));
    },
  };
}
```

`host.call(name, ...args)` sends the call to your page over the frame's port, and resolves with what your function returned. Arguments and results cross by copy, so they must be

copyable (no functions, no DOM). A name your app doesn't offer, or a function that throws, rejects the call with a message. A component without `host`, in an app that doesn't give it, gets `undefined`, and should say so rather than fail.

`busy(work)` says the component is still drawing until `work` settles, here until the answer is shown (spec §7.4). Nothing shows differently meanwhile, but a page that waits to settle, to print, to scroll to a link's target, or in a test, waits for it: the view's `settled()` (chapter 11).

What `host` holds is your app's: `createSandbox`'s `host` option, a `HostApi`, an object of functions by name.

Your trusted components get the same `host`, in their `compmd` prop (`compmd.host`, or `context.host` for a plain one), answered in your page directly: the same `call`, failing the same way, so a component written for `host` works trusted or permissive. Its type is `HostClient`, from `@compmd/sandbox`.

Your app can also tell them something changed. `view.emit(event, data)` sends an event to every component given `host`, trusted or permissive, and each hears it with `host.on(event, listener)`, which returns the function that stops listening: `view.emit("reviews-changed", { book })` after a reader adds a review, so `reviews` asks again. The data crosses by copy too. Strict components never hear it.

The page

The page as in the quick start, and `main.ts`, which renders the document in host mode with the app's `host`:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>The app's own components</title>
  </head>
  <body>
    <main id="page"></main>
    <script type="module" src="./main.ts"></script>
  </body>
</html>
```

```
import { init, OpenDocument } from "@compmd/processor";
import { createSandbox, type HostApi } from "@compmd/sandbox";
import { request, sandbox } from "virtual:compmd";
import text from "./notes.comp.md?raw";
```

```
await init();

// The app's data, which only the app's page reads: a document can't, and a
component asks for it through `host`.
const summaries = new Map([["The Left Hand of Darkness", { count: 12,
average: 4.2 }]]);

// What trusted components, and permissive ones of packs given `host`, may
call. Arguments come from documents: check them.
const host: HostApi = {
  reviewsOf: (book) => (typeof book === "string" ? (summaries.get(book) ??
null) : null),
};

let inputs: Record<string, unknown> = {};
const open = () =>
  new OpenDocument({ ...request, path: "notes.comp.md", files: {
"notes.comp.md": text }, now: new Date().toISOString(), inputs });
let opened = open();

// Host mode: the page in this page, `stars` and `spoiler` inline,
`reviews` in a permissive frame of its own.
const page = document.getElementById("page") as HTMLElement;
const view = createSandbox({ ...sandbox, page: "host", host }).open(page, {
components: opened.components });
view.render(opened.expanded);

view.on("set", ({ input, value }) => {
  inputs = { ...inputs, [input]: value };
  opened.free();
  opened = open();
  view.render(opened.expanded);
});
```

`page: "host"` matters: in frame mode the whole page is one strict frame, where a trusted component can't run, so `stars` and `spoiler` would show their fallbacks, and the view's `error` event would say `not-here`.

The functions in `host` are where your app's data meets documents that may be anyone's. `reviewsOf` takes its argument from a param, so it checks its type, and looks it up in a `Map`, which has no inherited keys for a document to name. A function that reads only what the reader may see anyway, as this one does, needs nothing more; one that acts (sends, writes, shares) asks the reader first, in your page's own interface.

Run it

In the quick start's app, add these files, set `"jsx": "react-jsx"` in its `tsconfig.json`, and install the components' frameworks:

```
npm install react react-dom svelte @sveltejs/vite-plugin-svelte
npm install -D @types/react
npm run dev
```

The page shows the heading, *My rating: 3 of 5*, three of five stars filled, in a frame below them *12 readers gave it 4.2 of 5 on average.*, and a button, *Show the ending*. Click the fifth star, and the line says *5 of 5*. Click the button, and the note on the ending shows, and the button says *Hide the ending*.

What just happened

- `@compmd/vite` found both folders the configuration lists, read their documents, and gave the processor both packs, with their levels, in `request`. It built `Stars.tsx` and `Spoiler.svelte` into your app, each in its binding, and gave them to the sandbox as `sandbox.trusted.stars` and `sandbox.trusted.spoiler`.
- The processor resolved `stars` and `spoiler` from the ambient pack and `reviews` from the import, checked each use against its signature, and marked each component with its pack's level.
- In host mode, the sandbox rendered the page in `#page` and mounted `stars` and `spoiler` inline, as any of your code. A click on a star called `set("value", 5)`; the view sent `set` for the input `rating`, and your page opened the document again with it. A click on the button changed only `spoiler`'s own state.
- For `reviews`, it compiled the source in its hidden compile frame, made a `permissive.html` frame where the use is, and gave it a port for `host`. The component's `host.call("reviewsOf", …)` came to your page's `reviewsOf` over that port, and its answer went back the same way.
- Only the host's page gives `host`, and only to its trusted components and to frames of packs the configuration gives it. A strict component, or a document, can't get it.

Next: the sandbox, closer, both page modes, the frames and how to serve them.

11. The sandbox, closer

The chapters so far used the sandbox as a box: trees in, a page out. This one opens it. It renders one document in both page modes side by side, then goes through what each mode makes, what the frames load and how to serve them, how a frame that hangs is stopped, and how an editor renders a document block by block.

Both modes, side by side

A document with a component it defines itself, used twice, and an input:

````
---
md:
  inputs:
    docs: number = 72
---

# Release status

Two meters, drawn by a component the document defines itself, so they run
strict wherever they are shown.

```compmd component=meter signature
params:
  value: number
  label: string
```

```jsx component=meter runtime=react@19
export default function Meter({ value, label }) {
  const shown = Math.min(Math.max(Number(value) || 0, 0), 100);
  return (
    <label style={{ display: "grid", gap: 4 }}>
      <span>{label}: <strong>{shown}%</strong></span>
      <progress max="100" value={shown} />
    </label>
  );
}
```
````

```
:: meter[Docs: 72%] {label="Docs" value=(md.input.docs)}

:: meter[Tests: 90%] {label="Tests" value=90}

:: number-input[Docs done, in percent: 72] {bind=docs}
```

Try it in the playground.

A page with a place for each mode:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Frame mode and host mode</title>
  </head>
  <body style="display: grid; grid-template-columns: 1fr 1fr; gap: 2rem">
    <section>
      <h2>Frame mode</h2>
      <div id="frame" style="height: 24rem"></div>
    </section>
    <section>
      <h2>Host mode</h2>
      <div id="host"></div>
    </section>
    <script type="module" src="./main.ts"></script>
  </body>
</html>
```

And `main.ts`, which opens the document once and gives the same tree to two views:

```
import { init, OpenDocument } from "@compmd/processor";
import { createSandbox, type View } from "@compmd/sandbox";
import { request, sandbox } from "virtual:compmd";
import text from "./status.comp.md?raw";

await init();

let inputs: Record<string, unknown> = {};
const open = () =>
  new OpenDocument({ ...request, path: "status.comp.md", files: {
"status.comp.md": text }, now: new Date().toISOString(), inputs });
```

```
let opened = open();

// The same document twice: in one strict frame, and in this page with a
frame per component use.
const views: View[] = (["frame", "host"] as const).map((page) => {
  const container = document.getElementById(page) as HTMLElement;
  const view = createSandbox({ ...sandbox, page }).open(container, {
components: opened.components });
  view.on("error", ({ component, code, message }) => {
    console.warn(`${component}: ${code ?? "no diagnostic"}, ${message}`);
  });
  view.render(opened.expanded);
  return view;
});

// An input set in either view opens the document again, and both show it.
for (const view of views) {
  view.on("set", ({ input, value }) => {
    inputs = { ...inputs, [input]: value };
    opened.free();
    opened = open();
    for (const each of views) each.render(opened.expanded);
  });
}
```

With the quick start's `vite.config.ts`, `npm run dev`. Both sides look the same: the heading, two meters, the number input. Set the input to 40 on either side, and both say *Docs: 40%*. Open your browser's element inspector to see the difference: on the left, one `iframe` holds the whole page; on the right, the heading, the text and the input are in your page, and each meter is an `iframe` of its own.

Frame mode

`page: "frame"`, the default. The view makes one `strict.html` frame in the container, which fills it (give the container a height), and renders the page in it with `@compmd/render`'s frame protocol: trees and splices go down the frame's port, `set`, links and errors come up.

- **Strict components share the frame**, so ten charts cost one frame, and one copy of React.
- **Permissive and trusted components show their fallbacks**, each with an `error` event whose `reason` is `not-here`.
- **The page's links come to your page** as `link` events, since nothing in the frame may navigate.

It is right for documents from anyone with strict packs: a site of others' documents, a preview, a playground.

Host mode

`page: "host"`. The view renders the page in the container itself, with `@compmd/render`, and decides for each use:

- **A trusted component mounts inline**, from `trusted`, as your own code does.
- **A strict or permissive component gets a frame where it is used**, `strict.html` or `permissive.html`, holding that use and its slots. The frame fills the use's box and reports its height, which the view sets; an inline use's frame reports its width too.
- **Built-in components, layout and components written in CompMD** run no code, so they render in your page, and a framed component in their slots gets its frame there.

Frames only go down in trust. Your page holds permissive and strict frames; a permissive frame holds strict frames, for strict components in its slots; a strict frame holds none. So a permissive or trusted component in the slot of a strict container shows its fallback, `not-here`, with a message that names why. Write such a container in CompMD, or, when it is your own, make it trusted.

The cost is a frame, and a copy of its runtime, per use. It is right for an app with components of its own, or permissive packs.

What the frames load

`@compmd/vite` writes the frames' files into your app's `compmd/` folder (`buildFrames` does it for other hosts, chapter 18):

File	What it is
`strict.html`	the strict frame: a fixed content security policy with no network and no nested frames, then the import map, then the frame runtime
`permissive.html`	the same, without the network directives; its parent sends the pack's origins, which the frame adds as a second policy before any component loads, and it may nest `strict.html` only
`runtime.js` and its files	the frame runtime: the sandbox's end of the protocol, the renderer, the bindings

File	What it is
`highlight.js`, `math.js`	Shiki and Temml, which highlight code blocks and typeset math: the frames load them through the runtime, and in host mode your page loads them from here, so your app ships each once
`libs/`	the runtimes every frame shares through the import map, React 19, React DOM and Svelte 5, and the libraries your configuration offers
`library-definition.json`	those runtimes and libraries with their versions, which the processor checks components against
`packs/<name>@<version>/…`	each installed pack's built modules

A browser enforces every policy a document has, so the second policy can only narrow the first: no message can widen a frame's network.

Besides the frames that render, each view makes one hidden `strict.html` frame, the **compile frame**, which compiles the document's source components (JSX, TSX, Svelte, TypeScript) once, and runs none of them. Each frame then gets the compiled code it needs, and only that: a frame never holds code of a level above its own. A pack's **built module** loads in its frame by its URL with its file's hash, `<script type="module" integrity>`, so a file that doesn't match never runs (W121); or, for a pack fetched from a registry, as the checked file itself (chapter 9).

Serving `compmd/`

The frames have an opaque origin: `sandbox="allow-scripts"` without `allow-same-origin`. So what they load from your app is a cross-origin load, and the folder needs two headers:

Path	Header	Why
everything under `compmd/`	`Access-Control-Allow-Origin: *`	the frames' scripts are modules, which load cross-origin only with it
`compmd/strict.html`, `compmd/permissive.html`	`Content-Security-Policy: sandbox allow-scripts`	the frame documents stay sandboxed even when someone opens one on its own, outside a frame

The dev server and `vite preview` send both. A deployed app sends them from its server, or its static host's header rules. Nothing under `compmd/` is secret: it is the same for every

reader.

The frames may be served from another origin than your page, with the same headers: `createSandbox`'s `base` takes a full URL. That is the way to keep the requests a frame can make to them (its scripts, styles and prefetches) apart from your app's own server and logs.

Pings and stopped frames

Every frame that runs components is pinged by its parent every second. One that hasn't answered within `timeout` (5000 ms unless you give `createSandbox` another) is removed, and its use shows its fallback: a component that loops forever stops there. The view's `error` event tells you:

```
view.on("error", ({ component, reason, code, message }) => {
  // A stopped frame: reason "threw", code "W118", message "its frame
stopped answering".
});
```

`reason`	`code`	When
`threw`	`W118`	the component threw, called `fail`, or its frame stopped answering
`failed`	`W121`	its code didn't compile or load: an import it may not make, a module that doesn't match its hash
`not-here`	none	its level can't run where it is used
`missing`	none	no code for it here

`diagnosticOf(reason)`, from `@compmd/sandbox`, gives the code for a reason.

In frame mode the pinged frame is the page's: when it stops answering, the error's `component` is empty, and its message says *the page's frame stopped answering*. In WebKit, a frame that spins freezes your page with it, since WebKit runs these frames in the page's process; Chrome keeps the page going and stops the frame (chapter 8).

When the page has settled

A page keeps moving after `render()` returns: each frame takes its height once it has drawn and reported it, its own frames later still, and a component that draws late, as `reviews` does once your app answers (chapter 10), later again. To print, to scroll to a link's target, or to click something in a test, wait for it to stop:

```
view.render(opened.expanded);
await view.settled();
```

`settled()` resolves once, after the latest render or splice, every use in your page and in every frame has drawn (a component that draws late says so with `busy`, spec §7.4), every frame has its last height, the page's images and fonts have loaded, and nothing moved for an animation frame; at once when that holds already. The view's `settled` event comes each time it settles again, and the container has the attribute `data-compmd-settled` while it is settled, so a browser test waits for it without reaching into your app: `await page.locator("#page[data-compmd-settled]").waitFor()`. A frame that hangs is stopped and counts as settled; a drawing or a frame that never says it is done holds it up for 10 s at most.

Editors that render block by block

An editor that shows a document's blocks in its own widgets, as CodeMirror's are, renders block by block rather than as a page. It uses the layer under `createSandbox`, the **host runtime**, one per document, and mounts each use in its widget:

```
import { HostRuntime } from "@compmd/sandbox";
import { sandbox } from "virtual:compmd";

const runtime = new HostRuntime(sandbox, {
  set: (input, value) => setInput(input, value),
  link: (url) => confirmAndOpen(url),
  error: (component, reason, message) => report(component, reason,
message),
});
await runtime.set(opened.components);

// In a widget, for one use of the expanded tree, with the document's
inputs.
const use = runtime.mountUse(widget, node, opened.expanded.inputs ?? {});
use.update(nextNode, nextInputs);
use.destroy();
```

`mountUse` does what host mode does for one use: a trusted component inline, any other in a frame of its level. `runtime.mounts` is the same as a `components` map for `@compmd/render`'s regions, for the blocks around the uses.

An editor removes widgets that scroll out of view and builds them again, and a frame whose element is removed or moved loads again: its component boots again, and its state is lost.

So a component's state belongs in its params and inputs, and an editor that minds keeps a widget's DOM while it is near the viewport.

A host page that is itself sandboxed

The window where the sandbox runs, the **host's page**, needn't be your top-level page. An app may run its editor in a sandboxed frame of its own; that frame is then the host's page, with an opaque origin, its trusted components are the app's code in it, and its `host` is what the app gives that frame. It works because a frame believes its parent and nothing else: it takes its port, its level, its code and its part only from the window that made it, by that window, never by an origin. A message from any other window, such as a strict frame posing as a parent through `top.frames`, is ignored.

The protocol, in brief

The windows talk over `MessageChannel` ports, one per frame, to its parent:

Window	Made by	Runs
the host's page	your app	the host runtime; in host mode the page and trusted components
the page frame (frame mode)	the host's page	the page and its strict components
a use frame, strict or permissive	the host's page; a permissive frame, for a strict use in its slots	one use and its slots
the compile frame	the host's page	the compilers, no components

A parent sends `hello` with the port, then `level` (and a permissive frame's second policy), `code`, and the part to render; it pings, and sends `host` to permissive frames of packs given it. A frame sends `ready`, `loaded`, `set`, `link`, `error`, `rendered`, its `size`, `settled` once it has, and `pong`. A parent checks each `set` against the use it rendered, so a frame can only set the inputs its own use binds, and passes what concerns the windows above it up. The protocol's design has every message and sequence; `@compmd/sandbox/protocol` is the protocol alone, with no DOM, and `protocol.schema.json` describes its messages, for an implementation elsewhere (the sandbox's README).

Next: the Developer Tools, which show all of this as it happens.

12. The Developer Tools

`@compmd/devtools` shows what a document becomes and what its sandbox does: its diagnostics, its components, its frames, its data, the messages between its windows. This chapter puts its inspector under an app's page in development, embeds its editor, and goes through what each tab tells you and how far to believe it.

Two ways to embed it

	The editor	The inspector
What it shows	a document's text and its page, side by side, and the tabs	the tabs, for a page your app already renders
Its page	its own, in a sandbox it opens	yours: it observes your view
You give it	`sandbox` (as `createSandbox` takes it) and `document`	`view` and `document`
For	a tool to write documents in, a preview, a playground	seeing what your app's documents do, in development

Both are one function, `mountDevtools(target, options)`, whose options say which. The tools are written in Svelte and compiled into the package, so a page in any framework embeds them, with no Svelte of its own. They render in a shadow root of `target`, so your page's styles can't reach in, and theirs can't leak out. They fill the element you give them.

```
npm install @compmd/devtools
```

The inspector, in development

This chapter's example is the quick start's app with the tools added: the same document, which counts the tasks done,

```
---
tasks: [Write the outline, Draft the first part, Ask for a review, Publish]
md:
  inputs:
```

```
    done: number = 1
---

# The guide, step by step

::: box {padding=m border=subtle border-radius=m}
**:value[1 of 4 done]{of=(string(md.input.done) + " of " +
string(size(md.doc.tasks)) + " done")}**

:: number-input[Done: 1] {bind=done}
::: /box

::: each {in=(md.doc.tasks) as=task join=list}
1. {{task}}
::: /each
```

and a page with a panel under the document's for the inspector:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Notes, inspected</title>
  </head>
  <body>
    <main id="page" style="height: 24rem"></main>
    <section id="inspector" aria-label="The inspector"></section>
    <script type="module" src="./main.ts"></script>
  </body>
</html>
```

Its `main.ts` opens the inspector on the view it renders, in a function it calls only in the dev server's mode, so a build leaves the tools out:

```
import { init, OpenDocument, type Request } from "@compmd/processor";
import { createSandbox } from "@compmd/sandbox";
import { request, sandbox } from "virtual:compmd";
import text from "./notes.comp.md?raw";

await init();

let inputs: Record<string, unknown> = {};
const requestOf = (): Request => ({ ...request, path: "notes.comp.md",
```

```
files: { "notes.comp.md": text }, now: new Date().toISOString(), inputs });
let opened = new OpenDocument(requestOf());
const view = createSandbox(sandbox).open(document.getElementById("page") as
HTMLElement, { components: opened.components });

/** In development, the inspector on the page's view, in a panel under it:
loaded only then, so a build leaves it out. */
async function inspect() {
  const { mountDevtools } = await import("@compmd/devtools");
  const panel = document.getElementById("inspector") as HTMLElement;
  panel.style.height = "60vh";
  return mountDevtools(panel, { view, document: { request: requestOf() }
});
}
// The dev server's mode: `DEV` follows `NODE_ENV`, which a build run by a
test runner may set to `test`.
const inspector = import.meta.env.MODE === "development" ? await inspect()
: undefined;

view.on("set", ({ input, value }) => {
  inputs = { ...inputs, [input]: value };
  opened.free();
  const processed = requestOf();
  opened = new OpenDocument(processed);
  view.render(opened.expanded);
  inspector?.open({ request: processed });
});
view.render(opened.expanded);
```

`view` is what `createSandbox(…).open(…)` gave, and `document` the request the app processed, which the inspector processes again in a worker of its own for its tabs. When the app processes the document again, as a reader sets an input, it tells the inspector, with `open`.

The inspector observes the view from when it mounts, however long the view has been open: the page and each frame report what is live first. `destroy()` removes it and stops observing.

The tools process documents in workers, which are files of their own beside the package's code, with the processor's WebAssembly. Vite builds a package's workers only when it doesn't pre-bundle the package, so list it in `optimizeDeps.exclude`:

```
import { join } from "node:path";
import { compmd } from "@compmd/vite";
import { defineConfig } from "vite";
```

```
export default defineConfig({
  plugins: [compmd()],
  // Two pages: the app's, with the inspector in development, and the
editor's.
  build: { target: "es2024", rolldownOptions: { input:
[join(import.meta.dirname, "index.html"), join(import.meta.dirname,
"editor.html")] } },
  // The tools' workers are files of their own, which Vite builds only in a
package it doesn't pre-bundle.
  optimizeDeps: { exclude: ["@compmd/devtools"] },
});
```

The dev server, `npx vite`, serves the page with the inspector under it; `npx vite build` builds it without.

Embedding the editor

The editor is the playground's: the text with the hint at the caret and the diagnostics under it, the page beside it, updated as you type. A click on the page puts the caret where what was clicked came from, and moving the caret outlines on the page what it is in (chapter 5). The example's second page is one, which fills the window:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Notes, edited</title>
  </head>
  <body style="margin: 0">
    <main id="tools" style="height: 100vh"></main>
    <script type="module" src="./editor.ts"></script>
  </body>
</html>
```

Give it where your frames are and a document:

```
import { mountDevtools } from "@compmd/devtools";
import { request, sandbox } from "virtual:compmd";
import text from "./notes.comp.md?raw";

mountDevtools(document.getElementById("tools") as HTMLElement, {
  sandbox,
```

```
  document: { request: { ...request, path: "notes.comp.md", files: {
"notes.comp.md": text }, now: new Date().toISOString() } },
});
```

Option	What it is
`sandbox`	the sandbox the page renders in, as `createSandbox` takes it: `virtual:compmd`'s `sandbox`, with `page: "host"` for host mode; frame mode by default
`document`	the document to open: `request`, a processor's request with its text in `files`, and `files`, what it reads, shown read-only beside its text (`{ name, text }`)
`registries`	where to fetch the packs a document imports that its request lacks, as `virtual:compmd`'s `registry` gives them (chapter 9); without it, those are F026
`stylesheets`	your stylesheets for the page, served from the frames' origin
`select`	a passage of the text to select when the document opens, a text fragment's `text=` value
`shareLink`	given the same value for the selected text, the link that opens it: the tools then offer to copy a link to the selection

`mountDevtools` returns a handle, which has `open(document)`, which opens another document in a new page; `styles(urls)`; `state`, with the diagnostics, problems and timings; and `destroy()`. Each key press is an edit, processed in a worker that keeps the document open, so only the blocks it touches are processed again, and the page follows as splices.

Reading the tabs

The editor has two tabs: **Page**, the document rendered, kept mounted under the other; and **CompMD Inspector**, which holds the views as tabs of its own, with the **Console** docked under them, as a browser's developer tools dock theirs. The inspector has only the Inspector, since the page is yours. In order, then the Console:

Tab	What it shows
Plain Markdown	what a plain CommonMark and GFM renderer shows, in a frame without scripts

Tab	What it shows
Parse tree, **Expanded tree**	the processor's trees as JSON that folds
Data	what the frontmatter gives the render: its fields, each data import with what was loaded, the inputs, the settings
Components	those the document defines and those it uses, and those it could use
Sandbox	the windows as a tree, from the host's page down to the frames its frames make: each with its level, its network, what it renders and its last render's time, the components mounted in it with how long they took to mount and update, and its fallbacks
Messages	the sandbox's protocol on a timeline, a lane per frame, each message opening as the trees do
Console, under the tabs	whichever tab shows; its bar hides or shows it (Esc does too), and its top edge drags. It lists what went wrong, newest last: each change's diagnostics, packs that couldn't be fetched, and from the sandbox each component that shows its fallback with why (W118, W121, `not-here`), each `set` it refused, each frame it stopped; and under them, an input that evaluates an expression in the document (below)

In the editor, a bar over the tabs lists the document's inputs, each with a control by its type; changing one there opens the document again with it. In the inspector, the inputs are your page's, and its Data tab shows them.

A component's time is its `mount`'s and its updates' calls. A React component's mount schedules its render, so most of its time is in the frame's own render time.

Evaluating in the Console

The input under the Console's list evaluates an expression in the document, as a browser's console evaluates JavaScript: write what you would write between a param's parentheses, and press Enter. For a document with a `title`, an input `region` set to `eu`, and three sales of 120 (`eu`), 80 (`us`) and 30 (`eu`):

```
› md.data.sales.filter(s, s.region == md.input.region).map(s, s.amount)
[120, 30]
› md.sum(md.data.sales.map(s, s.amount))
230
```

```
› md.doc.titel
         ^
F024 unknown field 'titel'; did you mean 'title'?
```

- **What it sees** is what the document's body sees outside any component: `md.doc`, `md.data` with what was loaded, `md.input` with the values the page shows, `md.version`, `md.lang`, `md.tz`, and the `md.` functions at the render's time, `md.today()` among them. Not what an `each` names (`as`) or a `compmd` definition's params, which only a place in the text has.
- **The keys:** Enter evaluates, Shift+Enter adds a line, and ↑ and ↓ go through what you evaluated before.
- **A value** shows as a tree, as the Data tab shows data. **An error** is F024, as in a document: the expression with `^` under where it is, and the message, with the nearest name the document has when you wrote one it doesn't. It is checked against the document's types first, as an expression in the document is, and runs within the same work budget, so what works here works in a param.
- **It changes nothing:** the document, the page and the diagnostics stay as they are. The inspector has the same input, on the request your app gives it.

From code, an open document has the same: `opened.evaluate("md.data.sales.size()")` gives `{value}` or `{error}` (`@compmd/processor`), and in Rust, the expanded processor's `evaluate`.

The observer behind them

The Console, Sandbox and Messages tabs read the sandbox's observer, which any developer tool can use. Only your page turns it on, with the view:

```
const observation = view.observe((events) => {
  show(observation.events);
});
observation.dropped;   // what was dropped, by source
observation.stop();    // the sandbox observes while an observation listens
```

`events` holds what the host's page saw and what its frames reported, in the order it came: `frame` and `gone` as frames come and go (with why: `removed`, `hung`, `version`), `message` for each message on a port (`down` or `up`, pings left out), `mounted`, `updated` and `unmounted` with their times, `fallback` with its reason, `refused` for a `set` the parent dropped, and `dropped`. Each has its time, `at`. Frames are ids that are paths: `"2/1"` is the first frame of the second.

Nothing a document, a pack or a frame sends turns it on. Off, nothing is copied. On, it keeps the last 2,000 events of each source, your page and each frame, so a frame that floods it pushes out only its own; long values are kept by their size (`{ $text: length, start }`). It changes nothing the sandbox does, and sends nothing anywhere.

What "reported by" means

Your page sees its own frames and the messages on their ports at first hand. What happens inside a frame (the frames a permissive frame makes, the components mounted in a page frame, their times) only that frame can see, so it reports it, once its parent asks. Such an event has `by`, the frame whose report it is, and the tabs mark it **reported by** that frame.

Code of that frame's level shares the frame's window with the code that reports, and could change what it says: a component could report that it rendered in 2 ms with no network when it did something else. The tools can't tell, so they mark it, show everything a frame sends as text, never as markup or a link, and nothing acts on it: which frame gets `host`, which `set` passes and which frame is stopped come from the sandbox's own checks, never from a report. A frame also learns it is observed when its parent asks, so code of its level can behave differently while the tools are open. The tools show what a frame does, not what it would do unobserved (the design's accepted risk).

A lie gains little: a frame is never less trusted than the frames it reports on, so code that can falsify a report controls what it reports on anyway.

Next: your first pack, a component of your own for any app's documents.

13. Your first pack

A pack is an npm package of components that any document can import, from any host. This chapter writes one with `npm create @compmd/pack`, opens its test page, and goes through the component it holds: one function in the plain binding, and the context the host gives it.

Writing one

```
npm create @compmd/pack hello -- --name @you/hello
cd hello
npm install
npm run dev
```

The command asks nothing: everything is a flag, so it runs the same in a script. The flags go after `--`, which passes them on from `npm create`.

Flag	What it does
`--name <@scope/pack>`	the npm package, which must be scoped; by default `@<your npm user>/<folder>`, or `@me/<folder>` when `npm whoami` doesn't answer, with a note to change it
`--component <name>`	the component's name, which documents use; by default the folder's name in lowercase, words joined by `-`
`--renders <language>`	makes the component a code renderer of that language's code blocks (chapter 16)
`--library <package>`	makes a library pack of an npm package instead, with no component (chapter 15)

It refuses a folder that has files in it, a name that isn't scoped, and `--library` together with `--component` or `--renders`. When it is done it says what it wrote, here `Wrote @you/hello in hello: 8 files, with the component hello.`, and the three commands to run next.

What it writes

Only what is the pack's own:

```
hello/
  package.json         the package, its scripts, and what it publishes
  index.comp.md        the pack's document: the export, the import of the
built module, the signature
  src/hello.ts         the component
  src/hello.test.ts    its tests
  demo.comp.md         the test page's document
  tsconfig.json        one line: @compmd/pack's
  README.md            what to change, how to try it, how to publish
  .gitignore           node_modules and dist
```

There is no build configuration, no test page and no publish check in it. Those are `@compmd/pack`'s, and the scripts run its command, `compmd-pack`, so upgrading `@compmd/pack` updates them in every pack made earlier:

```
{
  "name": "@you/hello",
  "version": "0.1.0",
  "description": "A CompMD pack: hello.",
  "keywords": ["compmd", "compmd-pack"],
  "license": "MIT",
  "type": "module",
  "compmd": "./index.comp.md",
  "files": ["dist", "index.comp.md", "README.md"],
  "scripts": {
    "build": "compmd-pack build",
    "dev": "compmd-pack dev",
    "test": "compmd-pack test",
    "lint": "tsc --noEmit",
    "prepublishOnly": "compmd-pack build && compmd-pack test"
  },
  "devDependencies": {
    "@compmd/pack": "0.3.0",
    "happy-dom": "21.0.3",
    "typescript": "6.0.3",
    "vitest": "5.0.5"
  }
}
```

The versions in `devDependencies` are the ones `@compmd/create-pack` was released with, pinned, so a pack made today builds the same later. Chapter 17 goes through the fields that make it a pack on a registry.

The pack's document is what a host reads. It imports the component's built module, `./dist/hello.js`, exports it under the pack's name and version, and declares its signature, which every use is checked against before anything runs:

````
---
md:
  import:
    - from: ./dist/hello.js
      names: [hello]
  export:
    pack: "@you/hello"
    version: "0.1.0"
    names: [hello]
---

# @you/hello

`hello`, a leaf: `:: hello[What a reader without it sees] {value="Hello"}`.

```compmd component=hello signature
params:
  value: string
```
````

The import names the built file, not the source: what a host loads is what you published. `compmd-pack build` builds `dist/hello.js` from `src/hello.ts` (or `.tsx`, `.js`, `.jsx`), one module for each `./dist/<name>.js` the document imports. Chapter 14 is about the signature.

The TypeScript configuration is `@compmd/pack`'s, strict, with JSX for React:

```
{ "extends": "@compmd/pack/tsconfig.json", "include": ["src"] }
```

The test page

`npm run dev` runs `compmd-pack dev`, which builds the pack's modules and serves a page that renders `demo.comp.md`. The template writes one that imports the pack and uses its component:

```
---
md:
  import:
```

```
    - from: "@you/hello@0"
      names: [hello]
---

# hello

:: hello[Hello, from hello] {value="Hello"}
```

The page renders it as a host renders it: the pack as a folder pack, in host mode, each use in a strict frame of its own that takes the height of what it draws. Under the page are the document's diagnostics and the components' failures, and above it a switch between the reader's colour scheme, light and dark. Saving a file in `src/` (but a test) builds the modules again, and the page reloads. `--port <n>` picks the port: `npm run dev -- --port 4300`.

Nothing of the page is written into the pack. Add uses to `demo.comp.md` as you add params, so you see each one, light and dark. The demo isn't published: `files` doesn't list it.

The plain binding

A built module is the plain binding (spec §7.6): its default export *is* `mount`, a function that gets an element and a context, draws into the element, and returns what to do on an update and when the component goes. The template's component:

```
// `hello`: draws its `value` in its box. Replace the drawing with yours;
what it imports is bundled in.
import { component } from "@compmd/pack/component";

/** The params, as the signature in `index.comp.md` declares them. */
export interface Params {
  value: string;
}

export default component<Params>((target, { params, fallback }) => {
  const shown = document.createElement("p");
  // The fallback is what a reader without the component sees: its text is
what you draw, as text.
  shown.setAttribute("aria-label", fallback.text);
  shown.style.cssText = "display: inline-block; margin: 0; padding: 0.4em
0.9em; border: 2px solid currentColor; border-radius: 999px; font-weight:
600;";
  target.append(shown);
  const draw = ({ value }: Params) => {
    shown.textContent = value;
```

```
  };
  draw(params);
  return {
    update: draw,
    destroy: () => {
      shown.remove();
    },
  };
});
```

`component<Params>` from `@compmd/pack/component` only types it: it returns your function as it is, a `Mount`, with `params` and `update`'s params typed as `Params` instead of `Record<string, unknown>`. The interface repeats the signature; the signature is what counts, and the processor has checked every value against it before the component sees it. Nothing generates one from the other yet, so keep them the same.

`target` is the component's box, which the host made: it has already applied the common params a document gives (`width`, `padding`, `border`, …), `class`, `id`, `lang` and `dir`, and the component doesn't receive them. What is inside the box is the component's.

The context

The second argument is the host contract's context (spec §7.4):

Field	What it is
`params`	the params as the signature declares them, checked and typed, with the defaults of those the document didn't give
`slots`	each slot's content by name, `default` too: a list of items, each `{ element, component? }`, which the host rendered and the component places (chapter 14)
`set(param, value)`	changes the input bound to an `input<…>` param (chapter 14); it throws for any other param, and during `update`
`fallback`	the use's fallback, `{ text, element }`: as plain text, the component's text alternative, and rendered
`headingLevel`	the level the component's top heading should have, so the page's outline stays right wherever it is used
`scheme`	`"light"` or `"dark"`: the colour scheme the page shows the component in

Field	What it is
`fail(message)`	the component can't draw what it was given, found after `mount` or `update` returned
`host`	what the host gives the components of packs it makes permissive; absent otherwise, and always in the strict sandbox

`fallback.text` is what a plain renderer shows in the component's place: `Hello, from hello` in the demo. A component that draws anything but text (a chart, a map, an icon) uses it as its text alternative, as `hello` does with `aria-label`; the host doesn't label the box, since only the component knows its role.

`scheme` is the page's, not the reader's preference, which may differ: a light page for a reader who prefers dark is light. A component styled with CSS colours (`currentColor`, as `hello` is) needs nothing. One that takes its colours when it draws reads `scheme`, and when the scheme changes it gets an update with the same params.

`fail(message)` is for a component that finds out late that it can't draw what it was given: an asynchronous parse, say. The host reports W118 with the message, and shows the fallback in the box until the next update, when it empties the box and gives it back. It is for input the component can't draw, not for a resource that may come back: a map whose tiles stop loading shows its own state and retries. Chapter 15 has a helper that calls it for you.

Update and destroy

`mount` runs once per use. It returns an object with two optional functions, or nothing at all when it needs neither:

- **`update(params, slots)`** is called when a param or a slot changes (a reader changed an input, the data changed), and when the scheme changes, with the same params. The component is the same instance, and keeps its state: update what you drew, don't draw it again from nothing. Read the new values from the arguments; `context.params` and `context.slots` are those of the mount, while `context.scheme` is always the current one.
- **`destroy()`** is called before the box is removed: stop timers, remove listeners, unmount a framework.

A throw in `mount` shows the fallback for good, with W118: the component never mounted, so there is nothing to update. A throw in `update` is the same as `fail`: the fallback shows until the next update.

Its tests

The template writes a test that runs the component as a host does, with `mountForTest` from `@compmd/pack/testing`:

```
import { mountForTest } from "@compmd/pack/testing";
import { describe, expect, it } from "vitest";
import mount from "./hello.ts";

describe("hello", () => {
  it("draws its value, follows new params, and goes when destroyed", () =>
{
    const shown = mountForTest(mount, { params: { value: "Hello" },
fallback: "Hello, as text" });
    expect(shown.target.textContent).toBe("Hello");
    expect(shown.target.querySelector("[aria-label]")?.getAttribute("aria-
label")).toBe("Hello, as text");
    shown.update({ value: "Again" });
    expect(shown.target.textContent).toBe("Again");
    shown.destroy();
    expect(shown.target.textContent).toBe("");
  });
});
```

`npm test` runs it in happy-dom; `npm run build` builds `dist/hello.js` and checks the pack as a host would. Both, and the rest of what `mountForTest` gives, are in chapter 17.

Next: signatures and params, what a component can declare and how a document's use of it is checked.

14. Signatures and params

A component's signature says what a document may give it: its params, their types and defaults, and its slots. Hosts check every use against it before any code runs, so a mistake shows as a diagnostic on the line that made it, and the component gets only values of the types it declared. This chapter builds a pack of two components, a rating the reader sets and lanes of records, and goes through every type, slots, template slots, inputs and how a use is checked.

The signature fence

An exported component written in code must have a signature (F029 otherwise), in a fence of the language `compmd` with `component=<name>` and the flag `signature`, in the pack's document. It holds YAML: `params`, a mapping of each param's name to its declaration, and `slots`, the slots it takes. A pack's document may declare several components, each imported from its own built module:

````
---
md:
  import:
    - from: ./dist/rating.js
      names: [rating]
    - from: ./dist/lanes.js
      names: [lanes]
  export:
    pack: "@you/rating"
    version: "0.1.0"
    names: [rating, lanes]
---

# @you/rating

`rating`, a leaf the reader sets, bound to a number input: `:: rating[3 of
5 stars] {value=stars}`.

```compmd component=rating signature
params:
  value: input<number>
  max: number = 5
````
````

````
  size: small | large = small
  color: color?
```

`lanes`, a container whose default slot is a template, once per record, in the lane of its `by` field.

```compmd component=lanes signature
params:
  rows: array
  by: string
  as: name
slots:
  default: {each: rows, as: as}
```
````

A signature fence takes no param list (F036); a component's box is styled by the document that uses it. `renders`, for code renderers, is chapter 16's.

The build makes one module for each `./dist/<name>.js` the document imports, `dist/rating.js` from `src/rating.ts` and `dist/lanes.js` from `src/lanes.ts`. The rest of the pack is the template's, with its own name:

```
{
  "name": "@you/rating",
  "version": "0.1.0",
  "description": "A CompMD pack: a rating the reader sets, and lanes of
records grouped by a field.",
  "keywords": ["compmd", "compmd-pack"],
  "license": "MIT",
  "type": "module",
  "compmd": "./index.comp.md",
  "files": ["dist", "index.comp.md", "README.md"],
  "scripts": {
    "build": "compmd-pack build",
    "dev": "compmd-pack dev",
    "test": "compmd-pack test",
    "lint": "tsc --noEmit",
    "prepublishOnly": "compmd-pack build && compmd-pack test"
  },
  "devDependencies": {
    "@compmd/pack": "0.3.0",
    "happy-dom": "21.0.3",
    "typescript": "6.0.3",
```
```

```
    "vitest": "5.0.5"
  }
}
```

```
{ "extends": "@compmd/pack/tsconfig.json", "include": ["src"] }
```

Declarations

A declaration's short form is a YAML string, its type, then `?` when it is optional, then `=` and a default:

```
max: number = 5
size: small | large = small
color: color?
```

The long form is a mapping, for a default the short form can't write (one with `:` in it, such as a JSON object, isn't a valid YAML string) or a description for editors:

```
meta:
  type: object
  default: {source: survey, year: 2026}
  doc: Where the numbers come from
```

A type that isn't one of those below, such as a misspelt `numbr`, is F037 at the type's name. The param is kept, but takes any value unchecked, so the one mistake gives one diagnostic, not one at each use.

The types

Type	A document writes	The component gets
`string`	`title="Setup"`, or bare, `title=Setup`	the text
`number`	bare: `max=5`, `max=-0.5`	a number
`boolean`	`open=true`, `open=false`, or the flag `open`	`true` or `false`
`date`	`due=2026-10-21`	the text `2026-10-21`

Type	A document writes	The component gets
`array`	JSON on one line: `values=[3, 5, 8]`	an array
`object`	JSON on one line: `meta={"year": 2026}`	an object
an enum, `small | large`	one of its members, bare or quoted: `size=large`	the member, as text
`length`	`px`, `rem`, `em` or `%`, or `0`: `width=12rem`	the text
`space`	`none`, `s`, `m`, `l`, or a length	the text
`color`	a CSS color: `#b8860b`, `teal`, `oklch(0.7 0.1 200)`	the text
`tracks`	grid tracks: `"1fr 2fr"`, `"repeat(auto-fill, minmax(12rem, 1fr))"`	the text
`areas`	quoted rows of area names, all as long: `"'head head' 'side main'"`	the text
`surface`	`none`, `subtle`, `strong`, `accent`, or a CSS color	the text
`border`	`none`, `subtle`, `strong`, or a CSS border: `"1px solid #d0d7de"`	the text
`radius`	`none`, `s`, `m`, `l`, `full`, or a length	the text
`name`	a bare identifier, never read as an expression: `as=t`	the text
`input<type>`	the name of an input the document declares: `value=stars`	the input's value

- **Any type but `name` and `input<…>` takes an expression**, `rows=(md.doc.tasks)`, checked by its type when the document is validated (F024 otherwise), and evaluated before the component sees it.
- **A number, a boolean and a name are written bare.** `max="5"` is F009: a quoted value is a string. A string may be written bare, and a quoted one may interpolate: `title="Step {{n}}"`.
- **Enum members are names**, written without quotes in the signature; a member with spaces needs the whole declaration quoted (`status: '"to do" | done = done'`). An enum has at least two members: a single name is a type name, so `size: small` is F037.

- **The CSS types are closed.** A value outside its grammar is F009, so no param passes raw CSS through. The component gets the text as written, and a token (`m`, `subtle`) is its to read: the host gives a component's own params no meaning.
- There are no other types: a list is an `array` and a record an `object`, whose contents the signature doesn't check. Slots are declared under `slots`, not as params.

Defaults and required params

- A param with a default is optional: the component gets the default when the document doesn't give one. `max` is `5` in every use that doesn't say.
- A param whose type ends in `?` is optional with no value: when the document doesn't give it, it isn't in `params` at all, so its TypeScript type is optional (`color?: string`).
- Any other param is required: a use without it is F010, and renders its fallback, so the component never sees a use that lacks it.

The component that reads them:

```
import { component } from "@compmd/pack/component";

/** The params, as the signature in `index.comp.md` declares them: `color`
is optional, so it may be missing. */
export interface Params {
  value: number;
  max: number;
  size: "small" | "large";
  color?: string;
}

export default component<Params>((target, { params, fallback, set }) => {
  const stars = document.createElement("span");
  stars.setAttribute("role", "group");
  stars.setAttribute("aria-label", fallback.text);
  target.append(stars);
  const draw = ({ value, max, size, color }: Params) => {
    stars.style.fontSize = size === "large" ? "1.6em" : "1em";
    stars.style.color = color ?? "inherit";
    stars.replaceChildren(
      ...Array.from({ length: max }, (_, i) => {
        const star = document.createElement("button");
        star.type = "button";
        star.textContent = i < value ? "★" : "☆";
        star.setAttribute("aria-label", `${String(i + 1)} of
```

```
${String(max)}`);
        // The reader's choice goes to the input `value` is bound to; the
host renders again with it.
        star.addEventListener("click", () => {
          set("value", i + 1);
        });
        return star;
      }),
    );
  };
  draw(params);
  return {
    update: draw,
    destroy: () => {
      stars.remove();
    },
  };
});
```

The signature checks types and an enum's members, never what values mean together, or a number's range. `max=-3` is a number, so the component is the one to decide what a negative count draws: here, no stars. Params come from documents anyone writes, so treat them as untrusted input. Never put one in HTML as markup: `rating` writes text with `textContent`.

Slots

`slots: [default, footer]` makes the component a container that takes a body, its content before the first `:: #footer` line the default slot, and what follows the footer. A component with no `slots` key takes no body: a use with one is F018, and the body follows the component as ordinary content. A slot the signature doesn't declare is F006, and its content joins the default slot.

The component gets each slot in `context.slots`, by name, and again as `update`'s second argument: a list of items, one per top-level block of the slot, each `{ element, component? }`. The host rendered each block into its `element`, a box of its own, and keeps it live, including the components inside it, whatever framework the parent uses. The component only places the elements in its own DOM: never copy their HTML. When the block is a component, `component` holds its name and params, so a tabs component can read its tabs' labels. Every slot is rendered up front, even what the component hides.

Template slots

A template slot is rendered once per element of an array param, with a name the author picks for the element: a board that puts each card in its lane, a calendar that puts each event on its day. The slot's options name the array param (`each`) and a `name` param (`as`) of the same component:

```
slots:
  default: {each: rows, as: as}
```

The component gets the slot as a list of instances instead of items, one per element: `{ value, items }`, the element and the blocks its template rendered.

```
import { component, type Slots } from "@compmd/pack/component";

/** The params, as the signature in `index.comp.md` declares them. `as` is
the author's name for a record. */
export interface Params {
  rows: Record<string, unknown>[];
  by: string;
  as: string;
}

/** One instance of the template slot: its record, and the blocks the
template rendered for it. */
interface Instance {
  value: unknown;
  items: { element: Element }[];
}

export default component<Params>((target, { params, slots }) => {
  const board = document.createElement("div");
  board.style.cssText = "display: flex; gap: 1rem; align-items: start;";
  target.append(board);
  const draw = ({ by }: Params, { default: instances = [] }: Slots) => {
    const lanes = new Map<string, HTMLElement>();
    for (const { value, items } of instances as Instance[]) {
      const field = (value as Record<string, unknown> | null)?.[by];
      const name = typeof field === "string" ? field : "";
      let lane = lanes.get(name);
      if (!lane) {
        lane = document.createElement("section");
        lane.append(Object.assign(document.createElement("h3"), {
```

```
textContent: name }));
        lanes.set(name, lane);
      }
      // The renderer made these elements, and keeps what is in them live:
the component only places them.
      lane.append(...items.map(({ element }) => element));
    }
    board.replaceChildren(...lanes.values());
  };
  draw(params, slots);
  return {
    update: draw,
    destroy: () => {
      board.remove();
    },
  };
});
```

`rows` is typed `Record<string, unknown>[]` for reading, but an `array` may hold anything a document gives, so `lanes` checks each field before it uses it. Its test builds the instances a renderer would give:

```
import { mountForTest } from "@compmd/pack/testing";
import { describe, expect, it } from "vitest";
import mount from "./lanes.ts";

/** A template slot's instance, as the renderer gives it: a record, and an
element for what its template rendered. */
const instance = (title: string, status: string) => ({ value: { title,
status }, items: [{ element: Object.assign(document.createElement("p"), {
textContent: title }) }] });

describe("lanes", () => {
  it("puts each record's rendered template in the lane of its field, lanes
in the order they first appear", () => {
    const rows = [
      { title: "Write", status: "doing" },
      { title: "Plan", status: "done" },
      { title: "Review", status: "doing" },
    ];
    const slots = { default: rows.map(({ title, status }) =>
instance(title, status)) };
    const shown = mountForTest(mount, { params: { rows, by: "status", as:
```

```
"t" }, slots });
    const lanes = [...shown.target.querySelectorAll("section")].map((lane)
=> lane.textContent);
    expect(lanes).toEqual(["doingWriteReview", "donePlan"]);
    shown.destroy();
  });
});
```

Inputs and set

Components have one output: changing an input the document binds to them. A param declared `input<number>` is given the name of an input the document declares, and the component gets that input's current value. `set("value", 4)` changes the input; the host processes the document again with the new value and updates every component that depends on it. A document using both components:

```
---
md:
  import:
    - from: "@you/rating@0"
      names: [rating, lanes]
  inputs:
    stars: number = 3
tasks:
  - {title: Write the outline, status: doing}
  - {title: Ask for a review, status: waiting}
  - {title: Draft the first part, status: doing}
---

:: rating[3 of 5 stars] {value=stars size=large}

::: lanes {rows=(md.doc.tasks) by=status as=t}
**{{t.title}}**
::: /lanes
```

`rating` gets `{ value: 3, size: "large", max: 5 }`, then 4 once the reader chooses the fourth star. `lanes` gets three instances, each rendered from `**{{t.title}}**` with `t` its record: `t` is the author's name, resolved in the document, so the component never sees the template, only what it rendered.

- **The binding is checked:** a name no input under `md.inputs` declares, or an input of another type, is F030. `input<enum>` binds an input of any enum, and `input<small |`

`large>` only one with exactly those members, in that order.

- `set` **is refused** for a param that isn't bound to an input, and during `update`: in `@compmd/render` it throws. Call it from what the reader does, as `rating` does from a click.
- **A value of the wrong type isn't the component's to send.** `set` sends any value; the host gives it to the processor as the input's value, and the spec has the processor use the input's default instead of a value that doesn't match its type (spec §8.7).
- **An input a component can change can't choose what is loaded:** a data `from` that interpolates it is F031, so component code can't send data out through it.

The test names the params bound to an input, and reads what the component set:

```
import { mountForTest } from "@compmd/pack/testing";
import { describe, expect, it } from "vitest";
import mount from "./rating.ts";

describe("rating", () => {
  it("draws its value of its max, and sets the bound input when a star is
chosen", () => {
    const shown = mountForTest(mount, { params: { value: 3, max: 5, size:
"small" }, fallback: "3 of 5 stars", inputs: ["value"] });
    const stars = () => [...shown.target.querySelectorAll("button")];
    expect(stars().map((star) => star.textContent).join("")).toBe("★★★☆☆");
    stars()[4]?.click();
    expect(shown.sets).toEqual([{ param: "value", value: 5 }]);
    shown.update({ value: 5, max: 5, size: "large", color: "#b8860b" });
    expect(stars().map((star) => star.textContent).join("")).toBe("★★★★★");
    shown.destroy();
  });
});
```

Fallbacks

Every use has a fallback, which the component gets as `fallback`: a leaf's or an inline component's text in `[…]`, a container's slots as plain Markdown. It is what a plain renderer shows, what a host shows when the component can't render, and the component's text alternative, `fallback.text`. A leaf without one is W101: plain renderers show nothing, and the component has no text alternative. Write it as a sentence that says what the component shows: `3 of 5 stars`, not `rating`.

The fallback is in a document's hands, so don't rely on it for anything but text.

How a use is checked

Hosts validate every use against its component's signature after resolving the document's imports and before loading data or running code (spec §5.3). Each diagnostic points at the param, not the whole line, and the document still renders:

Code	When	What renders
F008	a param the signature doesn't declare, and isn't a common param, `class`, `id`, `lang` or `dir`; or a breakpoint a param doesn't take (a component's own params take none)	the param is ignored
F009	a value outside its type: `max=many`, `max="5"`, an enum's non-member, an expression where a `name` or an input is bound	the param is treated as absent: its default, or the fallback when it is required
F010	a required param missing	the fallback
F024	an expression whose type isn't the param's	as F009
F030	an `input<…>` param naming no declared input, or one of another type	the fallback
F006	a slot the signature doesn't declare	its content joins the default slot
F018	a body given to a component with no slots	the body follows it as ordinary content

F008 and F009 suggest the nearest name or token when a value looks misspelt, with a fix an editor offers. One mistake gives one diagnostic: a required param whose value was refused isn't reported missing again. A document with a mistake in each use:

```
---
md:
  import:
    - from: "@you/rating@0"
      names: [rating]
  inputs:
    stars: number = 3
    note: string = ""
---

:: rating[Stars] {value=stars colour=gold}
```

```
:: rating[Stars] {value=stars max=many}

:: rating[Stars] {value=stars size=huge}

:: rating[Stars] {max=4}

:: rating[Stars] {value=note}
```

gives F008 (`rating` has no param `colour`, so it is ignored), F009 twice (`max` takes a number, `size` takes `small` or `large`), F010 (`rating` needs `value`) and F030 (`value` binds a number, but `note` is a string). The first three uses render the component, with the bad params left out; the last two render `Stars`.

Next: React, Svelte and libraries, rendering a framework inside `mount`, and what a pack bundles.

15. React, Svelte and libraries

A pack's module is plain JavaScript, but what it draws with is yours: React, a charting library, your own code. This chapter is what the sandbox's frames provide and what a pack bundles, how to render React inside `mount`, where Svelte stands, `draw` for components that take their time, and library packs, which several packs share.

What the frames provide

Every frame a host runs components in has one copy of each runtime of the spec's bindings (spec §7.6), shared by every component in it, through its import map:

Runtime	The import names a module may use
React 19	`react`, `react/jsx-runtime`
React DOM 19	`react-dom`, `react-dom/client`
Svelte 5	`svelte`, `svelte/internal/client`, `svelte/internal/disclose-version`, `svelte/internal/flags/legacy`, `svelte/internal/flags/async`, `svelte/reactivity`, `svelte/store`, `svelte/transition`, `svelte/easing`, `svelte/motion`, `svelte/animate`, `svelte/events`, `svelte/attachments`

A module never bundles a runtime: two copies of React in one frame break its hooks. `compmd-pack build` leaves exactly these names as imports, and bundles everything else, a subpath not in the table too. For each runtime a module imports, it adds the runtime to `package.json`'s `peerDependencies`, at the version installed in the pack (`"react": "^19.3.0"`), and says so when it writes it. A host whose frames have an older version reports W121 on the document's import of the pack when it resolves it, before anything renders, and the pack's components that need the runtime show their fallbacks. `@compmd/sandbox`'s frames have React 19.3 and Svelte 5.57 today.

React inside mount

A React component in a pack is rendered by its `mount`: make a root in the box, render into it on every update, unmount it on destroy. `counter` is a button that counts the reader's clicks. Its pack is the template's, with React added. Its document imports `./dist/counter.js`:

````
---
md:
  import:
    - from: ./dist/counter.js
      names: [counter]
  export:
    pack: "@you/counter"
    version: "0.1.0"
    names: [counter]
---

# @you/counter

`counter`, a leaf in React, which counts the reader's clicks: `::
counter[Cups of tea: 0] {label="Cups of tea"}`.

```compmd component=counter signature
params:
  label: string
  start: number = 0
```
````

and the build makes it from `src/counter.tsx`:

```
// `counter`: a button that counts the reader's clicks, drawn with React
inside the pack's `mount`.
import { component } from "@compmd/pack/component";
import { useState } from "react";
import { flushSync } from "react-dom";
import { createRoot } from "react-dom/client";

/** The params, as the signature in `index.comp.md` declares them. */
export interface Params {
  label: string;
  start: number;
}

function Counter({ label, start }: Params) {
  const [count, setCount] = useState(start);
  return (
    <button type="button" onClick={() => setCount(count + 1)}>
      {label}: {count}
    </button>
```

```
  );
}

export default component<Params>((target, { params }) => {
  const root = createRoot(target);
  const render = (next: Params) => {
    flushSync(() => {
      root.render(<Counter {...next} />);
    });
  };
  render(params);
  return {
    update: render,
    destroy: () => {
      root.unmount();
    },
  };
});
```

- **The same root on every update**, so React keeps the component's state: the count stays when `label` changes.
- **`flushSync`**, so the box holds what React drew when `mount` and `update` return, as the host expects; it is how `@compmd/render`'s own React binding renders.
- **JSX** compiles with React's automatic runtime, which `@compmd/pack`'s TypeScript configuration sets (`"jsx": "react-jsx"`), so the module imports `react/jsx-runtime`, which the frames have.
- **Slots** are elements the host rendered: place each item's `element` in a node React leaves alone, such as an empty `<div>` you append it to from a `ref`, and never let React render over it.
- **The rest of the context** is yours to pass down as props: `fallback.text` for an `aria-label`, `scheme` for a palette, `set` for a bound input.

React is installed for the build and the types, as the template's other tools are, `npm install -D react react-dom @types/react @types/react-dom`:

```
{
  "name": "@you/counter",
  "version": "0.1.0",
  "description": "A CompMD pack: a counter the reader clicks, in React.",
  "keywords": ["compmd", "compmd-pack"],
  "license": "MIT",
  "type": "module",
```

```
  "compmd": "./index.comp.md",
  "files": ["dist", "index.comp.md", "README.md"],
  "scripts": {
    "build": "compmd-pack build",
    "dev": "compmd-pack dev",
    "test": "compmd-pack test",
    "lint": "tsc --noEmit",
    "prepublishOnly": "compmd-pack build && compmd-pack test"
  },
  "devDependencies": {
    "@compmd/pack": "0.3.0",
    "@types/react": "19.3.0",
    "@types/react-dom": "19.3.0",
    "happy-dom": "21.0.3",
    "react": "19.3.0",
    "react-dom": "19.3.0",
    "typescript": "6.0.3",
    "vitest": "5.0.5"
  }
}
```

`npm run build` leaves `react`, `react/jsx-runtime`, `react-dom` and `react-dom/client` as the module's imports, which the frames give, and bundles the rest: your code and `@compmd/pack/component`. It adds the two runtimes to `package.json`, at the versions installed, and says so:

```
"peerDependencies": {
  "react": "^19.3.0",
  "react-dom": "^19.3.0"
}
```

Its `tsconfig.json` is the template's:

```
{ "extends": "@compmd/pack/tsconfig.json", "include": ["src"] }
```

Its test mounts it as a host does, clicks it, and updates its params. React draws the new count after `click()` returns, not during it, so the test waits for it with `expect.poll`:

```
import { mountForTest } from "@compmd/pack/testing";
import { describe, expect, it } from "vitest";
import mount from "./counter.tsx";
```

```
describe("counter", () => {
  it("counts the reader's clicks from its start, and keeps the count when
its label changes", async () => {
    const shown = mountForTest(mount, { params: { label: "Cups of tea",
start: 2 }, fallback: "Cups of tea: 2" });
    const button = () => shown.target.querySelector("button");
    expect(button()?.textContent).toBe("Cups of tea: 2");
    button()?.click();
    await expect.poll(() => button()?.textContent).toBe("Cups of tea: 3");
    shown.update({ label: "Cups of coffee", start: 2 });
    expect(button()?.textContent).toBe("Cups of coffee: 3");
    shown.destroy();
    expect(shown.target.textContent).toBe("");
  });
});
```

A document uses it as any leaf:

```
---
md:
  import:
    - from: "@you/counter@0"
      names: [counter]
---

:: counter[Cups of tea: 0] {label="Cups of tea"}
```

A component like this is what chapter 10 writes for an app; in a pack it runs in the strict sandbox, in a frame with no network, whoever imports it.

Svelte

The frames provide Svelte 5, for the Svelte components documents define in their own fences and that apps build with themselves. `compmd-pack build` has no Svelte compiler yet: it builds TypeScript and JSX with Vite's own transform, and a `.svelte` file it imports doesn't build. So a pack's components are written in plain TypeScript or React today; a pack with Svelte components waits for the build to compile them.

What a pack bundles

Each module the pack's document imports, `./dist/<name>.js`, is built from `src/<name>.ts` (or `.tsx`, `.js`, `.jsx`) into one ES module:

- **Everything it imports is in it**, but the runtimes above and the library packs the pack names in `peerDependencies` (below): the libraries from npm, your own files, `@compmd/pack/component`, which is a few KB. A bundler keeps only what the module uses.
- **One file, minified**, with no comments naming the files it was built from, and code splitting off: a library that loads parts of itself on demand stays in the one file, since a host that fetches the pack from a registry loads the module as text, where a second file has no path to load from. The build's check fails when a script is left beside the modules.
- **The licences of what it bundles** go into `dist/LICENSES.md`, or `dist/<name>.LICENSES.md` for each module of a pack with several. `files` publishes them with `dist/`.
- **Nothing from the network.** A strict frame has none, so fonts, icons and data the component needs are in the module, or come in its params.

A pack is at most 10 MB unpacked, the size a host takes by default, which the build checks. The Mermaid pack, the whole of Mermaid with its layout engines, is 5.2 MB built.

Drawing asynchronously

A component that draws with a library that takes its time, a diagram or a chart that is laid out asynchronously, would write the same code each time: the newest params win, a slow draw doesn't overwrite a newer one, the drawing follows the scheme, and a draw that can't be made calls `fail`. `draw` from `@compmd/pack/component` is that code. Its pack:

```
---
md:
  import:
    - from: ./dist/sparkline.js
      names: [sparkline]
  export:
    pack: "@you/sparkline"
    version: "0.1.0"
    names: [sparkline]
---

# @you/sparkline

`sparkline`, a leaf: `:: sparkline[Visits rose all week] {values=[3, 5, 4,
8, 12]}`.
```

````
```compmd component=sparkline signature
params:
  values: array
```
````

and its component, whose `render` gets the params and `{ scheme, signal }`, and returns an element, or SVG or HTML as text:

```
// `sparkline`: a line through a list of numbers, drawn with `draw`, which follows the page's colour scheme, lets the
// newest params win, and fails with the message of what `render` throws.
import { draw } from "@compmd/pack/component";

/** The params, as the signature in `index.comp.md` declares them. An `array` may hold anything a document gives. */
export interface Params {
  values: unknown[];
}

const WIDTH = 120;
const HEIGHT = 32;

export default draw<Params>(
  ({ values }, { scheme }) => {
    const numbers = values.filter((value): value is number => typeof value === "number" && Number.isFinite(value));
    if (numbers.length < 2 || numbers.length !== values.length) throw new Error("a sparkline needs two or more numbers, and nothing else");
    const low = Math.min(...numbers);
    const span = Math.max(...numbers) - low || 1;
    const points = numbers.map((n, i) => `${String((i * WIDTH) / (numbers.length - 1))},${String(HEIGHT - ((n - low) * HEIGHT) / span)}`);
    // Only numbers go into the markup, so a document can't write any of its own into it.
    const stroke = scheme === "dark" ? "#7ee2b8" : "#0b7a52";
    return `<svg viewBox="-2 -2 ${String(WIDTH + 4)} ${String(HEIGHT + 4)}" width="${String(WIDTH)}" height="${String(HEIGHT)}"><polyline fill="none" stroke="${stroke}" stroke-width="2" points="${points.join(" ")}"/></svg>`;
  },
  { role: "img" },
);
```

What `draw` does with it:

- `render` **may return a promise.** The drawing replaces the last one only once it is ready, so the box never flickers empty. When an update overtakes a draw, the older one is dropped and its `signal` aborted: pass the signal on to a library that takes one, or check `signal.aborted` between steps of a long draw.
- **A throw, or a rejected promise, fails** with its message: the host reports W118 and shows the fallback until the next update, which draws again. Throw for what the params can't draw, as `sparkline` does for a list with a word in it.
- **The scheme**: a change comes as an update with the same params, so `render` runs again with the new `scheme`.
- `role` gives the drawing a role, with `fallback.text` as its `aria-label`; `img` for a picture.
- `oneAtATime: true` runs one draw at a time in a frame, for a library with one global configuration, as Mermaid has, so two components' themes never mix.
- **Text is set as HTML.** A string `render` returns becomes the drawing's markup, so never put a document's text in it unescaped: `sparkline` writes only numbers it checked, and a library that renders markup should sanitize it, as Mermaid does with `securityLevel: "strict"`.

`destroy` aborts the draw in progress and empties the box. A document uses it as any leaf:

```
---
md:
  import:
    - from: "@you/sparkline@0"
      names: [sparkline]
---

:: sparkline[Visits rose from 3 to 12] {values=[3, 5, 4, 8, 12]}
```

Its tests wait for the drawing with `settled()`:

```
import { mountForTest } from "@compmd/pack/testing";
import { describe, expect, it } from "vitest";
import mount from "./sparkline.ts";

describe("sparkline", () => {
  it("draws a line labelled with its fallback, and again in the other
scheme", async () => {
    const shown = mountForTest(mount, { params: { values: [3, 5, 4, 8] },
fallback: "Visits rose all week" });
    await shown.settled();
    const image = shown.target.querySelector("[role=img]");
    expect(image?.getAttribute("aria-label")).toBe("Visits rose all week");
```

```
expect(image?.querySelector("polyline")?.getAttribute("stroke")).toBe("#0b7
a52");
    shown.setScheme("dark");
    await shown.settled();

expect(shown.target.querySelector("polyline")?.getAttribute("stroke")).toBe
("#7ee2b8");
    shown.destroy();
  });

  it("fails with what it can't draw, shows the fallback, and draws again at
the next update", async () => {
    const shown = mountForTest(mount, { params: { values: [3, "lots"] },
fallback: "Visits rose all week" });
    await shown.settled();
    expect(shown.failures).toEqual(["a sparkline needs two or more numbers,
and nothing else"]);
    expect(shown.target.textContent).toBe("Visits rose all week");
    shown.update({ values: [3, 5] });
    await shown.settled();
    expect(shown.target.querySelector("polyline")).not.toBeNull();
    shown.destroy();
  });
});
```

Its `tsconfig.json` is the template's:

```
{ "extends": "@compmd/pack/tsconfig.json", "include": ["src"] }
```

and its `package.json` the template's, with its own name and description:

```
{
  "name": "@you/sparkline",
  "version": "0.1.0",
  "description": "A CompMD pack: a sparkline of a list of numbers, in the
page's colour scheme.",
  "keywords": ["compmd", "compmd-pack"],
  "license": "MIT",
  "type": "module",
  "compmd": "./index.comp.md",
  "files": ["dist", "index.comp.md", "README.md"],
  "scripts": {
```

```
    "build": "compmd-pack build",
    "dev": "compmd-pack dev",
    "test": "compmd-pack test",
    "lint": "tsc --noEmit",
    "prepublishOnly": "compmd-pack build && compmd-pack test"
  },
  "devDependencies": {
    "@compmd/pack": "0.3.0",
    "happy-dom": "21.0.3",
    "typescript": "6.0.3",
    "vitest": "5.0.5"
  }
}
```

What component code may import

Whatever a pack bundles, its module's imports, and those of every component a host runs, resolve to four kinds of module only (spec §7.6):

An import of	Resolves to	Where it can't
a runtime, `react`, `svelte`, ...	the frame's one copy	W121 on the import, at resolve, when the pack's `peerDependencies` ask for a version the host doesn't have
a library the host offers, by its npm name (`d3`)	the host's copy, listed under `libraries` in its configuration	the component shows its fallback with W121 on a host that doesn't offer it
a library pack the pack names in `peerDependencies`	one copy for every pack in the document	W121 on the import, at resolve
the module's own files, by relative path	files in its pack	a host that loads the module as text can't load them: W121

Anything else resolves nowhere, and the component shows its fallback with W121. A library the host offers isn't portable, as an ambient pack's names aren't: bundle it, or depend on a library pack, so the pack works everywhere.

Library packs

A pack bundles every library it uses, which is right for most. A large library several packs use whole, such as three.js or KaTeX, can instead be a **library pack**: the library built as one module, which every pack in a document shares, so a frame loads one copy. Anyone may publish one.

```
npm create @compmd/pack three -- --name @kit/three --library three
cd three
npm install
npm run build
```

It writes a pack with no document, no component, no tests and no test page:

```
three/
  package.json     "compmd": { "library": "dist/index.js" }, "exports"
naming the same file, three in devDependencies
  src/index.ts     export * from "three";
  tsconfig.json    @compmd/pack's
  README.md        how a pack uses it, when to add "single", how to publish
  .gitignore
```

- `src/index.ts` **re-exports the library**: all of it, or only what packs use, to make the module smaller. A library with a default export needs `export { default } from "three"` as well. It must import one npm package, the library, or the build says so.
- **Its version is the library's.** `@kit/three@0.170.0` is `three@0.170.0`: pin `three` in `devDependencies` at the version to publish, and `npm run build` writes the installed version into `package.json`, with a warning; its check fails when they differ. Its `prepublishOnly` is the build.
- `"single": true`, beside `library` in the `compmd` field, says the library can't have two copies in one frame: it defines custom elements, patches globals, or keeps one instance by design. A pack whose range doesn't match the version the document already has then gets W121 on its import instead of a second copy, and its components that need it show their fallbacks.

A pack that uses one:

1. `npm install -D @kit/three`, for the build and the types.
2. Name it in `peerDependencies`, with the range the components work with: `"@kit/three": "^0.170"`. The build leaves every name there, and its paths, out of the module.
3. Import it by that name: `import * as THREE from "@kit/three"`.

4. For the test page, list it in a `compmd.config.json` beside `demo.comp.md`: `{ "libraries": ["@kit/three"] }`.

A host that fetches packs from a registry fetches the library pack the same way, and gives every pack in the document the newest version all their ranges allow. An app that installs its packs instead installs the library pack too, and lists it under `libraries` in its configuration, as the test page does: `@compmd/vite` serves an installed pack's modules by URL, and a frame shares a library pack only with a module it was given as code, as a pack fetched from a registry is. A peer no host can give is W121 on the import, with why.

Next: code renderers, components that draw a language's code blocks.

16. Code renderers

Some content is source in a language of its own: a diagram in Mermaid, a table as CSV, music as ABC. A document writes it as a fenced code block, which every Markdown renderer shows as code. A **code renderer** is a leaf whose signature also names a language, so it draws that language's blocks in every document that imports it, and is still a leaf for source made from data (spec §7.7). This chapter writes one that draws ```` ```csv ```` blocks as tables, then looks at the Mermaid pack.

Writing one

```
npm create @compmd/pack csv -- --name @you/csv --renders csv
```

`--renders <language>` writes the same pack as a leaf's, with three differences: the signature says `renders: csv` instead of declaring `value`, the component reads `params.code`, and `demo.comp.md` writes a ```` ```csv ```` block and the same component as a leaf. The language is a code block's language name: lowercase letters, digits, `+` and `-`, starting with a letter.

The signature

```
---
md:
  import:
    - from: ./dist/csv.js
      names: [csv]
  export:
    pack: "@you/csv"
    version: "0.1.0"
    names: [csv]
---

# @you/csv

`csv` draws every ` ```csv ` block of a document that imports it as a
table; a reader without it sees the values. Written as a leaf, its `code`
is a param, for rows made from data: `:: csv[The totals] {code=
```

````
("total,12\nopen,3")}`.

```compmd component=csv signature
renders: csv
params:
  header: boolean = true
```
````

- `renders` is a language, or a list of names of one language: the Mermaid pack's is `[mermaid, mmd]`.
- `code` is a param every code renderer has without declaring it: a required `string`, the block's text. A signature may declare it to document it, but only as a required `string` with no default; anything else is F009 on the declaration.
- **Other params** are declared as for any component, and a block gives them on its info string.
- **No slots**: a code renderer is a leaf.

The rest of the pack is the template's:

```
{
  "name": "@you/csv",
  "version": "0.1.0",
  "description": "A CompMD pack: draws ```csv blocks as tables.",
  "keywords": ["compmd", "compmd-pack"],
  "license": "MIT",
  "type": "module",
  "compmd": "./index.comp.md",
  "files": ["dist", "index.comp.md", "README.md"],
  "scripts": {
    "build": "compmd-pack build",
    "dev": "compmd-pack dev",
    "test": "compmd-pack test",
    "lint": "tsc --noEmit",
    "prepublishOnly": "compmd-pack build && compmd-pack test"
  },
  "devDependencies": {
    "@compmd/pack": "0.3.0",
    "happy-dom": "21.0.3",
    "typescript": "6.0.3",
    "vitest": "5.0.5"
  }
}
```

```
{ "extends": "@compmd/pack/tsconfig.json", "include": ["src"] }
```

The component

```
// `csv`: draws a ```csv block, its text in `code`, as a table; the first
row is its header unless `header=false`.
import { component } from "@compmd/pack/component";

/** The params, as the signature in `index.comp.md` declares them, and
`code`, which every code renderer has. */
export interface Params {
  code: string;
  header: boolean;
}

/** The rows of `code`: fields split at commas, a field in double quotes
holding commas and `""` for a quote. */
export function rowsOf(code: string): string[][] {
  return code
    .split("\n")
    .filter((line) => line.trim() !== "")
    .map((line) => [...line.matchAll(/(?:^|,)(?:"((?:[^"]|"")*)"|
([^,]*))/g)].map(([, quoted, plain]) => quoted?.replaceAll('""', '"') ??
plain?.trim() ?? ""));
}

export default component<Params>((target, { params, fallback }) => {
  const table = document.createElement("table");
  // The block's `alt`, or a leaf's fallback: the table's caption.
  const caption = Object.assign(document.createElement("caption"), {
textContent: fallback.text });
  target.append(table);
  const cells = (row: string[], tag: "th" | "td") => {
    const tr = document.createElement("tr");
    tr.append(...row.map((text) =>
Object.assign(document.createElement(tag), { textContent: text })));
    return tr;
  };
  const draw = ({ code, header }: Params) => {
    const [first = [], ...rest] = rowsOf(code);
    const head = header ? [cells(first, "th")] : [];
    const body = (header ? rest : [first, ...rest]).map((row) => cells(row,
```

```
"td"));
    table.replaceChildren(caption, ...head, ...body);
  };
  draw(params);
  return {
    update: draw,
    destroy: () => {
      table.remove();
    },
  };
});
```

It is an ordinary component in the plain binding, and the host contract is the same as a leaf's: the component can't tell a block from a leaf, and isn't told the block's language. Its cells are text, set with `textContent`, since a block's text is the document's.

In a document

A document imports the component, and writes the block as it would anywhere:

````
---
md:
  import:
    - from: "@you/csv@0"
      names: [csv]
---

# Visits

```csv {alt="Visits by city: Kyoto 12, Osaka 7"}
city,visits
Kyoto,12
Osaka,7
```

:: csv[Open and done] {code=("open,3\ndone,9") header=false}
````

- **The block's text is `code`**, as written: a code block is never interpolated.
- **The info string's param list**, after the language, gives the other params: `{header=false}`. It is read only when a code renderer renders the block; otherwise the info string stays as written, for other tools.

- `alt` **is the text alternative**, which every rendered block takes, as every component takes `class` and `id`: it is the component's `fallback.text`. Write what the block shows.
- **The block is the fallback**: a reader without the pack, a plain renderer, or a host where the component fails sees the code block.
- **The leaf** is the same component, its source the `code` param, for source made from data or inputs: `code=(…)` computes it, as `:: csv` does here. Its fallback is its text in `[…]`, as for any leaf.
- **A box**: a rendered block has an outer box, as a leaf has, so the common params apply: `` ```csv {alt="…" max-width=30rem} ``.
- **Blocks only.** A code span has no language, so a code renderer draws no inline code. A `` ```math `` block is always math, never a code renderer's.

Which component draws a block is found by its language, among the components the document can name, as a name is found: local definitions, then listed imports, then wildcard imports, then a host's ambient packs. A block whose language nothing renders stays a code block, with no diagnostic. Two code renderers of one language at the same level are F023 when a block of it is used; listing one of them by name resolves it. Renaming a code renderer on import (`{name: csv, as: table}`) renames the leaf only: `` ```csv `` blocks still render.

What is checked

As for any leaf, and the block's own rules. A document with three mistakes:

````
---
md:
  import:
    - from: "@you/csv@0"
      names: [csv]
---

```csv
city,visits
```

```csv {alt="Visits" code="a,b"}
city,visits
```

```csv {alt="Visits" header=maybe}
city,visits
```
````

gives W101 for the first block, which has no `alt`, F017 for the second, which gives `code` on its info string as well as its text, and F009 for the third, whose `header` isn't a boolean, so `header` is treated as absent and its default used.

Its tests

`mountForTest` gives the component `code` as any param, and the block's `alt` as the fallback:

```
import { mountForTest } from "@compmd/pack/testing";
import { describe, expect, it } from "vitest";
import mount, { rowsOf } from "./csv.ts";

describe("rowsOf", () => {
  it("splits fields at commas, but not inside quotes", () => {
    expect(rowsOf('city,visits\n"Kyoto, Japan",12\n')).toEqual([
      ["city", "visits"],
      ["Kyoto, Japan", "12"],
    ]);
  });
});

describe("csv", () => {
  it("draws the block's rows under its header, captioned with its alt, and
follows new params", () => {
    const shown = mountForTest(mount, { params: { code:
"city,visits\nKyoto,12", header: true }, fallback: "Visits by city" });
    expect(shown.target.querySelector("caption")?.textContent).toBe("Visits
by city");
    expect([...shown.target.querySelectorAll("th")].map((cell) =>
cell.textContent)).toEqual(["city", "visits"]);
    shown.update({ code: "Kyoto,12\nOsaka,7", header: false });
    expect(shown.target.querySelectorAll("th")).toHaveLength(0);
    expect(shown.target.querySelectorAll("tr")).toHaveLength(2);
    shown.destroy();
    expect(shown.target.children).toHaveLength(0);
  });
});
```

A real one: the Mermaid pack

`@comp-markdown/mermaid` draws Mermaid diagrams in any document that imports it, with no support from the host. Its signature, in its document:

```
renders: [mermaid, mmd]
params:
  theme: auto | default | dark | neutral | forest = auto
```

Its component, `src/mermaid.ts`, is written with `draw` (chapter 15):

- **Mermaid is bundled in**, pinned at 12.2.0, with everything it loads: one module of 5.2 MB minified, under the 10 MB a host takes.
- **One draw at a time** (`oneAtATime`), since Mermaid's configuration is global: each draw calls `mermaid.initialize` with its theme, then `mermaid.render`.
- **`theme: auto`** draws with Mermaid's dark theme when `scheme` is `dark` and its default theme otherwise, and again when the scheme changes; another theme a block gives (`{theme=forest}`) is used as given.
- **Mermaid's own markup is sanitized by it**, with `securityLevel: "strict"`, since `draw` sets the SVG it returns as markup.
- **A diagram Mermaid can't read** throws `Mermaid couldn't read this diagram:` and the first line of Mermaid's message: the host reports W118, which an editor shows on the block, and the reader sees the code until it is fixed.

A document draws a block and a diagram made from its data with one import:

````
---
md:
  import:
    - from: "@comp-markdown/mermaid@0"
      names: [mermaid]
steps: [Draft, Review, Published]
---

```mermaid {alt="Draft, then review, then published"}
graph LR
  Draft   Draft --> Review  Review --> Published
```

:: mermaid[The steps: draft, review, published] {code=("graph TD; " +
md.doc.steps.join("  + md.doc.steps.join(" --> "))}
````

Its unit tests stub Mermaid, which happy-dom can't lay out, and check the theme for each scheme and the failure; its browser tests in the repository draw real diagrams in Chrome and WebKit.

Next: testing and publishing, from a pack's tests to npm, and to an app that uses it.

17. Testing and publishing

A pack is ready to publish when its components are tested, its build has passed the checks a host would make, and its `package.json` says what it is. This chapter takes a small pack, a progress bar, from its tests to npm, and then to an app that uses it.

The pack

`npm create @compmd/pack` writes a pack like this one (chapter 13): a component in TypeScript, its signature in the pack's document, its tests, and a `package.json` whose scripts run `@compmd/pack`'s command, which builds, checks, tests and serves the pack, so the pack holds only what is its own.

```
import { component } from "@compmd/pack/component";

/** The params, as the signature in `index.comp.md` declares them. */
export interface Params {
  value: number;
  max: number;
}

export default component<Params>((target, { params, fallback }) => {
  const bar = document.createElement("progress");
  // A screen reader hears the fallback, which says in words what the bar
shows.
  bar.setAttribute("aria-label", fallback.text);
  const numbers = document.createElement("span");
  target.append(bar, " ", numbers);
  const draw = ({ value, max }: Params) => {
    bar.max = max;
    bar.value = Math.min(Math.max(value, 0), max);
    numbers.textContent = `${String(value)} of ${String(max)}`;
  };
  draw(params);
  return {
    update: draw,
    destroy: () => {
      bar.remove();
      numbers.remove();
```

```
    },
  };
});
```

The pack's document imports the component's built module, exports it under the pack's name and version, and declares its signature, which hosts check every use against before anything runs:

````
---
md:
  import:
    - from: ./dist/progress.js
      names: [progress]
  export:
    pack: "@you/progress"
    version: "0.1.0"
    names: [progress]
---

# @you/progress

`progress`, a leaf: `:: progress[3 of 4 done] {value=3 max=4}`.

```compmd component=progress signature
params:
  value: number
  max: number = 100
```
````

Its `tsconfig.json` is one line, `@compmd/pack`'s:

```
{ "extends": "@compmd/pack/tsconfig.json", "include": ["src"] }
```

Testing a component

`@compmd/pack/testing`'s `mountForTest` runs a component as a host does, through `@compmd/render`'s component layer, with the params, the fallback and the colour scheme you give it:

```
import { mountForTest } from "@compmd/pack/testing";
import { describe, expect, it } from "vitest";
import mount from "./progress.ts";

describe("progress", () => {
  it("draws its value of its max, with its fallback as its label", () => {
    const shown = mountForTest(mount, { params: { value: 3, max: 4 },
fallback: "3 of 4 done" });
    const bar = shown.target.querySelector("progress");
    expect(bar?.value).toBe(3);
    expect(bar?.max).toBe(4);
    expect(bar?.getAttribute("aria-label")).toBe("3 of 4 done");
    expect(shown.target.textContent).toContain("3 of 4");
  });

  it("follows new params, keeps a value outside its range inside the bar,
and goes when destroyed", () => {
    const shown = mountForTest(mount, { params: { value: 1, max: 10 },
fallback: "1 of 10" });
    shown.update({ value: 12, max: 10 });
    expect(shown.target.querySelector("progress")?.value).toBe(10);
    expect(shown.target.textContent).toContain("12 of 10");
    shown.destroy();
    expect(shown.target.querySelector("progress")).toBeNull();
  });
});
```

`npm test` runs `compmd-pack test`: every `src/**/*.test.ts` in happy-dom, with no test configuration in the pack. A test has the handle's `target`, what the component `set` (`sets`, for the params you name in `inputs`), what it failed with (`failures`), `setScheme` for an update in the other colour scheme, and, for a component that draws asynchronously, `settled()`, which waits for its drawing.

Params come from documents, so test what a document may give: here a value past its max stays inside the bar. A signature checks types and the values an enum allows, never that two params agree.

`npm run dev` serves the test page: a `demo.comp.md` beside the pack's document, which the template writes, rendered in the strict sandbox as a host renders it, light and dark. It shows what the tests can't see, such as how the bar looks.

What the build checks

`npm run build` runs `compmd-pack build`: each component's module, built from `src/` into `dist/` as one minified ES module, with every library bundled in but the runtimes the frames give (React, React DOM and Svelte, which it adds to `peerDependencies` at the version it was built with). Then it checks the pack as a host would, and fails, saying what to fix, when:

- the document's `export.version` isn't `package.json`'s version: a host reports `F022`;
- a file the document imports isn't there, or isn't in `package.json`'s `files`, so it wouldn't be published;
- a module isn't one file: a host that loads it as text can't load its chunks;
- the pack is over 10 MB unpacked, the size a host takes by default;
- a library pack's version isn't its library's.

A pack's `package.json`

```
{
  "name": "@you/progress",
  "version": "0.1.0",
  "description": "A CompMD pack: a progress bar with its numbers.",
  "keywords": ["compmd", "compmd-pack"],
  "license": "MIT",
  "type": "module",
  "compmd": "./index.comp.md",
  "files": ["dist", "index.comp.md", "README.md"],
  "scripts": {
    "build": "compmd-pack build",
    "dev": "compmd-pack dev",
    "test": "compmd-pack test",
    "prepublishOnly": "compmd-pack build && compmd-pack test"
  },
  "devDependencies": {
    "@compmd/pack": "^0.3.0"
  }
}
```

What makes it a pack on a registry (*CompMD packs on a registry*, section 2):

- **A scoped name**, `@you/progress`, the name documents import. No prefix is needed; a scope with a library of the same name may call its pack `@you/compmd-progress`.
- **The `compmd` field**, naming the pack's document, even when it is the default `index.comp.md`: a registry keeps every field, so a tool can tell a pack from other packages

without downloading it.

- **The keyword** `compmd-pack`, which only packs use, so a search for `keywords:compmd-pack` finds them. Tools that help build packs use `compmd`.
- `files`: what a host reads, the built modules, the document and the files it imports, and the README for npm's page; never the sources or the tests, so the download stays small.
- `dependencies` are never installed by a host: what the components use is in their modules, and the packs the pack imports are in its document's `md.import`, which the host resolves.

Publishing

1. Raise the version in `package.json` and in the document's `export.version`, the same in both.
2. `npm login`, then `npm publish --access public`. `prepublishOnly` builds, checks and tests first, so a pack that fails never goes out.

That's a pack on npm. For **provenance**, a record of where and from what it was built, which npm shows on the pack's page, publish from CI with npm's trusted publishing instead: no token is stored anywhere, GitHub proves to npm which repository and workflow publishes. On npmjs.com, the pack's settings name the workflow, under **Trusted publisher**; the workflow needs `id-token: write`, and npm 11.5.1 or later:

```yaml
name: Release
on:
  push:
    tags: ["v*"]
permissions:
  contents: read
  id-token: write
jobs:
  release:
    runs-on: ubuntu-24.04
    steps:
      - uses: actions/checkout@v7
      - uses: actions/setup-node@v7
        with:
          node-version: 24
      - run: npm install --global npm@11
      - run: npm ci
      - run: npm publish --access public
```

npm adds the provenance when it publishes with a trusted publisher. A package must exist before a trusted publisher can be set up for it, so publish its first version by hand.

Each version's tarball has a hash, its integrity, which npm publishes (`npm view @you/progress@0.1.0 dist.integrity`). A document that pins the version writes it as its import's `integrity`, and a host refuses a pack that doesn't match.

Sharing a large library

A pack bundles every library its components use, which is right for most: a bundler keeps only what the pack imports. A large library several packs use whole, such as three.js or KaTeX, can instead be shared, so a sandbox loads one copy for every pack in a document:

- **A library pack** holds the library, built as one module: `npm create @compmd/pack three --name @kit/three --library three` writes one, with `"compmd": { "library": "dist/index.js" }` and the library's version as its own. Anyone may publish one, the library's own authors included.
- **A pack that uses it** names it in `peerDependencies` (`"@kit/three": "^0.170"`) and imports it by that name; the build leaves it out of the module.
- **A host** that doesn't offer a library of that name fetches it from its registries like a pack, and gives every pack in a document the newest version all their ranges allow. A library that can't have two copies in one sandbox says so, with `"single": true`; a pack asking for a version the document can't have then gets `W121`, and its components that don't need the library still render.

A host adding the pack

An app that wants the pack installs it, `npm install @you/progress`, and its documents import it, `from: "@you/progress@0"`, then write `:: progress[3 of 4 done] {value=3 max=4}`: with `@compmd/vite`, nothing else (chapter 9). An app with registries in its configuration doesn't even install it: it fetches the pack when a document imports it. Either way it runs strict, unless the app's configuration lists it for more.

Next: Part V, beyond Vite, starting with other hosts.

18. Other hosts

`@compmd/vite` does four jobs for a Vite app: it reads the configuration, gives the processor the app's packs, builds and serves the sandbox's frame documents, and serves the packs' built modules. An app built another way does them itself, with the same functions the plugin uses. This chapter builds such an app: a Node server with no Vite plugin, which serves a page, its documents, the frames, a pack's built module and the registry proxy, and a page that draws a diagram with the Mermaid pack.

What you do by hand

The plugin's job	Without it
reads and checks `compmd.config.json`	`readConfig` and `policyOf`, from `@compmd/vite/config`
builds the frame documents into `compmd/`	`buildFrames`, and `buildLibraries` for the libraries you offer, from `@compmd/sandbox/build`
serves `compmd/` with its two headers	your server
finds installed packs and gives them to the processor (`request.packs`)	`PackRelease`s you make from the packs' files
serves the packs' built modules, and gives the sandbox their hashes (`sandbox.modules`)	your server, and a `modules` function you write
serves the registry proxy	`registryProxy`, with `toNodeListener` for Node
builds trusted components with the app (`sandbox.trusted`)	your bundler, and the bindings of `@compmd/render`

The example has six files: the configuration, a build script for the frames, the server, the page, its HTML, and a document.

```
npm install @compmd/processor @compmd/sandbox @compmd/registry @compmd/vite
@comp-markdown/mermaid
```

`@compmd/vite` is here only for its configuration functions, which need no Vite to run; npm installs Vite with it all the same, as a peer dependency.

The configuration

The same `compmd.config.json` a Vite app writes, here with one pack listed, only listed packs allowed, and npm as the registry for packs the app doesn't install:

```
{
  "packs": { "@comp-markdown/mermaid": {} },
  "imports": "listed",
  "registries": { "*": "https://registry.npmjs.org/" },
  "fetch": "host"
}
```

`@compmd/vite/config` reads it without Vite or Node, so it works on a server, in a worker or in a page:

- **`readConfig(value, where?)`** checks the parsed JSON and gives it with every default filled in, sizes in bytes and origins normalized (`ReadConfig`). What is wrong throws a `ConfigError`, whose `problems` lists every problem at once and whose message starts with `where` (`compmd.config.json` by default). The configuration's reference has every key and every problem.
- **`policyOf(config, nameOf, refused?)`** gives what the processor takes: `policy` (the packs with their trust, `imports`, `deny`, and `refused` when you give it) and `ambient`, the names of the ambient packs. `nameOf` turns a key into a pack name: a pack name is its own, and a folder pack (`./components`) is its `package.json`'s `name`. `refused` holds the packs you leave out, each with why; the plugin puts there the packs larger than `maxPackSize`.

The module also exports `isFolder(key)`, which tells a folder key from a pack name, and the types `Config`, `ReadConfig`, `PackOptions`, `Library` and `TrustLevel`.

The frame documents

The sandbox's frames load documents your app serves: `strict.html` and `permissive.html`, each with its content security policy and import map, the frame runtime, and the runtimes of the spec's bindings (React 19, React DOM, Svelte 5). `buildFrames` writes them, at build time, in Node:

```
// Builds the sandbox's frame documents into the folder the server serves
them from: `node build.ts public/compmd`.
```

```
import { buildFrames } from "@compmd/sandbox/build";

const [out = "public/compmd"] = process.argv.slice(2);
const { runtimes } = await buildFrames({ out });
console.log(`The frames are in ${out}, with ${runtimes.map(({ name, version
}) => `${name} ${version}`).join(", ")}.`);
```

It gives the frames' **library definition**, the runtimes and libraries they have, each with its version and import names, and writes it beside them as `library-definition.json`, which `libraryDefinition(out)` reads back, as the server below does.

To offer libraries besides the runtimes, as the configuration's `libraries` does, build them first, then the frames with their import map entries. The order matters: `buildFrames` copies the runtimes' files over the folder, so read the libraries before it does:

```
import { buildFrames, buildLibraries, readLibraries, RUNTIMES } from
"@compmd/sandbox/build";

const imports = await buildLibraries([{ name: "d3", imports: ["d3"] }], {
root, out: "public/compmd/libs", shared: RUNTIMES });
const libraries = await readLibraries("public/compmd/libs");
const definition = await buildFrames({ out: "public/compmd", imports,
libraries });
```

`buildLibraries` bundles each library with Vite's bundler, so it needs Vite installed; `buildFrames` doesn't. `root` is the folder whose `node_modules` holds the libraries, and the entries it gives start with `./libs/`, so the libraries go in `libs/` beside the frame documents. The package also exports the frames' policies, `strictPolicy` and `permissivePolicy`, which `buildFrames` writes into the documents; a host never needs to write one itself.

The server

```
// The app's server, without Vite: the page and its documents, the
sandbox's frames, the packs' built modules and the
// registry proxy. `node server.ts public 8080 compmd.config.json`
import { createHash } from "node:crypto";
import { readFile } from "node:fs/promises";
import { createServer, type ServerResponse } from "node:http";
import { dirname, extname, join } from "node:path";
import { fileURLToPath } from "node:url";
import type { PackRelease } from "@compmd/processor";
```

```
import { NpmRegistry, registryProxy } from "@compmd/registry";
import { toNodeListener } from "@compmd/registry/node";
import { libraryDefinition } from "@compmd/sandbox/build";
import { policyOf, readConfig } from "@compmd/vite/config";

const [root = "public", port = "8080", configFile = "compmd.config.json"] =
process.argv.slice(2);

// The configuration, checked as the Vite plugin checks it, and translated
for the processor, with what the frames have.
const config = readConfig(JSON.parse(await readFile(configFile, "utf8")));
const { policy, ambient } = policyOf(config, (key) => key);
const { runtimes, libraries } = await libraryDefinition(join(root,
"compmd"));
const named = ({ name, version }: { name: string; version: string }) => ({
name, version });

// An installed pack, given by hand: its manifest, its document, and its
built module as empty text, which the frames load.
const pack = dirname(fileURLToPath(import.meta.resolve("@comp-
markdown/mermaid/package.json")));
const manifest = await readFile(join(pack, "package.json"), "utf8");
const { name, version } = JSON.parse(manifest) as { name: string; version:
string };
const files = { "package.json": manifest, "index.comp.md": await
readFile(join(pack, "index.comp.md"), "utf8"), "dist/mermaid.js": "" };
const release: PackRelease = { version, integrity: null, files };
const place = `${name}@${version}/dist/mermaid.js`;
const module = await readFile(join(pack, "dist/mermaid.js"));

// What the page needs to open a document: the request's packs and policy,
and each built module's hash.
const app = JSON.stringify({
  request: { packs: { [name]: [release] }, ambient, policy: { ...policy,
runtimes: runtimes.map(named), libraries: libraries.map(named) } },
  modules: { [place]:
`sha384-${createHash("sha384").update(module).digest("base64")}` },
});

// The registries' proxy, when the configuration names registries and the
host fetches from them.
const registries =
Object.fromEntries(Object.entries(config.registries).map(([scope, url]) =>
[scope, new NpmRegistry({ url })]));
```

```
const proxy = config.fetch === "host" && Object.keys(registries).length > 0
? toNodeListener(registryProxy({ registries, base: "/compmd/registry/" }))
: undefined;

const TYPES: Record<string, string> = { ".html": "text/html", ".js":
"text/javascript", ".json": "application/json", ".css": "text/css",
".wasm": "application/wasm", ".md": "text/markdown" };

/** The frames have an opaque origin, so their files load cross-origin;
their documents are sandboxed even opened alone. */
const framed = (path: string): Record<string, string> => ({ "access-
control-allow-origin": "*", ...(path.endsWith(".html") ? { "content-
security-policy": "sandbox allow-scripts" } : {}) });

function send(response: ServerResponse, path: string, body: string |
Buffer, headers: Record<string, string> = {}): void {
  response.writeHead(200, { "content-type": TYPES[extname(path)] ??
"application/octet-stream", ...headers }).end(body);
}

createServer((request, response) => {
  // The URL's path, its `..` segments resolved, so a file is always under
`root`.
  const path = new URL(request.url ?? "/", "http://localhost").pathname;
  if (proxy && path.startsWith("/compmd/registry/")) return proxy(request,
response);
  if (path === "/app.json") return send(response, path, app);
  if (path === `/compmd/packs/${place}`) return send(response, path,
module, framed(path));
  const file = join(root, path === "/" ? "index.html" : path);
  void readFile(file).then(
    (body) => send(response, file, body, path.startsWith("/compmd/") ?
framed(path) : {}),
    () => response.writeHead(404).end(),
  );
}).listen(Number(port), () => console.log(`http://localhost:${port}/`));
```

What it does, part by part:

- **The policy.** `policyOf` gives the policy and the ambient packs; the server adds the frames' runtimes and libraries, by name and version, as `policy.runtimes` and `policy.libraries`. With them, the processor reports a component on a runtime the frames don't have (W102) and a pack whose `peerDependencies` they can't meet (W121). Without them, it assumes every runtime and checks no library.

- **The headers.** Every file under `compmd/` is served with `Access-Control-Allow-Origin: *`, since a frame has an opaque origin and loads its files cross-origin. The frame documents also get `Content-Security-Policy: sandbox allow-scripts`, so a frame document opened on its own, outside a frame, is sandboxed all the same. These are the headers the Vite plugin's servers send.
- **`/app.json`** is what `virtual:compmd` holds in a Vite app: the request's `packs`, `ambient` and `policy`, and each built module's hash. A server that renders many documents makes it once, as here.

Packs, by hand

The processor never reads a disk or a registry: a pack is in the request's `packs`, by name, as a list of `PackRelease`s:

- **`version`**, the release's version, from its `package.json`.
- **`files`**: `package.json`, the pack's document (its `compmd` field names it, `index.comp.md` by default), and every file the document imports, by path in the pack. A built module (a `.js` file the document imports) can be empty text: the processor needs only to know it is there, and names it by its place in the components it gives (`module: { pack, version, path }`), for the sandbox to load. Source files (`.jsx`, `.tsx`, `.svelte`, `.ts`) need their text, which the sandbox compiles.
- **`integrity`**: the hash the registry published for the version's tarball, `sha512-…`. A document that pins its import with `integrity` gets the pack only if the two are equal (F035 otherwise). The plugin reads it from the app's lock file; this server gives `null`, so a pinned import of the pack fails.
- **`broken`**: why the files can't be used, when you found they don't match their integrity. An import of the release is then F035.

What the plugin also does with installed packs, and this server leaves out: it checks each listed pack is installed at the version its `version` range asks, leaves out packs larger than `maxPackSize` (as `refused`), and processes each listed pack's document once at build, so a broken pack stops the build. A host that wants those checks makes them; the processor still reports every problem a document meets.

The page

The page opens a document with what the server gives, and renders it in frame mode. `modules` tells the sandbox where each built module is and its file's hash, which its frame checks before the module runs:

```
import { init, OpenDocument, type Request } from "@compmd/processor";
import { createSandbox } from "@compmd/sandbox";

await init();

// What the server gives every document, and the document itself.
const app = (await (await fetch("/app.json")).json()) as { request:
Omit<Request, "path" | "now">; modules: Record<string, string> };
const text = await (await fetch("/plan.comp.md")).text();
const opened = new OpenDocument({ ...app.request, path: "plan.comp.md",
files: { "plan.comp.md": text }, now: new Date().toISOString() });

// A built module loads in its frame from the server, checked against its
file's hash.
const sandbox = createSandbox({
  base: "/compmd/",
  modules: ({ pack, version, path }) => {
    const place = `${pack}@${version}/${path}`;
    const integrity = app.modules[place];
    return integrity === undefined ? undefined : { url: new
URL(`/compmd/packs/${place}`, location.href).href, integrity };
  },
});
const view = sandbox.open(document.getElementById("page") as HTMLElement, {
components: opened.components });
view.render(opened.expanded);
```

`base` is where the frame documents are served, ending in `/`; `modules` gives `{ url, integrity }` for a module served by URL, `{ code }` (a `Blob`) for one the page holds already, checked (as `resolvePacks` gives them, chapter 9), or `undefined` for none, whose uses show their fallbacks. The frame loads a module by URL with `<script integrity>`, so a file that doesn't match its hash never runs.

The HTML:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>CompMD on a server of its own</title>
  </head>
  <body>
    <main id="page" style="height: 24rem"></main>
```

```
    <script type="module" src="./page.ts"></script>
  </body>
</html>
```

And the document, which imports the Mermaid pack the server gives:

```
---
md:
  import:
    - from: "@comp-markdown/mermaid@0"
      names: [mermaid]
---

# Where a page's parts come from

The page, its documents and the sandbox's frames come from one server:

```mermaid {alt="The server gives the page, the frames and the built
modules"}
graph LR
  Server   Server --> Page
  Server   Server --> Frames
  Server   Server --> Modules
```
```

Try it in the playground.

Run it

`public/` is what the server serves: the bundled page, the frames under `public/compmd/`, and the documents.

1. Bundle `index.html` and `page.ts` into `public/` with the bundler your app uses. The processor's `init` loads its WebAssembly file, `compmd_wasm_bg.wasm`, from beside its own module, so the bundler must copy that file with the code; `@compmd/processor` exports it as `@compmd/processor/compmd_wasm_bg.wasm` for a bundler or server that needs its path.
2. Copy `plan.comp.md` into `public/`.
3. `node build.ts public/compmd`, after the bundler, which may empty `public/`.
4. `node server.ts public 8080 compmd.config.json`, and open the address it prints.

The page shows the heading, the sentence and the diagram, drawn in a strict frame by a module the server served. The guides' tests run these steps with Vite as the bundler alone,

with no CompMD plugin, and check each claim of this chapter.

The registry's proxy in any server

With `registries` and `"fetch": "host"`, a page fetches packs it doesn't have only through its own server, so the registry learns nothing about readers (chapter 9). `registryProxy({ registries, base })` from `@compmd/registry` is the proxy, as a fetch handler, `(request: Request) => Promise<Response>`:

- `GET <base><name>` gives the pack's short metadata from its scope's registry, each tarball's URL rewritten to the proxy's own;
- `GET <base>-/tarball/<name>/<version>` streams that version's tarball;
- every other request is 404, and a registry that fails is 502.

It passes on nothing the reader sent, cookies least of all, and caches nothing itself, so put an HTTP cache or a CDN in front of it. A private registry's token goes in its `NpmRegistry`'s `headers`, on the server. `toNodeListener` from `@compmd/registry/node` turns any such handler into a Node request listener, for `http.createServer` as here, or for Express and Connect middleware; a worker or a server with fetch handlers (Deno, Bun, Cloudflare Workers) uses the handler as it is:

```
import { NpmRegistry, registryProxy } from "@compmd/registry";

export default { fetch: registryProxy({ registries: { "*": new
NpmRegistry({ url: "https://registry.npmjs.org/" }) }, base:
"/compmd/registry/" }) };
```

The page then resolves its documents' packs with registries that point at the proxy, `{ "*": new NpmRegistry({ url: "/compmd/registry/" }) }`, as `virtual:compmd`'s `registry` does in a Vite app.

Trusted components

The plugin builds an app's own trusted components with the app and gives them to the sandbox as `trusted`. Without it, import each component in your page, compiled by your bundler, and wrap it in its binding: `react` from `@compmd/render/react` for React, `svelte` from `@compmd/render/svelte` for Svelte (`@compmd/vite/react` and `@compmd/vite/svelte` give the same), a plain mount as it is. Then open the sandbox in host mode, `createSandbox({ base, page: "host", trusted: { stars: react(Stars) }, modules })`, and give the processor the folder pack as a release with `{ level: "trusted" }` in its policy. The sandbox's README has every option, and chapter 10 the components themselves.

Next: the Rust crate, processing documents outside a browser.

19. The Rust crate

The processor is a Rust crate, `compmd`; `@compmd/processor` is that crate compiled to WebAssembly. In Rust, it processes documents outside a browser: in a build, a server, a native app, or CI. This chapter shows its stages, what a host gives it, its diagnostics, and a program that checks a folder's documents and fails on an error. The crate's README and its API documentation go further.

Getting it

The crate is `compmd` on crates.io, with `serde_json` beside it for the trees as JSON:

```
cargo add compmd serde_json
```

Its parser, a fork of markdown-rs, comes with it as the crate `compmd-markdown`.

Versions. The crate shares its version with the npm packages, and each release publishes both (RELEASING.md): this chapter is written for 0.3.0, as the rest of the guide is. Its source is the folder `processor/` of the repository, and its API documentation is on docs.rs.

The stages

A `Processor` goes through the spec's stages as states: each stage takes the processor and gives it back in its next state, so they run only in order, and each state gives what its stage made and the diagnostics so far. Each stage takes only what it uses. From the crate's documentation:

```
use compmd::context::RenderContext;
use compmd::expand::Options;
use compmd::{Ambient, ParseOptions, Processor};
use std::collections::HashMap;
use std::rc::Rc;

let text = "---\nteam: [Ana, Kenji]\n---\n\n::: each {in=(md.doc.team) as=p
join=inline sep=\", \"}\n{{p}}\n::: /each\n";
let files: HashMap<String, String> = HashMap::new();
let context = RenderContext { now: 0, tz: "UTC".into(), time: Rc::new(()),
```

```
inputs: Default::default() };
let processor = Processor::new("team.comp.md", text)
    .parse(&ParseOptions::default())
    .resolve(files, (), &Ambient::default())
    .validate(&[])
    .load(context, (), ())
    .expand(&Options::default());
assert!(processor.diagnostics().is_empty());
let json = serde_json::to_value(processor.expanded()).unwrap();
assert_eq!(json["children"][0]["children"][0]["value"], "Ana, Kenji");
```

Call	Stage	Takes
`Processor::new(path, text)`		the document's path under its root folder, and its text
`.parse(&ParseOptions)`	1, parse	`ParseOptions { metadata }`: the document's metadata, when a host keeps it apart from the text
`.resolve(files, packs, &Ambient)`	2, resolve	the root folder's files, the packs, and the ambient packs (`Ambient { packs, report }`, `report` for W113)
`.resolve_with(files, packs, &Ambient, &Policy)`	2, resolve	the same, with the host's policy for packs
`.validate(&allowed_origins)`	3, validate	the origins remote data may come from, `https://api.example.com`
`.load(context, fetch, data)`	4a, load	the render context, remote data, `host:` data
`.expand(&expand::Options)`	4b, evaluate	the options; `origins: true` puts an origin on every expanded node, and `mode` is `Mode::Lenient` (the default) or `Mode::Strict`

`RenderContext` is the render: `now` in nanoseconds since 1970 (`compmd::context::parse_time` gives it from RFC 3339 text), the processor's time zone `tz`, the host's time zones `time`, and the values given for the document's `inputs`, as JSON by name.

Each state gives what its stage made: `tree()` after parsing (the parse tree, spec §12.1), `inputs()` after loading, and after expanding `expanded()` (the expanded tree), `host_components()` (each host-language component with its code, for a renderer), `failed()` (whether strict mode failed), `cache_keys()` (each data import's final URL or path, which a

host keys its cached copies by), `frontmatter()`, `components()` and `hint_at(offset)` for editors. Every tree serializes, with serde, to the spec's JSON. From the expanded state, `reload(context, fetch, data)` renders again for other inputs or data without the first three stages, and `edit(&TextEdit)` takes an edit of the text and redoes only the blocks it touches, with `splice()` saying what changed. The README lists them all.

What the host gives

The processor does no I/O. What it reads comes through the traits of `compmd::host`, each stage taking only those it uses:

Trait	Gives	In memory	None
`Files`	`file(path)`: a file under the document's root folder	`HashMap<String, String>`	`()`
`Packs`	`releases(pack)` and `pack_file(pack, version, path)`	`HashMap<String, Vec<LoadedRelease>>`	`()`
`Fetch`	`fetch(url)`: what the host fetched for a remote URL, a `Fetched { text, content_type }`	`HashMap<String, Fetched>`	`()`: every URL fails
`Data`	`data(source)`: a `host:` source's data, as JSON	`HashMap<String, serde_json::Value>`	`()`
`Time`	`offset(zone, at)`: a time zone's offset from UTC, in seconds		`()`: only `UTC`

The crate carries no time zone data; a host that gives `md.tz` other zones implements `Time` with its own. Each trait is also implemented for an `Rc` of it, so a host that keeps using its files gives the processor a clone. A host of its own implements a trait directly, as this pack store of one pack does, from the crate's documentation:

```
use compmd::host::{Packs, Release};
use semver::Version;

/// One pack, `@acme/kit@1.0.0`, with its manifest alone.
struct Kit;

impl Packs for Kit {
    fn releases(&self, pack: &str) -> Vec<Release> {
        let kit = Release { version: Version::new(1, 0, 0), integrity:
None, broken: None };
```

```
        if pack == "@acme/kit" { vec![kit] } else { Vec::new() }
    }

    fn pack_file(&self, pack: &str, version: &Version, path: &str) ->
Option<String> {
        let found = pack == "@acme/kit" && *version == Version::new(1, 0,
0) && path == "package.json";
        found.then(|| r#"{"name": "@acme/kit", "version":
"1.0.0"}"#.to_string())
    }
}
```

A `Release` is a version, the `integrity` its registry published (an import's pinned `integrity` must equal it, F035 otherwise), and `broken`, why its files can't be used. A `LoadedRelease` is a release with its files, and reads from the same JSON as `@compmd/processor`'s `PackRelease`, so a host can keep its packs in one format for both.

The policy. `compmd::policy::Policy` is the request's `policy`, and reads from the same JSON, which `policyOf` gives from a configuration (chapter 18). From the crate's documentation:

```
use compmd::policy::{Imports, Policy, Trust};

let policy: Policy = serde_json::from_str(r#"{
    "packs": { "@acme/maps": { "level": "permissive", "network":
["https://tiles.example.com"] } },
    "imports": "listed",
    "deny": [{ "pack": "@bad/pack", "version": "<2", "reason": "a flaw
found in 1.x" }],
    "refused": { "@huge/pack": "larger than this host serves" },
    "runtimes": [{ "name": "react", "version": "19.3.0" }, { "name":
"svelte", "version": "5.57.2" }],
    "libraries": [{ "name": "d3", "version": "7.9.0" }]
}"#).unwrap();
assert_eq!(policy.imports, Imports::Listed);
assert_eq!(policy.trust("@acme/charts"), Trust::Strict);
```

What to fetch first. The processor fetches nothing. After validating, `sources(&context)` says what a render needs that the host hasn't given: the remote URLs, those of origins it doesn't allow (`disallowed`, for a host that asks its reader), the `host:` sources, and the packs no release matches (each `NeededPack` with its range and pinned integrity), for a host that fetches them, then loads; `compmd::host::newest_matching(range, versions)` picks a version as the processor will.

Diagnostics

`processor.diagnostics()` gives every stage's so far, in stage order, each a `compmd::diagnostic::Diagnostic`:

- `code`, a `Code` (`Code::F004`, written `F004`), whose `severity()` is `Severity::Error` for an F code and `Severity::Warning` for a W code;
- `message`, for people;
- `file`, the document it is in when it isn't the processed one; `range`, its `start` and maybe `end` (`Location { line, column }`, from 1, columns in code points); `related` places; `stage`, a `Stage`; and `fix`, an edit, when the message suggests one.

A `Diagnostic` serializes to the spec's machine form, the JSON `@compmd/processor` gives. Every code is in the diagnostics' appendix. Two things differ from WebAssembly: the crate takes host metadata (`ParseOptions::metadata`), so it reports W115, and its `Code` has no W104, which no package reports yet.

Checking documents in CI

The spec's strict mode fails on any F code: for CI and publishing. It reports the same diagnostics and makes the same tree as lenient mode; `failed()` says whether it failed. A program that checks every document among the files it is given, prints each diagnostic, and fails on an error:

```
use compmd::context::RenderContext;
use compmd::diagnostic::Mode;
use compmd::expand::Options;
use compmd::{Ambient, ParseOptions, Processor};
use std::collections::HashMap;
use std::process::ExitCode;
use std::rc::Rc;

/// The diagnostics of the document at `path`, each as `file:line:column:
code message`, and whether it failed.
fn check(path: &str, files: &HashMap<String, String>) -> (Vec<String>,
bool) {
    let context = RenderContext { now: 0, tz: "UTC".into(), time:
Rc::new(()), inputs: Default::default() };
    let strict = Options { mode: Mode::Strict, ..Options::default() };
    let processor = Processor::new(path, files[path].as_str())
        .parse(&ParseOptions::default())
        .resolve(files.clone(), (), &Ambient::default())
        .validate(&[])
```

```
        .load(context, (), ())
        .expand(&strict);
    let lines = processor
        .diagnostics()
        .iter()
        .map(|d| format!("{}:{}:{}: {} {}",
d.file.as_deref().unwrap_or(path), d.range.start.line,
d.range.start.column, d.code, d.message))
        .collect();
    (lines, processor.failed())
}

fn main() -> ExitCode {
    // Every file of the folder by its path: the documents, and the files
they import and read.
    let files: HashMap<String, String> = std::env::args()
        .skip(1)
        .filter_map(|path| Some((path.clone(),
std::fs::read_to_string(&path).ok()?)))
        .collect();
    let mut failed = false;
    for path in files.keys().filter(|path| path.ends_with(".comp.md")) {
        let (lines, failed_here) = check(path, &files);
        lines.iter().for_each(|line| eprintln!("{line}"));
        failed |= failed_here;
    }
    if failed { ExitCode::FAILURE } else { ExitCode::SUCCESS }
}
```

Run it from the documents' root folder with their paths as `git ls-files` prints them, relative to it, so imports and data files resolve: `cargo run --release -- $(git ls-files docs)`. The render time is fixed (1970, in UTC) so the check gives the same result every run; a document whose content depends on the date, through `md.today()`, gets a fixed one.

What it leaves to you:

- **Packs.** It gives none, so every pack import is F026. Give the packs your documents use as a `HashMap<String, Vec<LoadedRelease>>`, read from their folders or from JSON, and your app's policy with `resolve_with`.
- **Remote and `host:` data.** None is given, so a remote import loads its `fallback` or is F026, and a URL's origin is F028 unless you list it in `validate`. A check that should reach the network fetches what `sources` lists first.
- **Warnings** are printed and pass. Fail on them too, if your documents should have none.

The same check runs in Node with `@compmd/processor`'s `process` and `mode: "strict"`, which gives the same diagnostics and `failed`; the Vite plugin checks a listed pack's documents that way when an app builds (chapter 20).

What the crate doesn't do

It implements stages 1 to 4b, and is a "compmd/1 processor with remote data and origins" in the spec's conformance terms. It doesn't run host-language components: it gives their code (`host_components`) to a renderer that does, such as `@compmd/sandbox` in a browser, and W118, found only by running one, is a renderer's. It doesn't analyse a component's code against its signature fence (W104). And it renders no HTML, but for what a plain CommonMark renderer shows, `compmd::plain::html`.

Next: the appendices, starting with the configuration's reference.

20. The configuration's reference

Every key of `compmd.config.json` and every option of `@compmd/vite`'s plugin: its type, its default, an example, and what is checked. Then what the plugin checks when the app builds, and the errors it gives. Chapter 9 teaches the configuration; this is the list to look things up in.

Where it is

The configuration is one of these, never both:

- **`compmd.config.json`**, beside the app's `package.json`;
- **the `compmd` key of `package.json`**, as an object. A string there is a pack's document (a pack's own `package.json` names it so), never a configuration;
- **the plugin's `config` option**, which takes the place of both, for a tool that serves a page of its own.

None is an empty configuration: every installed pack is available, strict, and nothing more. The schema, `@compmd/vite/compmd.config.schema.json`, gives an editor completion and checks as you type:

```
{ "$schema": "./node_modules/@compmd/vite/compmd.config.schema.json" }
```

`readConfig` from `@compmd/vite/config` checks it as the plugin does, without Vite, for any host (chapter 18). It finds every problem at once, and throws a `ConfigError` whose `problems` lists them.

The keys

`packs`

An object: by pack name, `@scope/pack`, or by folder path, `./components` or `../shared/ui`, each with its options (`{}` for none). Default: `{}`.

```
{ "packs": { "@acme/maps": { "trust": "permissive", "network":
["https://tiles.example.com"] }, "./components": { "trust": "trusted",
```

```
"ambient": true } } }
```

List a pack only to give it more than the strict sandbox, to make it ambient, or to check its version: every installed pack is available without it. A name must be scoped; a folder is relative to the app's root (Vite's `root`), and must be a folder holding a pack, a `package.json` with a `name` and a document. A folder pack's name is its `package.json`'s `name`, which the policy uses, and which its document's `export.pack` must equal (F022). Under `"imports": "listed"`, these are the packs a document may import. Becomes `policy.packs`, and `ambient`.

`packs.*.trust`

`"strict"`, `"permissive"` or `"trusted"`. Default: `"strict"`.

```
{ "packs": { "@acme/reviews": { "trust": "permissive", "host": true } } }
```

- **`strict`**: a sandbox with no network, which sees only its params, slots and box.
- **`permissive`**: a frame of its own, with the network to `network`'s origins, and `host` if `host` is `true`.
- **`trusted`**: in the app's page, built with the app: for a folder pack only. A pack name with `trusted` is an error, since a registry pack is never the app's own code.

A document can never raise a pack's trust (chapter 8).

`packs.*.network`

A list of origins a permissive pack's code may reach, each `https://host[:port]` (`http:` only for `localhost`, `127.0.0.1` and `[::1]`). Default: none.

```
{ "packs": { "@acme/maps": { "trust": "permissive", "network":
["https://tiles.example.com", "https://tiles.example.com:8443"] } } }
```

An origin alone: no path, query or credentials, no wildcard. Only a permissive pack takes it.

`packs.*.host`

`true` or `false`: whether a permissive pack's components get `host`, the calls the app gives them (`createSandbox`'s `host`). Default: `false`. Only a permissive pack takes it.

```
{ "packs": { "@acme/reviews": { "trust": "permissive", "host": true } } }
```

`packs.*.ambient`

`true` or `false`: whether every document gets the pack's names without importing it. Default: `false`.

```
{ "packs": { "./components": { "trust": "trusted", "ambient": true } } }
```

An ambient pack's names have the lowest priority, so a document's own definitions and imports win. A document may still import it by name, to say what it depends on. Becomes the request's `ambient`.

`packs.*.version`

A semver range the installed version must satisfy. Default: none, any version.

```
{ "packs": { "@kit/charts": { "version": "^2.0.0" } } }
```

Checked when the app builds: a pack at a version outside it stops the build. It says nothing about which version a document gets; the document's import does.

`imports`

`"any"` or `"listed"`. Default: `"any"`.

```
{ "imports": "listed" }
```

`any`: a document may import any pack the app has, installed or from its registries. `listed`: only the packs in `packs` and the ambient ones; any other import is F026, saying the host allows only the packs its configuration lists. Becomes `policy.imports`.

`deny`

A list of `{ "pack", "version"?, "reason" }`: packs the app won't serve, all versions or those in the semver range `version`, with the reason readers see. Default: `[]`.

```
{ "deny": [{ "pack": "@bad/pack", "version": "<2", "reason": "a flaw found
in 1.x" }] }
```

`pack` is a scoped name; `reason` isn't empty. A denied import is F026, with the reason. A pack from a registry that is denied is never downloaded. `deny` is the app's own policy, not what

keeps readers safe: the sandbox does that for every pack. Becomes `policy.deny`.

`maxPackSize`

A size: a number of bytes, or a string, a number and `B`, `KB`, `MB` or `GB` (`"5 MB"`, `"2.5 MB"`; a kilobyte is 1,000 bytes). Default: no limit for installed packs; 10 MB for a pack from a registry.

```
{ "maxPackSize": "5 MB" }
```

- **An installed pack** whose files (its `package.json`, its documents and what they import) are larger is left out: its imports are F026, saying why, and the plugin warns when it builds. Becomes `policy.refused`.
- **A pack from a registry** is checked by the unpacked size its registry's metadata states, before it is downloaded; one larger, or whose metadata states no size, isn't fetched, and is F026. Becomes `registry.maxPackSize`.

`libraries`

A list of npm packages the app installs, which the frames offer component code by name, besides React, React DOM and Svelte: a name, `"d3"`, or `{ "name", "imports" }` for a package whose components import more names than its own. Default: `[]`.

```
{ "libraries": ["d3", { "name": "d3-scale", "imports": ["d3-scale", "d3-scale-chromatic"] }] }
```

Each one is built once into `compmd/libs/`, and the frames' import map gives components that copy. A library that isn't installed stops the build. Their names and versions go in the request's `policy.libraries`, so the processor reports a pack whose `peerDependencies` the app can't meet (W121). A document whose components import an offered library works only in hosts that offer it.

`registries`

An object: by scope, `"@kit"`, with `"*"` for every other scope, each the URL of an npm registry, named in full. Default: none, so a document's import of a pack the app doesn't install is F026.

```
{ "registries": { "@kit": "https://npm.kit.example/", "*": "https://registry.npmjs.org/" } }
```

Each URL is `https:` (`http:` only for `localhost`, `127.0.0.1` and `[::1]`), without credentials, query or fragment: a private registry's token belongs to the app's proxy, in its `NpmRegistry`'s `headers`. A `/` is added at its end when it has none. Neither a document nor a pack can name a registry. Becomes `virtual:compmd`'s `registry`, which `resolvePacks` takes (chapter 9).

`fetch`

`"host"` or `"direct"`: who reaches the registries. Default: `"host"`.

```
{ "registries": { "*": "https://registry.npmjs.org/" }, "fetch": "direct" }
```

- **`host`**: the page reaches the registries only through the app's proxy, at `compmd/registry/`, so they learn nothing about readers. The dev server and `vite preview` serve it; a deployed app serves it from its server or a worker (chapter 18).
- **`direct`**: the reader's browser fetches from the registries named, without credentials, which then learn the reader's address and the names and versions of the packs a document imports. For an app with no server; tell your readers.

The plugin's options

`compmd(options)`, in `vite.config.ts`. Both are optional.

`folder`

The folder of the app, under Vite's `base`, where the frame documents, their libraries, the packs' built modules and the registry proxy are served. Default: `"compmd/"`.

```
compmd({ folder: "sandbox/" });
```

Leading slashes are dropped and a trailing one added: `"/sandbox"` is `sandbox/`. `virtual:compmd`'s `sandbox.base` and `registry` follow it.

`config`

The configuration itself, as the parsed JSON, in place of `compmd.config.json` and `package.json`'s `compmd` key: for a tool that serves a page of its own, as `@compmd/pack`'s test page does. Default: none, and the plugin reads the files.

```
compmd({ config: { packs: { "./": {} } } });
```

It is checked as the file is, and its problems are listed under *the plugin's* `config`; folder packs in it are relative to Vite's root.

What the plugin checks

At the start of every build and of the dev server, and in dev again when `compmd.config.json`, `package.json` or a file of a pack changes:

1. **The configuration**, with every problem at once (below).
2. **The files.** A configuration in both `compmd.config.json` and `package.json` is an error (*keep one*), and so is a file that isn't JSON.
3. **The packs.** Each listed pack is installed (`@kit/charts` *is listed in the configuration, but it isn't installed:* `npm install @kit/charts`), at the version its `version` asks (`@kit/charts` *is 1.4.0, which isn't* `^2.0.0` *as the configuration asks*); each listed folder is a folder of the app, holding a pack; each library is installed; and each file a pack's document imports is in the pack, and doesn't leave it.
4. **The listed packs' documents**, through the processor, in one document that imports all of each: an error stops the build (*the listed packs have errors*, each as `file:line:column: code message`), a warning is a Vite warning. F026 is a warning there, since it is how a left-out pack shows. A pack left out by `maxPackSize` is a warning too.

This configuration has a problem in nearly every line:

```
{
  "packs": {
    "@acme/maps": { "trust": "trusted", "network":
["https://tiles.example.com/v2"] },
    "charts": {}
  },
  "imports": "some",
  "maxPackSize": "lots"
}
```

The plugin, and `readConfig`, give them all:

```
compmd.config.json:
- packs["@acme/maps"].trust: only a folder in the app may be trusted, never
a registry pack; a registry pack that needs more is `permissive`
- packs["@acme/maps"].network: only a permissive pack reaches the network
- packs["@acme/maps"].network: `https://tiles.example.com/v2` must be an
origin alone, without a path
- packs["charts"]: a pack is a scoped name (`@scope/pack`) or a folder path
```

```
(`./components`)
- imports: `any` or `listed`
- maxPackSize: bytes, or a size such as `"5 MB"`
```

A key the configuration doesn't have, or an option a pack doesn't take, is a problem too, whose message names the keys it takes.

In a build, a problem stops it. On the dev server, a change that brings one is logged, `[compmd] …`, and the app keeps what it had until the problem is fixed.

The plugin finds installed packs among the app's `dependencies` and `devDependencies`: each scoped one whose `package.json` has a `compmd` field or whose folder has `index.comp.md`. Each one's integrity comes from the nearest lock file above the app, `pnpm-lock.yaml` or `package-lock.json`; without one, a document that pins a pack's integrity gets F035.

Next: the diagnostics.

21. The diagnostics

Every diagnostic in the spec's catalogue, as the packages report it: its code, its severity, the stage that finds it, what causes it, who causes it, and what fixes it. Writers have their own appendix, with an example of each: the diagnostics and their fixes in the Author's Guide. This one is for the developer whose app shows them, or whose pack causes them.

Reading a diagnostic

A diagnostic is the spec's machine form, as `process`, `OpenDocument` and the crate give it:

Field	What it is
`code`	`F…` for an error, `W…` for a warning
`severity`	`error` for an F code, `warning` for a W code
`message`	what is wrong, for people; not part of the spec, and it may change
`file`	the document it is in, when it isn't the one processed: a path, or `@scope/pack@version/path` for a pack's
`range`	where: `start` and maybe `end`, lines and columns from 1, columns in code points
`related`	other places, innermost first: an unclosed container's opener, the invocation that led into a pack
`stage`	`parse`, `resolve`, `validate`, `load` or `evaluate`
`fix`	an edit that fixes it, when the message suggests one: the text at a range on one line becomes `text`

An editor turns a diagnostic into an LSP diagnostic and its `fix` into a quick fix; columns are code points, so convert them to UTF-16 for an editor that counts that way. Errors don't stop processing: the processor recovers as the spec says (section 10.3) and gives the whole tree, fallbacks where something failed. The spec calls that lenient mode, for previews; strict mode, for CI and publishing, fails on any F code: the request's `mode: "strict"` gives the same diagnostics and tree, with `failed` set (the Rust crate has a check that uses it).

Who causes it, in the tables: a **writer**, in the document's text; a **pack author**, in a pack's files or `package.json`, which the document's writer can't fix but can work around (another

version, another pack); a **host**, the app, by its configuration or by what it gives the processor. A writer can cause a pack-author code by writing a pack.

The guides' tests make each code below with a small document and check the processor reports it at the stage this appendix gives.

Stage 1, `parse`

The document's text alone: no imports, signatures or data.

Code	Severity	What causes it	Who	What fixes it
F001	error	`md.version` names a major version the processor doesn't implement; the file is read as plain CommonMark and GFM	writer	`version: 1`, or a processor that implements it
F003	error	a closer names no open container while one is open, usually a misspelt name	writer	the closer's name, `::: /box`
F004	error	a container isn't closed before an outer closer or the end of the file	writer	its closer, in the same list item or quote
F005	error	a closer while no container is open	writer	removing it, or the opener it was meant for
F007	error	a named slot twice in one container	writer	one slot line per slot
F011	error	a broken param list: an unterminated string, an unbalanced bracket, invalid JSON, a bare value with `(` or `{`, an empty value; also on a rendered code block's info string, found in validation	writer	the quote or bracket; quoting the value, or an expression `(…)`
F012	error	a continuation line, `::: + …`, not right after an opener, a leaf or another continuation	writer	moving it under its opener
F015	error	a slot line outside any container	writer	moving it into its container

Code	Severity	What causes it	Who	What fixes it
F016	error	a line that starts with `::` but is no CompMD form, such as `:::box` or `::: @kit/card`	writer	the form the message names, `::: box`
F017	error	a param given twice, `id` with `#name`, or a code renderer's `code` on a code block's info string	writer	one of them
F032	error	a closer names a container opened outside its list item or quote	writer	the closer at the container's own level
F033	error	frontmatter that isn't YAML 1.2 core: tags, anchors, aliases or a repeated key; the file is read as plain CommonMark and GFM	writer	plain YAML
F036	error	a param list on a signature fence, which is ignored	writer, pack author	moving the params to the definition fence
W101	warning	a leaf without fallback text, or a rendered code block without `alt`, found in validation	writer	`[the sentence it stands for]`, or `alt="…"`
W106	warning	`{{` followed by something that isn't a path; kept as text	writer	a path, `\{{` for literal braces, or `:value` for a computed value
W107	warning	an inline component, a space, then a param list, which is text	writer	the param list right after `]`
W108	warning	a container opener or leaf right after a paragraph line	writer	a blank line before it
W110	warning	a later minor version than the processor implements	writer, host	a newer processor, or an earlier version
W111	warning	an unknown key under `md`, or an unknown name in `md.syntax`; ignored	writer	the key the message suggests

Code	Severity	What causes it	Who	What fixes it
W116	warning	a line that starts with `::` inside an HTML block, which makes it HTML	writer	a blank line after the HTML's opening line

Stage 2, `resolve`

The imports, packs and names. What a pack's files cause is reported in the pack's file (`@scope/pack@version/path`), with the importing document's invocation as `related`.

Code	Severity	What causes it	Who	What fixes it
F002	error	an unknown component name, or a signature fence with nothing to attach to; the message suggests the nearest name	writer	the name, or the import that gives it
F013	error	a name declared twice, a keyword (`if`, `each`, `value`, `box`) defined, imported or renamed to, `md` declared, a CEL reserved word or type name as a param, `as` name or input, or two params differing only in `-` and `_`; found in validation for names in a body	writer, pack author	another name
F019	error	a component whose definition invokes itself, directly, through others, or through a code block it renders	writer, pack author	breaking the cycle
F020	error	an import cycle	writer, pack author	removing the import that closes it
F021	error	an import names something its source doesn't export	writer	the name the source exports, or `"*"`
F022	error	a pack's `export.pack` or `export.version` disagrees with its `package.json`	pack author	the two the same; `compmd-pack build` checks it

Code	Severity	What causes it	Who	What fixes it
F023	error	a name, or a code block's language, that two wildcard imports, or one and `@compmd/core`, give is used; for a code block's language found in validation	writer	listing the name in one import
F026	error	an import that can't be loaded: a missing file, no release matching the version, an unscoped name, `@compmd/*` other than core, a path leaving its root, or a pack the host's policy doesn't allow, denies, or left out (its size); for data, found when loading. Never for a pack or data source the request names as `pending`: it waits for the host to give up	writer, host	the path or version; for a pack the host refuses, the host's configuration, or another pack
F029	error	an exported host-language component without a signature fence	pack author	its signature fence
F035	error	a pack whose release doesn't match the import's `integrity`, or that the host found broken	writer, host	the integrity npm publishes for that version, or the host giving the release's real integrity
F037	error	a declaration's type isn't in the type grammar, in a signature, an input or a data `schema`; it takes any value	writer, pack author	the type the message suggests
W105	warning	a local host-language component, or an imported component file (`./Chart.jsx`), without a signature fence, whose params can't be typed	writer	a signature fence
W112	warning	a local definition or explicit import shadows a `@compmd/core` component	writer	another name, if it wasn't meant

Code	Severity	What causes it	Who	What fixes it
W113	warning	a name from an ambient pack is used, when the host asks for it (`reportAmbient`); for a `host:` data source, found when loading	host	nothing: it tells a writer the document isn't portable
W117	warning	an exported host-language component without `runtime=`	pack author	`runtime=react@19` or `svelte@5` on its fence
W121	warning	a pack's `peerDependencies` names a runtime or library the host doesn't have in that range, or a library pack that can't be had (see also rendering, below)	pack author, host	the host offering it, a pack range it meets, or the library pack published

Stage 3, `validate`

Every use against its signature, the paths, the expressions' types, the inputs and the data sources.

Code	Severity	What causes it	Who	What fixes it
F006	error	a slot its component doesn't declare, whose content joins the default slot; or content before the first slot line of a component without a default slot, which follows the component	writer	a declared slot
F008	error	an unknown param, or a breakpoint on a param that takes none	writer	the param the message suggests
F009	error	a param value that doesn't match its type or grammar, or a `grid-area` naming no area of its box's literal `grid-template-areas`, and, found in resolving, a code renderer's `code` declared as anything but a required `string`, or an import's `integrity` on a range instead of an exact version	writer, pack author	a value of the type, which the message lists

Code	Severity	What causes it	Who	What fixes it
F010	error	a required param missing, or an input without a default, which then starts empty	writer	the param, or the input's default
F014	error	an unknown interpolation path	writer	the path, or `\{{` for literal braces
F018	error	a body given to a component that declares no slots	writer	the component as a leaf
F028	error	remote data from an origin the host doesn't allow (`allowedOrigins`), or that the reader hasn't allowed yet (`loadData`)	writer, host	the host or the reader allowing the origin, or a `fallback`
F030	error	a bound param or input component's `bind` names an undeclared input, or one of another type	writer	the input declared, of that type
F031	error	a `from` that interpolates something other than `md.input` and `md.doc`, an input a component can set, or a URL's scheme or host	writer	literal scheme and host; values in the path and query only
W102	warning	a use of a component whose definition language or runtime the processor doesn't run (`vue`, or a runtime the host's `policy.runtimes` doesn't have); its fallback is shown	writer, pack author, host	a supported language and runtime, or the runtime in the frames
W103	warning	an `each` whose default slot is one list item, with `join` unset	writer	`join=list`
W120	warning	a box param given to a `display=contents` component, which has no box; ignored	writer, pack author	removing it

Stage 4a, `load`

The inputs' values and the data.

Code	Severity	What causes it	Who	What fixes it
F027	error	a data record that violates its `schema`, or a CSV row with the wrong number of cells; the record is dropped	writer	the data, or the schema

F026 for data is found here too: a file that is missing or unparseable, a `host:` source the host doesn't give, a CSV header with an empty or repeated name.

Stage 4b, `evaluate`

Expanding the tree, for one render context.

Code	Severity	What causes it	Who	What fixes it
F024	error	an expression's parse, type or evaluation error, or a list or a map without `name` written into text; type errors are found in validation	writer	the expression; a `schema` for the data it reads
F025	error	an `each` iteration that isn't the shape its `join` needs	writer	the iteration's content, or another `join`
F034	error	an expansion larger than the processor allows, past 100,000 nodes	writer	less nesting, or filtering the data
W109	warning	an interpolated link or image URL with a scheme other than `https:`, `http:` or `mailto:`; shown without the link	writer	the data's URLs
W114	warning	a box whose blocks use breakpoints, with no `width` where its width would come from its content	writer, pack author	a `width` on the box
W119	warning	a box that places blocks by position, with no `height`, `aspect-ratio` or `fill` at some width	writer, pack author	one of them on the box

When rendering

What only running a component finds. `@compmd/sandbox` reports it in the view's `error` event, `{ component, reason, code?, message }`, with `code` from `diagnosticOf(reason)`:

`reason`	`code`	What happened
`threw`	W118	the component threw while mounting or updating, called `fail`, or its frame stopped answering
`failed`	W121	its code didn't compile or load: an import it may not make, a compile error, a built module that doesn't match its hash
`missing`	none	the request gave no code for it; the processor reported why
`not-here`	none	a component of a level its place can't hold: a trusted or permissive one in frame mode, or in a strict component's slot; this is the sandbox's design

Code	Severity	What causes it	Who	What fixes it
W118	warning	a host-language component threw, failed or hung; its fallback is shown	pack author	the component; a test with `mountForTest` (chapter 17)

W121 at rendering is the same code as above: the sandbox reports a module that can't load, and the processor a pack's `peerDependencies`.

Only in the Rust crate

Code	Severity	What causes it	Who	What fixes it
W115	warning	the host gave metadata for a document that has its own frontmatter, which wins; at line 1, in parsing	host	giving metadata only for documents without frontmatter

The crate takes a document's metadata apart from its text (`ParseOptions::metadata`); `@compmd/processor`'s request has no field for it, so a host on WebAssembly can't give metadata and never sees W115.

In the catalogue, not reported

Code	Severity	What causes it	Who	What fixes it
W104	warning	a signature fence and its component disagree: a prop, type or slot	pack author	the fence and the component the same

W104 needs the spec's *component analysis* feature, reading a component's code for its props, which no package implements yet: a signature fence is believed as it is written. W113 for a library the host offers, which the spec allows, isn't reported either.

Next: a glossary.

22. A glossary

The terms a developer meets in this guide and the packages, each in a sentence or two, with where the spec defines it and the chapter that teaches it. The spec's own list is its section 1.1; where a term here is one of those, this is the same meaning, said for building.

Term	What it is	Spec	Taught in
Ambient pack	A pack every document gets the names of without importing it, at the lowest priority, so a document's own names win. Its names aren't portable: another host doesn't have them (W113 when the host asks).	§6.3	9, 20
Binding	How the host contract maps to one framework: plain JavaScript, React or Svelte 5. `@compmd/render` has `react` and `svelte`; a plain module's default export is `mount` itself.	§7.6	15
Built module	A component file in a pack that is a built ES module, `.js`, one file with its libraries bundled in but its runtime and its `peerDependencies`. The sandbox loads it by its place, checked against its file's hash.	§6.4	17, 18
Code renderer	A leaf whose signature `renders` a language, so it draws that language's code blocks, the block's text as its `code` param, and is still a leaf.	§7.7	16
Configuration	`compmd.config.json`, or `package.json`'s `compmd` key: the app's policy for packs, which only the app sets, never a document or a pack.		9, 20
Diagnostic	What the processor, or a renderer, found wrong, by a code from the spec's catalogue: `F…` an error, `W…` a warning, with where it is and the stage that found it.	§10	4, 21

Term	What it is	Spec	Taught in
Expanded tree	What the processor gives a renderer: the document's content only, every definition, `if`, `each`, `value` and interpolation replaced by what it gives, for one render context. It names host components but holds no code.	§12.3	3, 7
Fallback	What shows where a component can't render: a leaf's or inline component's text in `[…]`, a container's slots as plain Markdown, a rendered code block's code. A component gets it as `fallback`, as its text alternative.	§4.5	8
Frame documents	`strict.html` and `permissive.html`, with the frame runtime and the runtimes' modules, which a host serves (under `compmd/`) for the sandbox's frames, with `Access-Control-Allow-Origin: *` and, for the documents, `Content-Security-Policy: sandbox allow-scripts`. `buildFrames` writes them.		11, 18
Frame mode	The sandbox's default: the whole page in one strict frame, for documents from anyone. Permissive and trusted components show their fallbacks there.		3, 11
Host	The application whose processor renders documents: it gives the processor everything (files, packs, data, the time), sets each pack's trust in its configuration, and serves the frames.	§1.1	3
Host component	A component defined in a host language (`js`, `ts`, `jsx`, `tsx`, `svelte`), which a renderer runs. The processor gives each with its code, or its built module's place, and its trust (`HostComponent`, `OpenDocument.components`).	§7.1	10
Host contract	How a host component is run, whatever its framework: `mount(target, context)`, with its params, slots, `set`, fallback, heading level, colour scheme, `fail` and maybe `host`, then `update` and `destroy`.	§7.4	13, 14

Term	What it is	Spec	Taught in
Host mode	The sandbox's other mode, `page: "host"`: the page in the app's page, trusted components inline, every other component in a frame of its own level. For an app with components of its own, or permissive packs.		10, 11
`host`	What the app gives the components of permissive packs it lists with `"host": true`: calls to its own data or actions. A document can't ask for it.	§7.4	10
Input	A typed value the reader changes, declared in a document's `md.inputs`, read as `md.input.<name>`. The host keeps its values and gives them in the request; the page tells the host with `set`.	§8.7	6
Integrity	The hash a registry publishes for a version's tarball, `sha512-…`. A document that pins an exact version may write it as its import's `integrity`, and gets the pack only if the release's matches (F035). Different from a built module's file hash, `sha384-…`, which the sandbox checks.	§6.1	9, 17
Library definition	What the frames provide component code: the runtimes and the libraries the host offers, each with its version and import names. `buildFrames` gives it; its names and versions are the processor's `policy.runtimes` and `policy.libraries`.	§7.6	18
Library pack	A package that holds one library several packs share, such as three.js, so a sandbox loads one copy: `"compmd": { "library": "dist/index.js" }`, no document. A pack names it in `peerDependencies`.	the companion, §5	17
Observer	What the sandbox gives developer tools when the app's page asks, with `view.observe`: the view's frames, the messages between them, the components mounted and their fallbacks. Nothing a document or a frame sends turns it on.		12

Term	What it is	Spec	Taught in
Origin	On an expanded node, when the request asks for `origins`: the range of the document that made it, with its occurrence, so an editor can keep what it rendered through an edit.	§12.3	5
Pack	A document, or a published package, that exports components. Published, it is an npm package whose `package.json` names its document in a `compmd` field.	§6	9, 13
Place	Where a built module is in its pack: the pack, its version and the file's path (`@kit/charts@2.1.0/dist/chart.js`). The processor names a module by it, and the sandbox's `modules` turns it into a URL and a hash, or the file.	§6.4	18
Policy	The processor's form of the configuration, the request's `policy`: each listed pack's trust, `imports`, `deny`, `refused`, and the runtimes and libraries the host has. `policyOf` makes it.	§6.4	18
Proxy	The app's own route to its registries, `compmd/registry/`, so a page fetches packs without telling a registry who reads what. `registryProxy` is one.	the companion, §4	9, 18
Registry	A server that speaks npm's registry protocol: npm's own, a mirror, a private one, or a proxy. Only the host's configuration names one.	the companion, §1	9
Registry pack	A pack published to a registry as an npm package, with a scoped name, the `compmd` field and the keyword `compmd-pack`.	the companion, §2	17
Render context	What a render depends on besides the document, its imports and its data: the time, the time zone and the inputs' values. The same context gives the same expanded tree.	§8.5	4

Term	What it is	Spec	Taught in
Request	Everything the processor needs for one document, since it does no I/O: `path` and `files`, `packs`, `ambient`, `policy`, `allowedOrigins`, `remote` and `host` data, `now`, `tz`, `inputs`, `origins`. `virtual:compmd`'s `request` holds the app's part.		3, 4
Runtime	A binding's framework at a major version, `react@19` or `svelte@5`, which the frames provide, one copy for every component, so component code imports it and never bundles it.	§7.6	15
Sandbox	Where a component's code runs when it isn't trusted: a frame that sees only its contract, with no network unless its pack is permissive. `@compmd/sandbox` makes them.	§15	8, 11
Signature	A component's params, with their types and defaults, and its slots. A host component's is a signature fence, `compmd component=<name> signature`, which every exported one needs (F029).	§7.3	14
Slot	A named part of a container's body, `::#name`; the content before the first is the default slot. The renderer renders slots and gives them to the component. A **template slot** is rendered once per element of an array param.	§4.2, §7.5	14
Splice	What an edit changed in the expanded tree: top-level nodes `from` to `to` replaced by `nodes`, and how far the rest moved. An editor's preview applies it instead of the whole tree.		5
Stage	One step of processing: parse, resolve, validate, load, evaluate (expand), then render. Each needs only what it lists, and finds its own diagnostics.	§11	3, 19

Term	What it is	Spec	Taught in
Trust level	How far a pack's code is trusted, which the host's configuration sets and a document can never raise. **Strict**, the default: a sandbox with no network. **Permissive**: a sandbox of its own, with the network to the origins listed, and `host` if given. **Trusted**: the app's own code, in its page, for a folder of the app only.	§15	8
`virtual:compmd`	The module `@compmd/vite` gives the app: `request` for the processor, `sandbox` for `createSandbox`, and `registry` for `resolvePacks`. A host without the plugin makes the same by hand.		2, 18

Next: back to the contents.