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

CompMD Author's Guide

How to write in CompMD: documents that read as Markdown and render as pages, with boxes, grids, values from data, inputs the reader changes, and components of your own, written in CompMD itself. It is for writers, not programmers: no code is needed. Building CompMD into an application is the Developer's Guide.

The playground's *Learn CompMD* tour is the five-minute introduction, hands on; this book is what you keep open while writing. Every example in it opens in the playground, where you can change it and watch the page.

Who it is for

People who write: notes, newsletters, reports, documentation, anything you would write in Markdown and want to do more with. You don't need to program. The book uses a few small formulas, to compute a total or choose a color, and explains each one where it first appears. It covers the syntax, data, inputs, layout, components written in CompMD itself, and the packs a document uses.

Components written in code, in React, Svelte or JavaScript, are the Developer's Guide's subject. So are applications that show CompMD documents. This book says only what such a component looks like to the document that uses it.

Contents

Part I. Writing

1. A first document
2. The Markdown you know
3. Containers and params
4. Values and expressions
5. Showing and repeating

Part II. Data and inputs

6. Data from files
7. Inputs the reader changes
8. Dates, numbers and lists

Part III. Layout

9. Boxes, rows, stacks and grids
10. Widths and breakpoints
11. Placing blocks

Part IV. Components of your own

12. Definitions
13. Slots
14. Template slots
15. Signatures and fallbacks

Part V. Packs

16. Using a pack
17. Code blocks a pack draws
18. Components in code, in brief

Appendices

19. The syntax at a glance
20. The diagnostics and their fixes

The examples

Each chapter's document is a file in `examples/`, which the guides' tests process and compare with the chapter's text. Every CompMD block in the book processes without a diagnostic, so what you copy works.

1. A first document

A CompMD document is a Markdown file with a little more in it: values from its own header, and components, such as a box around a paragraph. In this chapter you write one, see it as a page, and see what it looks like where CompMD isn't known.

The book follows one document as it grows: the newsletter of a community garden. Each chapter adds to it, and each version opens in the playground, where you can change it and watch the page.

Where to write

Anywhere you write Markdown: a text editor, a notes app, a repository on GitHub. A CompMD file ends in `.comp.md`, so tools know what it is.

To see the page as you type, use the playground: the text on the left, the page on the right. Its *Learn CompMD* tour is a quick, hands-on introduction to everything this book covers.

The document

Save this as `garden.comp.md`:

```
---
md:
  version: 1
title: Elm Street Garden
issue: November 2026
---

# {{md.doc.title}}

News from the plots, {{md.doc.issue}}.

::: box {padding=m border=subtle border-radius=m}

**Work day:** Saturday, November 7, from nine to noon. Bring gloves; we
bring the tea.

::: /box
```

```
The compost bins are full again, so the next load goes to the bins by the
shed.
```

Try it in the playground.

The page shows the heading *Elm Street Garden*, the line *News from the plots, November 2026.*, the work day in a box with a thin border and rounded corners, and the paragraph about compost.

Its parts

The frontmatter is the part between the two `---` lines at the top. It is YAML: a key, a colon, and a value, one per line. Here it says three things:

- `md:` holds CompMD's own settings. `version: 1` says which version of CompMD the document is written in. With the `md` key, the file stays CompMD wherever it goes, even renamed to `.md`.
- `title` and `issue` are yours. Any key that isn't `md` is the author's, and you can name it what you like.

`version` is one of a few settings under `md`. The others come up in the chapters ahead:

Under `md`	Says	Chapter
`version`	the version of CompMD, `1`	this one
`lang`	the document's language, such as `en` or `de`, for dates and numbers	4
`dir`	the direction the text runs, `ltr`, `rtl` or `auto`; by default the direction of `lang`'s script	3
`tz`	the time zone of "today", such as `Europe/Berlin`	8
`syntax`	extra syntax the document turns on, such as wiki links	2
`data`	files and addresses the document reads values from	6
`inputs`	values the reader changes	7
`import`	the packs the document uses	16

A key under `md` that CompMD doesn't know is a warning that names the nearest one it does, so a misspelt `lnag` doesn't go unnoticed.

`{{md.doc.title}}` puts the value of your `title` key where it stands. `md.doc` is your frontmatter, so `{{md.doc.issue}}` is *November 2026*. Change the value at the top, and every place that uses it follows. Chapter 4 says more about values.

The box is a *component*: a piece of the page with a name, here `box`. It starts with a line of its own, `::: box`, and ends with `::: /box`. What's between the two lines is its body, ordinary Markdown; the blank lines around it are there for plain renderers, as the next section shows. What's in the braces, `{padding=m border=subtle border-radius=m}`, are its *params*: a medium padding, a subtle border and medium rounded corners. Chapter 3 is about components and params.

Everything else is Markdown, as you know it: a heading, paragraphs, bold text. Chapter 2 says what Markdown you can use.

Where CompMD isn't known

A CompMD file is a Markdown file, so it reads well where nobody knows CompMD: on GitHub, in an editor's preview, in any Markdown viewer. Such a *plain renderer* shows the heading, the paragraphs and the bold text as usual, and the rest as you wrote it. The box's lines are short paragraphs of their own, around the work day:

```
::: box {padding=m border=subtle border-radius=m}

Work day: Saturday, November 7, from nine to noon. Bring gloves; we bring
the tea.

::: /box
```

They are paragraphs of their own because of the blank lines around them. Without those, a plain renderer would run the box's lines and the work day together into one paragraph. So leave a blank line before and after every line that starts with colons; this book always does.

The heading reads `{{md.doc.title}}`, which tells a reader where the title comes from. Nothing is lost, and nothing turns into noise. This matters as you write: a document should still make sense as text. The chapters ahead point out how each part of CompMD keeps it so.

> [!TIP] In the playground, the **Plain Markdown** tab beside the page shows what a plain renderer shows.

When something is wrong

Mistakes don't stop the page. Delete the `::: /box` line, and the page still renders, with the box running to the end of the document. Under the text, the playground lists a *diagnostic*: what is wrong, where, and often a button that fixes it. Here it says that the box was never closed.

Each diagnostic has a code, such as `F004`, so you can look it up. Codes starting with `F` are failures, which change what renders; codes starting with `W` are warnings, which don't.

A diagnostic also says where it is, as a line and a column, and what is wrong, in a sentence. The one for the missing closer reads:

```
F004 12:1  `::: box` isn't closed
```

Line 12, column 1 is where the box opens, counting the frontmatter's lines too; the playground marks the place in the text, so you needn't count. A mistake gives one diagnostic, not a cascade of them: the box that runs on doesn't add a diagnostic for everything after it.

Whatever goes wrong, CompMD shows your text rather than hide it: a component it can't show is replaced by its fallback (chapter 3), never by a blank. Chapter 20 lists every code with its fix.

Next: the Markdown you know, and what CompMD adds to it.

2. The Markdown you know

Everything you write in Markdown works in CompMD, as GitHub shows it. This chapter is a tour of it: what you already know, what GitHub added, and the few things CompMD adds before any component. The newsletter grows a notice, a work day's list, a table of plots and a formula.

The newsletter, in Markdown

```
---
md:
  version: 1
  syntax: [wiki-links]
title: Elm Street Garden
issue: November 2026
---

# {{md.doc.title}}

News from the plots, {{md.doc.issue}}.

> [!IMPORTANT]
> The water goes off on November 15 for the winter. Empty your cans and
hoses before then.

## Work day

Saturday, November 7, from nine to noon.[^tea] On the list:

- [x] Order wood chips
- [ ] Turn the compost
- [ ] Mend the gate by plot 4
- [ ] Plant the garlic

[^tea]: Tea and soup from eleven, by the shed.

## Plots for next year

| Size | Plots | Fee for the year |
```

```
| --- | ---: | ---: |
| Small, 2 × 3 m | 12 | $25 |
| Large, 3 × 4 m | 8 | $40 |

A large plot is $3 \times 4 = 12$ square meters, twice a small one. Compost
heats well with about thirty parts brown to one part green:

$$
\frac{\text{brown}}{\text{green}} \approx 30
$$

More in [[Compost]], and the sign-up rules in [[Plots|the plot rules]].

<details>
<summary>Who to ask</summary>

Maya keeps the keys, and Tomás the tools.

</details>
```

Try it in the playground.

Markdown

Headings, paragraphs, *emphasis*, **strong text**, `code`, links, images, quotes, lists, code blocks and lines between sections all work as in CommonMark, the standard most Markdown tools follow. If you know Markdown, you know this part of CompMD.

What GitHub added

GitHub's Markdown adds a few things, and CompMD has them all:

- **Tables**, with a line of dashes under the header. A colon on the right of the dashes aligns the column to the right, as the numbers above; on the left, to the left; on both sides, in the center.
- **Task lists**: `- [ ]` for a box to tick, `- [x]` for one ticked.
- **Strikethrough**: `~~like this~~`.
- **Links without brackets**: a web address such as `https://example.com` becomes a link by itself.
- **Footnotes**: `[^tea]` where the note is referred to, and `[^tea]: …` anywhere for its text. Notes gather at the end of the page, numbered.

- **Alerts**: a quote whose first line is `[!NOTE]`, `[!TIP]`, `[!IMPORTANT]`, `[!WARNING]` or `[!CAUTION]` becomes a notice of that kind, with its own color and icon.

Tables, closer

A table needs its header row and the line of dashes under it; the rows after them are optional. Each row is one line, with its cells between `|` signs. A row with fewer cells than the header gets empty ones, and cells past the header's count are dropped.

A cell holds one line of inline Markdown: emphasis, links, code, images and values (`{{…}}`, chapter 4), but no list and no second paragraph. A `|` that is part of a cell's text is written `\|`, even inside code or a wiki link, because the table splits its rows into cells first:

```
| Tool | Shortcut |
| --- | --- |
| Pipe | `a \| b` |
```

Footnotes, closer

A footnote's label is yours, a word or a number, and only links the reference to its note: the page numbers the notes in the order they are referred to. A note's text can go anywhere in the document, and a reference with no note is shown as written.

Code blocks

A code block starts and ends with a line of three backticks, and the word after the first three names its language, so the page can color it: ```` ```yaml ````. What's inside always shows as written, values and colons included. Some languages draw something instead, such as a diagram, when the document uses a pack that draws them (chapter 17); without one, they stay code.

Math

Math is written in TeX, the language of scientific papers. Inline, it goes between single dollar signs: `$3 \times 4 = 12$`. On a line of its own, it goes between two lines of `$$`, or in a code block whose language is `math`, as GitHub also reads it:

````
```math
\text{area} = \text{width} \times \text{length}
```
````

The page typesets it; a plain renderer shows the TeX.

A dollar sign that is money is just text. `$25` is not math, because math needs a dollar sign at both ends, and the rule for the closing one keeps prices out: it can't be followed by a digit or come after a space. If a sentence still reads as math by mistake, write the dollar sign as `\$`.

Wiki links

`[[Compost]]` links to a page by its name, as wikis and many notes apps write it, and `[[Plots|the plot rules]]` shows *the plot rules* for the page *Plots*. They are an option, because in plain Markdown the same brackets mean something else, so a document turns them on in its frontmatter:

```
md:
  syntax: [wiki-links]
```

Without that line, `[[Compost]]` stays as written, brackets and all. A name under `syntax` that CompMD doesn't know is a warning; `wiki-links` is the only one in version 1.

Where a wiki link leads is up to the app that shows the document. Where there are no pages to link to, as in the playground, the link's text shows as text.

HTML

Markdown lets you write HTML, and CompMD shows a short list of safe tags, as GitHub does, because a document may come from anyone:

Tag	For	Keeps
`<details>`, `<summary>`	a part the reader opens, as above	`open`, to show it opened
`<sub>`, `<sup>`	H₂O, 10³	
`<kbd>`	a key to press: <kbd>Ctrl</kbd>	
` `	a line break inside a paragraph	
`<img>`	an image	`src` and `alt`
`<a>`	a link	`href`

Any other attribute, such as `style` or `class`, is dropped, and so is an address that isn't a web address, an email address (`mailto:`) or a relative one. Any other tag shows as text, as written, and so does a tag that is opened and never closed. A comment, `<!-- like this -->`, doesn't show at all, on the page or on GitHub, so it can hold a note to yourself.

CompMD reads nothing inside HTML. A line that starts with a tag begins an *HTML block*, which runs to the next blank line: `{{md.doc.title}}` in it shows as written, and so does a line of colons, which is a warning (`W116`) since it looks like a component and isn't one. So leave a blank line after `<details>` and before `</details>`, as above: the lines between are Markdown again. A tag inside a paragraph, such as `<kbd>`, ends where it closes, and the text around it is Markdown as usual.

Prefer Markdown and components where they do the job: they read better as text, and they look like the rest of the page.

Links to a passage

A link can point at a passage of the page by its words: `[the water notice](#:~:text=The%20water%20goes%20off)` scrolls to *The water goes off* and marks it. The address is the web's own form for this; each space is written `%20`. In the playground, such a link selects the words in the text instead, so a document can point at what to change.

After `#:~:text=` come the words, in up to four parts separated by commas:

Written	Marks
`#:~:text=Empty%20your%20cans`	the first place these words appear
`#:~:text=The%20water,the%20winter`	from *The water* to the next *the winter*: a longer passage, without writing it all
`#:~:text=Work-,day`	*day* where it comes right after *Work*, when the same words appear more than once
`#:~:text=nine,-to%20noon`	*nine* where *to noon* comes right after it

Since a comma and a dash have a meaning here, one that is part of the words is written `%2C` or `%2D`, and an `&` is `%26`. A link to a passage is an ordinary link, so a plain renderer shows it as one, and a browser that doesn't know the form opens the page at its top.

Writing colons and braces as text

CompMD gives a meaning to lines that start with colons, to a colon right before a word and brackets (`:value[…]`), and to double braces (`{{…}}`). When you mean them as text, put a

backslash before them: `\::: not a box`, `\{{not a value}}`. The backslash doesn't show. Inside code, nothing needs it: `` `{{title}}` `` shows as written.

A time such as 10:30 or a word such as `std::vec` needs nothing either: a colon only starts a component after a space, at the start of a line, or after punctuation.

Plain Markdown is CompMD too

A document without a single component is still a CompMD document, and gets what CompMD gives every document: the page as you type, diagnostics, and values from its frontmatter. A file ending in `.md` without an `md` key in its frontmatter stays ordinary Markdown, so CompMD never changes a file you didn't mean for it.

Next: containers and params, the components that make a page out of a text.

3. Containers and params

Components turn a text into a page: a notice that stands out, a box inside a box, a passage in another language. This chapter is about how you write any component, and the params that set how it looks. The newsletter gets a notice, a work day with its list, and a greeting from the neighbors.

The newsletter, with boxes

```
---
md:
  version: 1
title: Elm Street Garden
issue: November 2026
---

# {{md.doc.title}}

News from the plots, {{md.doc.issue}}.

::: box {padding=m border-radius=m background=accent}

**The water goes off on November 15** for the winter. Empty your cans and
hoses before then.

::: /box

## Work day

::: box {padding=m border=subtle border-radius=m}

Saturday, November 7, from nine to noon. Bring gloves; we bring the tea.

::: box {padding=s border-radius=s background=subtle}

**On the list:** turn the compost, mend the gate by plot 4, plant the
garlic.

::: /box
```

```
::: /box

## From our neighbors

::: box {#greeting lang=es padding="s m" border-radius=m background=subtle}
::: + max-width=32rem

*¡Hola, vecinos!* La huerta de la calle Olmo os invita a la jornada del sábado.

::: /box
```

Try it in the playground.

Three ways to write a component

A component is written in one of three ways, and the number of colons tells which:

Form	Written	For
Container	`::: box {…}`, a body, then `::: /box`	a component around content you write
Leaf	`:: chart[Weekly sign-ups] {…}`, one line	a component that makes its content itself
Inline	`:value[12 plots]{…}`, inside a sentence	a component in the middle of text

Each has a name, and params in braces, which may be left out. A leaf and an inline component also have a *fallback* in brackets, which the next sections explain.

The components you can use come from three places: those built into CompMD, such as `box`; those from *packs* other people publish (chapter 16); and your own, written in CompMD itself (Part IV). Built in are `box` and the layouts `stack`, `row` and `grid` (Part III), `if` and `each` (chapter 5), `value` (chapter 4), and the inputs a reader changes (chapter 7).

Containers

A container opens with three colons and its name, and closes with three colons, a slash and the same name. Its body is any Markdown: paragraphs, lists, tables, and other containers.

Containers nest, as the work day's list nests in the work day. Because every closer names its container, it's always clear which one a line ends, even with boxes in boxes. A closer with a misspelt name, such as `::: /bxo`, is a diagnostic (`F003`) that names the container still open; a container never closed is `F004`, and it runs to the end of what holds it.

A container can sit in a list item or a quote, indented like the item's text:

```
- Sign up by November 30.

  ::: box {padding=s border=subtle}

  Plots go to last year's gardeners first.

  ::: /box
```

In a quote, each line of the container starts with `>`, as the quote's other lines do:

```
> ::: box {padding=s border=subtle}
>
> The early bird gets the sunniest plot.
>
> ::: /box
```

A container opens and closes in the same list item or quote. A closer in another one (`F032`) is ignored, and the container closes where its own list item or quote ends.

A container may be empty, its closer right under its opener, and may hold another of the same name.

Fallbacks

A leaf or an inline component has no body you write: it makes its content itself, such as a chart or a computed number. What's in its brackets is its *fallback*: the text shown where the component can't be, by a plain renderer, or when something goes wrong. Write in it what the component usually shows, so the text reads well on its own: `:: chart[Sign-ups rose from 12 to 31 this year] {…}`.

A container's fallback is its body, as Markdown, so a plain renderer shows everything you wrote in it.

Params

Params go in braces after the name, separated by spaces. They read like a list of settings:

Written	Is
`padding=m`	a word or a number, as written
`padding="s m"`	text in quotes, which may hold spaces
`scroll`	a setting turned on
`background=(…)`	an expression, computed (chapter 4)
`tags=["kale","leeks"]`	a list, written as in JSON, for a component that takes one
`#greeting`	the component's id, for a link to it: `[the greeting](#greeting)`
`.notice`	a class, which a stylesheet of the app showing the document can style
`md:padding=l`	from a width up (chapter 10)

A few rules about how values are written:

- A value without quotes ends at the next space or `}`, and can't hold parentheses or braces: `title={{md.doc.title}}` is a mistake (`F011`), and `title="{{md.doc.title}}"` is what you mean. A value with nothing after the `=` is one too; `title=""` is empty text.
- In quotes, a backslash writes a quote or a backslash: `"Say \"hi\""`.
- Quotes always make text. Without them, the component decides: `count=3` is the number 3 to a component that takes a number, and `count="3"` is the text *3*.
- Each param is given once, continuation lines included (below); the second time is a diagnostic (`F017`), and the first one counts.

Which params a component takes, and what values each takes, is up to the component. A param it doesn't know (`F008`), or a value it doesn't take (`F009`), is a diagnostic that says what it does take, and often offers to fix it: write `border=subtel`, and the playground offers `subtle`.

`lang` and `dir` are taken by every component, for a passage in another language: `lang=es` above tells the reader's browser, a screen reader and a spell checker that the greeting is Spanish. `dir=rtl` writes a passage right to left, as Arabic or Hebrew is, and `dir=auto` lets the text decide, for text whose language you don't know.

The common params

Every block, whatever its component, has a box around it, and every component takes the same params for that box. These set its look and its size:

Param	Values
`padding`	`none`, `s`, `m`, `l`, or a length such as `1rem`; two of them for top and bottom, then the sides: `padding="s m"`
`padding-block`, `padding-inline`	the same, for top and bottom only, or for the sides only; two of them for one end, then the other
`padding-block-start`, `padding-block-end`, `padding-inline-start`, `padding-inline-end`	one of them, for one side: the top, the bottom, the side the text starts from, the side it ends at
`border`	`none`, `subtle`, `strong`, or a CSS border such as `"1px dashed"`
`border-radius`	`none`, `s`, `m`, `l`, `full` for a pill, or a length
`background`	`none`, `subtle`, `strong`, `accent`, or a color such as `#e9f3ff`
`width`, `max-width`, `min-width`, `height`	a length, a percentage, or `auto`: `max-width=32rem`
`aspect-ratio`	width over height, such as `16/9`, or `none`

A length is a number with `px`, `rem`, `em` or `%`; `1rem` is the page's text size. A width or height counts the padding in it.

The words (`s`, `subtle`, `accent`) follow the look of the app that shows the document, light or dark, so a document written with them fits wherever it is shown. Unless the app changes them, `s`, `m` and `l` are half the text's size, the text's size and twice it (0.5, 1 and 2 rem) for padding, and 4, 8 and 16 pixels for corners; `subtle` and `strong` are faint and stronger tints of the text's own color. A color or a length is the same everywhere: use them when that's what you mean.

Sides are named `start` and `end`, not left and right, so a passage written right to left (`dir=rtl`) gets its padding on the right sides too.

The rest of the common params place a block among others: `grid-area`, `order`, `align-self`, `justify-self`, `show`, `hide`, `inset-…`, `scroll`, `fill` and `sticky`. They are Part III's, in

chapters 9, 10 and 11. These are all the styling a document can write: there is no param for raw CSS, so a name outside the list is `F008`, and a value outside a param's values is `F009`.

Ids and classes

An id names one block, so a link can lead to it. Write it with `#` in the braces, `{#greeting}`, or as `id=greeting`, but not both (`F017`). `[the greeting](#greeting)` then scrolls to the block.

A class is a name for a kind of block, which a stylesheet of the app showing the document can style: `{.notice}`. A class does nothing by itself; the app has to know it. A block can have several, `{.notice .wide}`, and `class=(…)` adds one that is computed: `{.plot class=("crop-" + p.crop)}` gives the classes *plot* and *crop-garlic* (chapter 5 has `p`). Classes add up in the order written.

Ids and classes are names: a letter first, then letters, digits, `-` or `_`.

Long param lists

When the params don't fit on a line, continue them on the next line, which starts with `::: +`, directly under the opener:

```
::: box {#greeting lang=es padding="s m"}
::: + border-radius=m background=subtle max-width=32rem

¡Hola, vecinos!

::: /box
```

A continuation line goes directly under the opener, a leaf or another continuation line, with no blank line between them. Anywhere else, it is a diagnostic (`F012`), and is ignored.

When a component line is wrong

Each mistake in a component line is one diagnostic, and the page keeps as much as it can:

You wrote	Code	The page
`:::box`, without the space	`F016`	shows the line as text
`::: /bxo`, a misspelt closer	`F003`	closes the innermost open container
a container with no closer	`F004`	runs it to the end of what holds it

You wrote	Code	The page
`::: /box` with no box open	`F005`	ignores the line
`padding=xl`, a value the param doesn't take	`F009`	leaves the param out
`title=Hi`, a param the component doesn't take	`F008`	leaves the param out
`padding=m padding=s`	`F017`	uses the first
`title={{md.doc.title}}`, braces without quotes	`F011`	keeps the params before the mistake
`:: #else` in a container without that slot	`F006`	adds what follows to the body
a container line right under a paragraph's last line	`W108`	shows it as intended; a plain renderer may not, so add a blank line

Slots

Some containers have parts with names, called *slots*, each started by a line of two colons and a `#`: `:: #else`. You'll use them with `if` and `each` in chapter 5, and give your own components slots in Part IV.

Next: values and expressions, for text that comes from the frontmatter or is computed.

4. Values and expressions

A document can say a thing once and use it everywhere: the issue's date in the heading and the notice, a fee in the table and in the text. This chapter shows how to insert values from the frontmatter, how to compute and format them, and how a param can follow a value. The newsletter's dates, fees and names now come from its frontmatter.

The newsletter, with values

```
---
md:
  version: 1
title: Elm Street Garden
issue: November 2026
editor: {name: Maya Okafor, phone: 555-0142}
workday: "2026-11-07"
water_off: "2026-11-15"
fees: {small: 25, large: 40}
plots: {small: 12, large: 8, taken: 17}
helpers: [Maya, Tomás, Priya, Jun]
---

# {{md.doc.title}}

News from the plots, {{md.doc.issue}}, from {{md.doc.editor}}.

::: box {padding=m border-radius=m background=(md.daysBetween(md.today(),
md.doc.water_off) <= 14 ? "accent" : "subtle")}

**The water goes off on :value[November 15, 2026]{of=
(md.date(md.doc.water_off, "long"))}** for the winter. Empty your cans and
hoses before then.

::: /box

## Work day

:value[Saturday, November 7, 2026]{of=(md.date(md.doc.workday, "full"))},
from nine to noon, with :value[Maya, Tomás, Priya, and Jun]{of=
```

```
(md.list(md.doc.helpers))} leading the jobs.

## Plots for next year

| Size | Plots | Fee for the year |
| --- | ---: | ---: |
| Small | {{md.doc.plots.small}} | :value[$25.00]{of=
(md.currency(md.doc.fees.small, "USD"))} |
| Large | {{md.doc.plots.large}} | :value[$40.00]{of=
(md.currency(md.doc.fees.large, "USD"))} |

:value[3 plots are]{of=(string(md.doc.plots.small + md.doc.plots.large -
md.doc.plots.taken) + " " + md.plural(md.doc.plots.small +
md.doc.plots.large - md.doc.plots.taken, "plot is", "plots are"))} still
free. Call {{md.doc.editor.name}} at {{md.doc.editor.phone}} to take one.
```

Try it in the playground.

The frontmatter's values

The frontmatter is YAML, and a value can be more than a line of text:

Written	Is
`title: Elm Street Garden`	text
`fees: {small: 25, large: 40}`	a record: values under names, written in braces or one per line, indented
`helpers: [Maya, Tomás, Priya, Jun]`	a list, in brackets or one `- item` per line
`small: 25`	a number
`workday: "2026-11-07"`	a date, written as year, month and day
`open: true`	`true` or `false`

Quote a value that would otherwise read as something else: a time such as `"9:30"`, or a version such as `"1.10"`, which unquoted is the number 1.1.

Inserting a value

`{{…}}` inserts a value where it stands. Inside it goes a *path*: where the value is, written as names joined by dots.

- `md.doc` is your frontmatter, so `{{md.doc.issue}}` is the `issue` key: *November 2026*.
- A dot goes into a record: `{{md.doc.plots.small}}` is *12*, and `{{md.doc.editor.phone}}` is *555-0142*.
- A record with a `name` reads as its name, since it stands for someone or something: `{{md.doc.editor}}` is *Maya Okafor*.
- `md.lang` is the document's language, `en` unless the frontmatter says otherwise (below).
- A number is written as it is (*25*, *12.5*), and a date as you wrote it (*2026-11-07*). To write them for readers, *$25.00* or *November 7*, use `value`, below.

`true` and `false` are written as they are, and an empty value (`note:` with nothing after it) as nothing. A list isn't inserted as it is, since there are many ways to write one; `{{md.doc.helpers}}` is a diagnostic (`F024`), and so is a record without a `name`. Write it with `md.list`, below, or one item at a time with `each` (chapter 5).

`{{…}}` works in text, in headings, in table cells, in a link's address (`[the rules](https://example.org/{{md.doc.rules}})`), and in params written in quotes (`title="Plot {{md.doc.plot}}"`). It doesn't work in code, which always shows what you wrote.

A path that doesn't exist, such as `{{md.doc.isue}}`, is a diagnostic (`F014`) that suggests the nearest one, here `md.doc.issue`. To write double braces as text, start them with a backslash: `\{{like this}}`.

Computed text: `value`

A path only finds a value; it can't do anything to it. For text that is computed or formatted, use the inline component `value`. Its `of` param is the text to show, and its brackets hold the fallback, the text a plain renderer shows instead:

```
---
fees: {small: 25, large: 40}
---

A small plot is :value[$25.00]{of=(md.currency(md.doc.fees.small, "USD"))}
a year.
```

The fallback should be what the value usually is, so the text still reads well on GitHub. It doesn't change when the frontmatter does, so it's a sample, not a promise.

Expressions

The `(…)` in `of=(…)` is an *expression*: a small formula, computed when the page is made. Any param can be one, not only `of`. In the notice above, `background` is computed: the accent color when the water goes off within two weeks, the subtle one before.

Expressions are written in CEL, a small language for formulas. You'll need only a little of it:

Written	Is
`md.doc.fees.small * 4`	arithmetic: `+`, `-`, `*`, `/`, and `%` for the remainder
`md.doc.plots.taken < 20`	a comparison: `==`, `!=`, `<`, `<=`, `>`, `>=`
`a && b`, `a || b`, `!a`	and, or, not
`soon ? "accent" : "subtle"`	one value or another: if `soon` is true, `"accent"`, otherwise `"subtle"`
`"Plot " + md.doc.plot`	text joined to text
`string(md.doc.fees.small)`	a number as text, *25*
`size(md.doc.helpers)`	how many items a list has
`md.doc.helpers[0]`	an item of a list by its place, counting from 0: *Maya*
`"Jun" in md.doc.helpers`	true when the list holds the value
`["Maya", "Jun"]`	a list, written in the expression
`has(md.doc.subtitle)`	true when the frontmatter has the key

Text in an expression is in double quotes, or single ones: `'subtle'`. Paths work as in `{{…}}`: `md.doc.fees.small` is the small plots' fee. Parentheses group, as in arithmetic: `(md.doc.plots.small + md.doc.plots.large) * 2`. An expression is written on one line.

`value` shows text, so a number goes through `string(…)` or one of the formatting functions below first: `of=(string(size(md.doc.helpers)))`, not `of=(size(md.doc.helpers))`.

Text has a few functions of its own, written after it with a dot:

Written	Gives
`md.doc.title.contains("Garden")`	true when the text holds the other; also `startsWith` and `endsWith`
`md.doc.title.upperAscii()`	*ELM STREET GARDEN*; `lowerAscii()` for small letters (A to Z only)

Written	Gives
`md.doc.title.replace("Street", "St.")`	*Elm St. Garden*
`" Elm ".trim()`	the text without spaces at its ends: *Elm*
`"a,b".split(",")`	a list of the parts: `["a", "b"]`
`md.doc.helpers.join(", ")`	the items of a list of text, joined: *Maya, Tomás, Priya, Jun*

Keys that may be missing

A path to a key the frontmatter doesn't have is a mistake, in `{{…}}` and in expressions alike. When a key is there in some issues and not in others, ask first with `has`, and read it only when it's there:

```
---
title: Elm Street Garden
---

# :value[Elm Street Garden]{of=(has(md.doc.subtitle) ? md.doc.title + ": "
+ md.doc.subtitle : md.doc.title)}
```

Numbers, text and true or false

An expression knows what kind of value each part is, and checks that they fit before the page is made:

- **Text and numbers don't mix.** `"Plot " + 4` is a mistake: write `"Plot " + string(4)`. Comparing them is one too.
- **Whole numbers.** A number written in an expression without a decimal point, `4`, and what `size(…)` and `md.loop` give, are whole numbers, and dividing two of them drops the remainder: `size(md.doc.helpers) / 3` is *1* with four helpers. Write `double(size(md.doc.helpers)) / 3` for *1.333…*. A number from the frontmatter is never whole in this sense, so `md.doc.fees.small / 2` is *12.5*. A list's place must be a whole number, and `int(…)` makes one: `md.doc.helpers[int(md.doc.plots.taken) % 4]`.
- **Dates are text**, written year, month and day, so they compare as dates do: `md.doc.workday < md.doc.water_off` is true.
- **`true` and `false`** are what comparisons give, and what `when` takes in chapter 5. `if` and `?` take nothing else: a number or a text is a mistake there, never a stand-in for true.

When an expression is wrong

A mistake in an expression is a diagnostic (`F024`) that points to the place in it, and says what is wrong:

Written	The diagnostic says
`md.doc.fess.small`	unknown field 'fess'; did you mean 'fees'?
`"Fee " + md.doc.fees.small`	'+' can't be applied to string and double
`of=(size(md.doc.helpers))`	`of` takes string, but the expression gives int

string, *double* and *int* are the expression's words for text, a number and a whole number. Where the mistake is, the component shows its fallback, or the param is left out as if you hadn't written it, so the rest of the page is still there.

`{{…}}` takes only a path, never a formula: `{{ md.doc.fees.small * 4 }}` shows as written, with a warning (`W106`) that suggests `value`.

Formatting for readers

CompMD's own functions start with `md.`. These format values for readers:

Function	Example	Gives
`md.date(date, style)`	`md.date(md.doc.workday, "long")`	*November 7, 2026*; the other styles give *11/7/26* (`"short"`), *Nov 7, 2026* (`"medium"`) and *Saturday, November 7, 2026* (`"full"`)
`md.currency(n, code)`	`md.currency(md.doc.fees.large, "USD")`	*$40.00*; the code is the currency's three letters, such as `"EUR"` or `"GBP"`
`md.number(n, digits)`	`md.number(1636.5, 2)`	*1,636.50*: rounded to that many digits after the point, with the language's separators
`md.plural(n, one, other)`	`md.plural(3, "plot", "plots")`	*plots*: the first word for one, the second for any other number
`md.list(list)`	`md.list(md.doc.helpers)`	*Maya, Tomás, Priya, and Jun*; `md.list(md.doc.helpers, "or")` gives *Maya, Tomás, Priya, or Jun*

And these work with dates:

Function	Gives
`md.today()`	today's date, when the page is made, in the document's time zone, `tz` under `md` (chapter 8), or else the app's
`md.daysBetween(a, b)`	the days from `a` to `b`: `md.daysBetween(md.today(), md.doc.water_off)`
`md.addDays(date, n)`	the date `n` days later, or earlier for a negative `n`

Chapter 8 has the rest, for dates, numbers and lists.

The document's language decides how they write: with `lang: de` under `md` in the frontmatter, `md.date(md.doc.workday, "long")` gives *7. November 2026*, and `md.currency(25, "EUR")` gives *25,00 €*.

Next: showing and repeating, for parts that show only sometimes, and lists made from the frontmatter.

5. Showing and repeating

Some parts of a page depend on what the frontmatter holds: a line that shows only while plots are free, a list with one item per job, a box per gardener. Two built-in containers do this, `if` and `each`. With them, the newsletter's lists come from its frontmatter, and next month's issue is mostly a change of values.

The newsletter, from its lists

```
---
md:
  version: 1
title: Elm Street Garden
issue: November 2026
jobs: [Turn the compost, Mend the gate by plot 4, Plant the garlic]
helpers: [Maya, Tomás, Jun]
plots:
  - {number: 1, gardener: Maya Okafor, crop: garlic}
  - {number: 2, gardener: Tomás Reyes, crop: kale}
  - {number: 3, gardener: "", crop: ""}
  - {number: 4, gardener: Priya Nair, crop: leeks}
  - {number: 5, gardener: "", crop: ""}
news: []
---

# {{md.doc.title}}

News from the plots, {{md.doc.issue}}.

## Work day

Saturday, November 7, from nine to noon. On the list:

::: each {in=(md.doc.jobs) as=job join=list}

1. {{job}}

::: /each
```

```
Thanks to last month's helpers:

::: each {in=(md.doc.helpers) as=helper join=inline sep=", " last=" and "}

**{{helper}}**

::: /each

## The plots

::: each {in=(md.doc.plots.filter(p, p.gardener != "")) as=p}

::: box {padding=s border=subtle border-radius=s}

**Plot {{p.number}}**: {{p.gardener}}, growing {{p.crop}}.

::: /box

::: /each

::: if {when=(md.doc.plots.exists(p, p.gardener == ""))}

:value[2 plots are free]{of=(string(size(md.doc.plots.filter(p, p.gardener
== ""))) + " " + md.plural(size(md.doc.plots.filter(p, p.gardener == "")),
"plot is", "plots are") + " free")} for next year. Ask Maya to take one.

:: #else

Every plot is taken for next year. Ask Maya to put you on the waiting list.

::: /if

::: each {in=(md.doc.news) as=item join=list}

- {{item}}

:: #before

## From the committee

:: #empty

No news from the committee this month.
```

```
::: /each
```

Try it in the playground.

The page shows the three jobs as one numbered list, a line thanking *Maya, Tomás and Jun*, a box for each of the three plots with a gardener, the line *2 plots are free for next year*, and *No news from the committee this month.*

A list of records

The `plots` key is a list of records, one per plot, each with the same names: `number`, `gardener` and `crop`. In YAML, each item starts with `-`, and a record can be written in braces on one line. This is how a document holds a table's worth of values; Part II reads the same from a spreadsheet file.

`if`

`if` shows its body when its `when` param is true:

```
---
water_on: true
---

::: if {when=(md.doc.water_on)}

The taps work until November 15.

:: #else

The water is off for the winter.

::: /if
```

`when` is an expression that is true or false. What's after the `:: #else` line shows when it is false; leave that part out to show nothing. A `when` that is neither, such as a number, or that fails, is a diagnostic (`F024`), and the page then shows both parts, so a mistake never hides your text. The `:: #else` line starts a *slot*, a named part of the container (chapter 3).

In the newsletter, `when` asks whether any plot has no gardener: `md.doc.plots.exists(p, p.gardener == "")`. `exists` goes through a list, calling each item `p`, and is true when the condition after the comma is true for at least one of them.

When only one value differs, a param computed with `? :` is shorter than two branches: `background=(md.doc.water_on ? "subtle" : "accent")`, as in chapter 4.

each

`each` shows its body once for each item of a list. `in` is the list, and `as` gives each item a name to use in the body:

```
---
jobs: [Turn the compost, Mend the gate, Plant the garlic]
---

::: each {in=(md.doc.jobs) as=job}

**Job:** {{job}}

::: /each
```

This shows three paragraphs, each starting **Job:**. With a list of records, a dot reads a value of the record: `{{p.gardener}}`.

The `as` name is yours to choose. Make it one word, or words joined by `_`, such as `job` or `plot_number`: a `-` isn't allowed (`F009`). A few words are taken by the expression language, and can't be names either (`F013`): among them `in`, `if`, `else`, `true`, `false`, `null`, `list`, `map`, `string` and `int`.

`in` takes a list, and nothing else: a number or a text is a mistake (`F024`). `in` and `as` are both needed; without `as`, the body can't name its item (`F010`).

One list, or one paragraph

Each time round, the body makes its own blocks. To make them one list instead of three lists of one item, write the body as one list item and add `join=list`. A numbered list then counts on, 1, 2, 3, from one item to the next.

To make them one paragraph, write the body as a paragraph and add `join=inline`. `sep` goes between the items, a space unless you say otherwise, and `last` between the last two, so `sep=", " last=" and "` gives *Maya, Tomás and Jun*.

Without `join`, every time round adds its blocks to the page, one after another, as the boxes of the plots do. An `each` around a box gives a box per item; Part III puts such boxes side by side, as cards.

`join`	The body must be	Gives
`none`, the default	anything	the blocks of every time round, one after another
`list`	list items, all of one kind: `-` or `1.`	one list, with every item; a body may have more than one item
`inline`	exactly one paragraph	one paragraph, with `sep` between the items and `last` before the last one

A body of the wrong shape is a diagnostic (`F025`), and the items are added one after another, as without `join`. A body that is a single list item without `join` is a warning (`W103`), since it gives a list of one item per time round, which is rarely what you mean. `last` is `sep` unless you give it.

Before and empty

Two slots handle the edges. What's under `:: #before` shows once, above the items, but only when there is at least one; it's the place for a list's heading. What's under `:: #empty` shows when there are none. In the newsletter, `news` is an empty list, `[]`, so the heading *From the committee* doesn't show, and the empty message does.

Slot lines start at the container's own level, not indented under the list item, which would put them inside it.

Each container has its own slots: `if` has `else`, and `each` has `before` and `empty`. A slot the container doesn't have, such as `:: #else` in an `each`, is a diagnostic (`F006`), and what follows it joins the body. Each slot is given once.

Which time round

Inside the body, `md.loop` says where the list is:

```
---
jobs: [Turn the compost, Mend the gate, Plant the garlic]
---

::: each {in=(md.doc.jobs) as=job}

**Job {{md.loop.number}} of {{md.loop.count}}:** {{job}}

::: /each
```

Path	Is
`md.loop.number`	1 for the first item, 2 for the next, and so on
`md.loop.count`	how many items there are
`md.loop.first`, `md.loop.last`	true for the first item, or for the last
`md.loop.index`	0 for the first item, as list positions in expressions count

`md.loop` and the `as` name are there only in the body, not in `before` or `empty`.

An `each` in an `each`

An `each` can go inside another, for a list in each record. The inner body sees both names, its own item and the outer one, and `md.loop` is the inner list's:

```
---
plots:
  - {number: 1, beds: [garlic, onions]}
  - {number: 2, beds: [kale]}
---

::: each {in=(md.doc.plots) as=p}

**Plot {{p.number}}:**

::: each {in=(p.beds) as=bed join=inline sep=", "}

bed {{md.loop.number}}, {{bed}}

::: /each

::: /each
```

This shows *Plot 1:* above *bed 1, garlic, bed 2, onions*, then *Plot 2:* above *bed 1, kale*. Give the two lists different `as` names, so each stays reachable.

Choosing items

`in` is an expression, so it can choose and order items before `each` sees them. The newsletter shows only plots with a gardener: `md.doc.plots.filter(p, p.gardener != "")`. `filter` keeps the items for which the condition is true.

A few more, all written after a list:

Written	Gives
`md.doc.plots.filter(p, p.crop == "garlic")`	the items for which the condition is true
`md.doc.plots.exists(p, p.gardener == "")`	true when the condition is true for at least one item
`md.doc.plots.all(p, p.gardener != "")`	true when it is true for every item
`md.doc.plots.map(p, p.gardener)`	a list of one value from each item: the gardeners

And a few functions that take a list, for sorting and counting: `md.sortBy(md.doc.plots, "crop")` sorts by a name of the records, `md.reverse(…)` turns a list around, `md.slice(list, 0, 3)` takes the first three, and `size(…)` counts. Chapter 8 lists them all.

The params, at a glance

Container	Param	Takes
`if`	`when`	required: true or false
`each`	`in`	required: a list
	`as`	required: the name of each item
	`join`	`none` (the default), `list` or `inline`
	`sep`	the text between items with `join=inline`; a space by default
	`last`	the text before the last item with `join=inline`; `sep` by default

And their slots: `else` for `if`; `before` and `empty` for `each`.

Where CompMD isn't known

A plain renderer shows each line of an `if` and an `each` as written: both parts of an `if`, with its lines between them, and an `each`'s body once, `{{job}}` and all. That's the one place where CompMD can't keep a text readable, since the items are only known when the page is made. Where the text must read well on GitHub too, write such a list out by hand.

What `each` can't do

`each` makes whole blocks, so it can't make the rows of a table: a Markdown table is one block. A table from a list comes from a component that makes tables, such as one from a pack (chapter 16).

Next: Part II, data from files, where the lists come from spreadsheets and files instead of the frontmatter.

6. Data from files

The frontmatter holds a few values well, but a list of plots belongs in a spreadsheet, and the club's fees in one place every issue reads. This chapter loads data from files beside the document: a CSV file of the plots, a YAML file of the work days, and another CompMD document's frontmatter. It also says what happens when a file is missing or a row is wrong, and where else data can come from.

The newsletter, from its files

The document:

```
---
md:
  version: 1
  data:
    - from: ./plots.csv
      as: plots
      schema:
        number: {from: Plot, type: number}
        gardener: {from: Gardener, type: string?}
        crop: {from: Crop, type: string?}
        size: {from: "Size (m2)", type: number}
    - from: ./workdays.yaml
      as: workdays
      schema: {date: date, jobs: array}
    - from: ./club/about.comp.md
      as: club
title: Elm Street Garden
issue: November 2026
---

# {{md.doc.title}}

News from the plots, {{md.doc.issue}}.

## Work days

::: each {in=(md.data.workdays) as=day}
```

```
**:value[Saturday, November 7, 2026]{of=(md.date(day.date, "full"))}**,
from nine to noon: :value[turn the compost, mend the gate by plot 4, and
plant the garlic]{of=(md.list(day.jobs))}.

::: /each

## The plots

::: each {in=(md.data.plots.filter(p, p.gardener != null)) as=p join=list}

- **Plot {{p.number}}**, {{p.size}} m²: {{p.gardener}}, growing {{p.crop}}

::: /each

:value[2 plots are free]{of=(string(size(md.data.plots.filter(p, p.gardener
== null))) + " " + md.plural(size(md.data.plots.filter(p, p.gardener ==
null)), "plot is", "plots are") + " free")} for next year. A small plot is
:value[$25.00]{of=(md.currency(md.data.club.fees.small, "USD"))} a year, a
large one :value[$40.00]{of=(md.currency(md.data.club.fees.large, "USD"))}.
Call {{md.data.club.contact}} at {{md.data.club.contact.phone}} to take
one.
```

Try it in the playground.

The plots, as a spreadsheet saves them:

```
Plot,Gardener,Crop,Size (m2)
1,Maya Okafor,garlic,12
2,Tomás Reyes,kale,12
3,,,20
4,Priya Nair,leeks,20
5,Jun Park,beans,12
6,,,12
```

The work days:

```
- date: 2026-11-07
  jobs: [turn the compost, mend the gate by plot 4, plant the garlic]
- date: 2026-11-21
  jobs: [clear the bean poles, cover the empty beds]
```

And the club's page, a CompMD document of its own in the folder `club/`, whose frontmatter holds the fees and the contact:

```
---
md:
  version: 1
name: Elm Street Garden
contact: {name: Maya Okafor, phone: 555-0142}
fees: {small: 25, large: 40}
---

# {{md.doc.name}}

Six plots behind the library on Elm Street, kept by the neighbors who grow
in them. Call {{md.doc.contact}} at {{md.doc.contact.phone}}.
```

The page lists both work days with their jobs, the four plots with a gardener, and *2 plots are free for next year*. When a plot changes hands, you change a row of the spreadsheet; the newsletter follows.

Loading a file

A document lists the files it reads under `md.data`, one entry each:

- **`from`** is where the file is: `./plots.csv` is beside the document, `./club/about.comp.md` in the folder `club` beside it.
- **`as`** is the name the data gets: `as: plots` makes it `md.data.plots`.

From then on, `md.data.plots` works wherever `md.doc` does: in `{{…}}`, in expressions, in `each`. A list of records is a list of records, wherever it came from: chapter 5's `filter`, `exists` and `md.loop` work on it as they did on the frontmatter's.

The formats

The file's extension says how it is read:

File	Gives
`.csv`	a list of records, one per row, named by the header row
`.tsv`	the same, with tabs between the cells

File	Gives
`.yaml`, `.yml`	what the YAML holds, as in the frontmatter: a list, a record, a value
`.json`	what the JSON holds
`.comp.md`	the document's frontmatter keys, its `md.doc`; not its text, and not its own `md` settings

YAML in a data file follows the frontmatter's rules (chapter 4): `2026-11-07` is a date written as text, and `no` is text, not false.

A CompMD document is a good home for values several documents share, such as the club's fees: it reads as a page of its own, and every issue that loads it gets the same values. `{{md.data.club.contact}}` is *Maya Okafor*, since a record with a `name` reads as its name.

Spreadsheets: CSV

Save a sheet as CSV, and its first row becomes the names. Each row after it is a record, its cells under those names. Without anything else, every cell is text, and an empty cell is empty text, `""`. That is often enough to show a cell: `{{p.Gardener}}`.

But text doesn't count. `p.Plot + 1` is a diagnostic (`F024`), since `"1"` is text, not the number 1. A *schema* says what each column holds.

A schema

`schema` lists the columns, each with its type, as you'd declare a param (Part IV uses the same types):

Type	A cell	Read as
`string`	`Maya Okafor`	text
`number`	`12`, `4.5`	a number, with a point for the decimals
`boolean`	`true`, `false`	true or false
`date`	`2026-11-07`	a date, year, month and day

A YAML or JSON file's values already have their types, and `array` and `object` take its lists and records: `jobs: array` above. A schema there checks the records rather than reading them.

Write a schema in one line when the names fit: `schema: {date: date, jobs: array}`. Columns the schema doesn't name stay in each record, as text.

Empty cells

A cell may be empty, as plot 3's gardener is. Add `?` to its type to allow that: `type: string?`, and an empty cell is `null`, no value. That's how the newsletter tells free plots from taken ones: `p.gardener != null`.

Without the `?`, an empty cell is a mistake in the row, as below.

Names with spaces

A header such as `Size (m2)` can't be written in a path. Give the column a name of your own, with the header as its `from`:

```
schema:
  size: {from: "Size (m2)", type: number}
```

The data then has `size`, and `{{p.size}}` works. A name is one word, or words joined by `_`, such as `plot_size`.

Other separators

Some spreadsheets save CSV with a semicolon between the cells. Say so with `delimiter`:

```
md:
  data:
    - from: ./plots.csv
      as: plots
      delimiter: ";"
```

Such files often write decimals with a comma, `1,5`, which isn't a number in a schema: declare the column as `string`, or save it with points.

When something is wrong

Data comes from outside the document, so the document still shows when it is wrong:

What	The diagnostic	What the page gets
The file isn't there, or isn't valid CSV, YAML or JSON	`F026`, with where in the file when it can tell	`md.data.<name>` is `null`, and an `each` over it shows its `:: #empty` slot
A header with an empty or repeated name	`F026`	`null`
A row with more or fewer cells than the header	`F027`, at its line	the row is left out
A cell that doesn't fit its type, such as `x` for a `number`, or an empty cell without `?`	`F027`, naming the record and the column: *record 2: `Plot` isn't a number: `x`; it is dropped*	the row is left out

A row left out is left out of every list made from the data, so a total over a list with an `F027` is short until the row is fixed. The diagnostic says which row.

Where files may be

A path is relative to the document that writes it, as a link is: `./plots.csv` is beside it, `./club/about.comp.md` in a folder below. A path can't go up out of the document's project, the folder the app gives it: `../../other.csv` there is `F026`.

Values inside data are never paths: a `logo: ./logo.png` in the club's frontmatter, written into this document as `![]({{md.data.club.logo}})`, is a path from this document, not from the club's folder.

A pack (chapter 16) can bring data files too. A document that imports the pack reads one by the pack's name: `from: "@acme/garden-kit/crops.json"`.

Data from a website

`from` can be an `https://` address:

```
md:
  data:
    - from: https://weather.example.org/elm-street/frost.json
      as: frost
      fallback: ./frost.json
```

The document can't decide on its own to fetch from a site. The app that shows it has a list of sites its documents may read from, and for any other site it asks the reader, the first time a document needs it:

> This document wants to load data from weather.example.org. Allow?

The reader's answer is kept for every document, so they are asked once per site. Until they answer, and if they say no, the site is `F028`, and the document gets its `fallback`, a file of its own folder, or `null` without one. Once they allow it, the page shows the site's data. A fetch that fails also takes the fallback. An app with no reader to ask, such as one that prints documents on a server, reads only from its list.

Try it in the playground: a document that reads a pack's latest release from npm, a site the playground doesn't list. It shows its fallback, and the playground asks you.

So give remote data a `fallback`, kept up to date now and then, and the page reads well wherever it's opened, before the reader answers and after a no. The site you fetch from learns that the document was opened, and the values it asks with (chapter 7). A document carries no passwords: data that needs one comes through the app, below.

Data from the app

An app can give its documents its own data, such as a calendar app its events. Its documents read it with `from: host:`, and what follows the colon is the app's to define:

```
md:
  data:
    - from: host:events?calendar=garden
      as: events
      schema: {title: string, date: date}
      fallback: ./events.json
```

It lands in `md.data` like any file, with the same `schema` and `fallback`. Give it a schema, since two apps may give the same name different data. In an app that has no such data, the import is `F026` and takes its fallback, so a document that names `host:` works fully only in its own app. The playground has no `host:` data.

The spec's data imports have every rule.

Next: inputs the reader changes, for values the reader picks on the page: a plot size, a name, a month.

7. Inputs the reader changes

So far the page shows what the document and its files say. An *input* is a value the reader changes on the page: a name typed in, a size picked, a box ticked. Everything computed from it follows at once. This chapter gives the newsletter a sign-up form that works out the year's fee, and a list of jobs that loads the month the reader picks.

The newsletter, with a sign-up form

```
---
md:
  version: 1
  inputs:
    name: string = ""
    size: small | large = small
    plots: number = 1
    tools: boolean = false
    start: date = (md.doc.workday)
    month: november | december = november
  data:
    - from: ./jobs-{{md.input.month}}.yaml
      as: jobs
title: Elm Street Garden
issue: November 2026
workday: "2026-11-07"
fees: {small: 25, large: 40, tools: 5}
---

# {{md.doc.title}}

News from the plots, {{md.doc.issue}}.

## Take a plot for next year

::: grid {min=14rem gap=m}
:: text-input[Your name: none yet] {bind=name}
:: select-input[Size: small] {bind=size}
:: number-input[Plots: 1] {bind=plots}
:: toggle-input[Share the tool shed: no] {bind=tools}
```

```
:: date-input[First work day: November 7, 2026] {bind=start}
::: /grid

::: box {padding=m border=subtle border-radius=m}

::: if {when=(md.input.name != "")}

Thank you, **{{md.input.name}}**.

::: /if

:value[1 small plot]{of=(md.number(md.input.plots, 0) + " " + md.input.size
+ " " + md.plural(md.input.plots, "plot", "plots"))} for the year:
:value[$25.00]{of=(md.currency(md.doc.fees[md.input.size] * md.input.plots
+ (md.input.tools ? md.doc.fees.tools : 0.0), "USD"))}, :value[without the
tool shed]{of=(md.input.tools ? "with the tool shed" : "without the tool
shed")}. Your first work day is :value[Saturday, November 7, 2026]{of=
(md.date(md.input.start, "full"))}.

::: /box

## Jobs by month

:: select-input[Month: november] {bind=month}

::: each {in=(md.data.jobs) as=job join=list}

- {{job}}

::: /each
```

Try it in the playground. Type a name, pick *large*, make it 2 plots and tick the tool shed: the box says *Thank you*, then *2 large plots for the year: $85.00, with the tool shed*. Pick *december*, and the jobs change.

The jobs for each month are a file beside the document:

```
- Turn the compost
- Plant the garlic
- Mend the gate by plot 4
```

```
- Cover the empty beds
- Oil the tools
```

```
- Order next year's seeds
```

Declaring an input

Inputs are declared in the frontmatter, under `md.inputs`, one per line: a name, a type, `=` and the value it starts with, its *default*.

```
md:
  inputs:
    plots: number = 1
```

The name is one word, or words joined by `_`, as an `each` name is (chapter 5). The page reads the input as `md.input.plots`, in `{{…}}` and in expressions, as it reads `md.doc`.

Give every input a default: it is what the page shows before the reader changes anything, and what a reader without an interactive page sees.

The types, and their components

Each type has a built-in component that shows the input and lets the reader change it:

Type	Written	Its component	The reader
text	`name: string = ""`	`text-input`	types text
a number	`plots: number = 1`	`number-input`	types a number
true or false	`tools: boolean = false`	`toggle-input`	ticks a box
a date	`start: date = "2026-11-07"`	`date-input`	picks a day
one of a few words	`size: small | large = small`	`select-input`	picks one from a list

The last is a *choice* (an enum): its words, separated by `|`, are what the list offers. Each is a name, so it is written without quotes, and starts with a letter. A choice with spaces in it needs the whole line quoted: `status: '"to do" | done = done'`.

A default can be an expression in parentheses, computed when the page is made. It sees the frontmatter and the `md.` functions, but not data or other inputs:

Written	Starts as
`start: date = (md.doc.workday)`	the frontmatter's work day
`start: date = (md.today())`	today
`start: date = (md.addDays(md.today(), 14))`	two weeks from today

Showing an input

An input component is a leaf. `bind` names the input it changes, and its brackets hold its fallback:

```
:: number-input[Plots: 1] {bind=plots}
```

Write the fallback as a label, a colon, and the default: *Plots: 1*. A plain renderer, such as GitHub's, shows it as it is, so the text still tells the reader what the value is. The playground uses the part before the colon as the control's label, *Plots*, next to a box holding 1.

`bind` must name an input the document declares, of a type the component edits: a `text-input` bound to `plots` is a diagnostic, `F030`, that says *`bind` binds a string, but `plots` is a number*. So is a `bind` to an input no one declared.

The common params of chapter 3 work as on any block: `:: number-input[Plots: 1] {bind=plots max-width=12rem}`. The grid in the newsletter puts the controls side by side, as many as fit (Part III).

The same input can have two components, or none. The playground shows every input of a document in a bar above its tabs, so you can try other values there too.

Using an input

An input is a value like any other, so everything in chapters 4 and 5 works with it:

- `{{md.input.name}}` writes it.
- `:value[…]{of=(…)}` computes with it: `md.doc.fees[md.input.size]` looks up the fee under the name the reader picked, and the `? :` adds the tool shed's fee when the box is ticked.
- `::: if {when=(md.input.name != "")}` shows the thanks only once a name is typed.
- A param can follow it: `background=(md.input.tools ? "accent" : "subtle")`.

When the reader changes an input, the page is made again with the new value: every part that uses it changes, and nothing else does. There is nothing else to write, no events and no scripts: values flow one way, from the inputs to what the page shows.

A number input holds any number the reader types, 1.5 too. `md.number(md.input.plots, 0)` writes it as a whole number, and the fee is computed from what was typed.

Loading by an input

A data import's `from` can use inputs, so the reader's choice picks the file:

```
md:
  inputs:
    month: november | december = november
  data:
    - from: ./jobs-{{md.input.month}}.yaml
      as: jobs
```

With *december* picked, the document loads `./jobs-december.yaml`. A `from` can use `md.input` and `md.doc`, nothing else: never `md.data`, so data from one place is never sent to another (`F031` otherwise). Each value fills one part of the path: it can't add a `/` or a `..`, so the file stays in its folder.

The same works for an address on a website (chapter 6): `from: https://weather.example.org/frost?street={{md.input.street}}`. The site's address itself must be written out, so the app can check it before anything is fetched; only the part after it may hold inputs. The site then learns what the reader picked, since it is part of the address.

Where the values come from

On the page, from the reader. An app can also give the values itself, for example from the page's address, and keep them between visits; how is the app's business. A value that doesn't fit its input's type is ignored, and the default is used, so a page never shows a number input holding a word.

Where nothing is interactive, a plain renderer or a printout, the page shows the defaults, and the input components show their fallbacks or their values.

Inputs and components

Inputs belong to the document being read. A component of your own (Part IV) doesn't see `md.input`: give it the value as a param, `:: fee-table[…] {size=(md.input.size)}`. An input of another document that this one loads (chapter 6) isn't used.

So input components go in the document's own text, never in a component's definition: there, no input is declared for `bind` to name, and `:: select-input[Size: small] {bind=size}` is `F030`. A form that lays its controls out, with their labels, is written in the document; a box or a grid around them can still be a component of your own, with the controls in its slot.

A component from a pack can change an input too, when its params say so and you bind one to it: a chart whose bars the reader clicks to pick a region, written `selected=region`. Its README says which params do. Since such a component's code can set the input, that input can't be used in a data import's `from` (`F031`): code could otherwise send what it likes through it.

The spec's inputs have every rule.

Next: dates, numbers and lists, for the functions that count, sum, sort and write dates.

8. Dates, numbers and lists

Chapters 4 and 5 used a few functions to write a date or filter a list. This chapter has the rest: counting days, sums and averages, sorting, grouping, and writing it all for readers. The newsletter gets the next work day, counted from today, and the season's harvest, added up from a spreadsheet.

The newsletter, by the numbers

```
---
md:
  version: 1
  data:
    - from: ./harvest.csv
      as: harvest
      schema: {date: date, plot: number, crop: string, kg: number}
title: Elm Street Garden
issue: November 2026
workdays: ["2026-11-07", "2026-11-21", "2026-12-05"]
---

# {{md.doc.title}}

News from the plots, {{md.doc.issue}}, for the week of :value[October 25,
2026]{of=(md.date(md.startOf(md.today(), "week"), "long"))}.

## The next work day

::: each {in=(md.slice(md.doc.workdays.filter(d, d >= md.today()), 0, 1))
as=next}

:value[Saturday, November 7, 2026]{of=(md.date(next, "full"))}, in :value[7
days]{of=(string(md.daysBetween(md.today(), next)) + " " +
md.plural(md.daysBetween(md.today(), next), "day", "days"))}. After it:
:value[November 21, 2026 and December 5, 2026]{of=
(md.list(md.doc.workdays.filter(d, d > next).map(d, md.date(d, "long"))))}.

:: #empty
```

```
No more work days this year. See you in the spring.

::: /each

## The harvest

Since :value[September 12]{of=(md.date(md.sort(md.data.harvest.map(h,
h.date))[0], "long"))}, the plots gave :value[30.0 kg]{of=
(md.number(md.sum(md.data.harvest.map(h, h.kg)), 1) + " kg")} in :value[7
pickings]{of=(string(size(md.data.harvest)) + " " +
md.plural(size(md.data.harvest), "picking", "pickings"))}, :value[4.3 kg]
{of=(md.number(md.sum(md.data.harvest.map(h, h.kg)) /
size(md.data.harvest), 1) + " kg")} each on average. The biggest was
:value[9.0 kg]{of=(md.number(md.max(md.data.harvest.map(h, h.kg)), 1) + "
kg")} of :value[squash]{of=(md.reverse(md.sortBy(md.data.harvest, "kg"))
[0].crop)}.

::: each {in=(md.sort(md.unique(md.data.harvest.map(h, h.crop)))) as=crop
join=list}

- **{{crop}}**: :value[7.7 kg]{of=
(md.number(md.sum(md.groupBy(md.data.harvest, "crop")[crop].map(h, h.kg)),
1) + " kg")} from :value[2 pickings]{of=
(string(size(md.groupBy(md.data.harvest, "crop")[crop])) + " " +
md.plural(size(md.groupBy(md.data.harvest, "crop")[crop]), "picking",
"pickings"))}

::: /each

The last three:

::: each {in=(md.slice(md.reverse(md.sortBy(md.data.harvest, "date")), 0,
3)) as=h join=list}

1. :value[October 24]{of=(md.date(h.date, "long"))}: {{h.crop}} from plot
{{h.plot}}, {{h.kg}} kg

::: /each

Bring your spare :value[beans, kale, leeks, or squash]{of=
(md.list(md.sort(md.unique(md.data.harvest.map(h, h.crop))), "or"))} to the
swap table by the gate.
```

Try it in the playground.

The harvest, one row per picking:

```
date,plot,crop,kg
2026-09-12,1,beans,3.5
2026-09-19,5,beans,4.2
2026-09-26,2,kale,2
2026-10-03,4,leeks,5.5
2026-10-10,2,kale,1.8
2026-10-17,1,squash,9
2026-10-24,4,leeks,4
```

Made on October 31, the page says the next work day is *Saturday, November 7, 2026, in 7 days*, then lists the harvest: *30.0 kg in 7 pickings, 4.3 kg each on average*, a line per crop (*beans: 7.7 kg from 2 pickings*), and the last three pickings, newest first. Add a row to the spreadsheet, and every figure follows. After December 5, the work days' `:: #empty` slot shows instead.

Dates

A date is text written as year, month and day, `"2026-11-07"`, in the frontmatter, in data and in inputs alike. Dates written this way sort and compare as they should, so `d >= md.today()` is true for today and the days after it, as `filter` uses it above.

Function	Gives	Example, on October 31, 2026
`md.today()`	today's date	*2026-10-31*
`md.addDays(date, n)`	the date `n` days later, earlier for a negative `n`	`md.addDays(md.today(), -7)` is *2026-10-24*
`md.daysBetween(a, b)`	the days from `a` to `b`, negative when `b` is before `a`	`md.daysBetween(md.today(), "2026-11-07")` is *7*
`md.startOf(date, unit)`	the first day of the date's `"week"`, `"month"` or `"year"`	`md.startOf(md.today(), "month")` is *2026-10-01*

A week starts on the day the document's language starts it: Sunday in English, so `md.startOf(md.today(), "week")` is *2026-10-25*, and Monday in German.

Today is the day the page is made, wherever that is. A page made once and kept, such as a printed one, keeps the day it was made. Within one page, every `md.today()` is the same day.

Writing a date

`md.date(date, style)` writes a date for readers, in one of four styles:

Style	Gives
`"short"`	*11/7/26*
`"medium"`	*Nov 7, 2026*
`"long"`	*November 7, 2026*
`"full"`	*Saturday, November 7, 2026*

The document's language decides the order and the words (chapter 4): with `lang: de`, `"long"` gives *7. November 2026*. `{{…}}` writes a date as it is stored, *2026-11-07*.

Times of day

Most documents need only dates. For the few that need the hour, `md.now()` is the moment the page is made, a *timestamp*, and its getters read it in the document's time zone, which `tz` under `md` sets:

```
---
md:
  tz: Europe/London
---

:value[Good morning]{of=(md.now().getHours() < 12 ? "Good morning" : "Good
afternoon")}, from the garden.
This page was made on :value[October 31, 2026]{of=(md.date(md.now(),
"long"))}.
```

Written	Gives
`md.now().getHours()`	the hour, 0 to 23
`md.now().getMinutes()`	the minute
`md.now().getDayOfWeek()`	the day of the week, 0 for Sunday
`md.dateOf(md.now())`	the date of a timestamp: *2026-10-31*

Without `tz`, the time zone is the app's, usually the reader's. A `tz` is a name such as `Europe/London` or `America/Chicago`, and it sets `md.today()` too.

Numbers

Numbers in the frontmatter, in data and in inputs can have decimals: `4.2`, `12`. Some numbers are always whole: counts, such as `size(…)` and `md.daysBetween(…)`, `md.loop`'s, and numbers you write without a point. The two mix in sums and comparisons, `md.input.plots * 25` and `size(rows) > 2.5`, with two things to know:

- **Dividing two whole numbers drops the rest.** `size(md.data.harvest) / 2` with 7 pickings is 3. Make one of them a number with decimals first: `double(size(md.data.harvest)) / 2` is 3.5. A number with decimals divided by a whole one is fine, as the average above is.
- **A position in a list is whole.** `md.doc.helpers[0]` is the first helper; with a position from an input, write `md.doc.helpers[int(md.input.n)]`.

`int(n)` drops the decimals, `double(n)` gives a number with them, and `string(n)` writes it as text, as `{{…}}` would: *4.2*, *12*.

Writing numbers

Function	Example	Gives
`md.number(n, digits)`	`md.number(1234.567, 1)`	*1,234.6*: grouped, rounded to that many decimals
`md.currency(n, code)`	`md.currency(1234.5, "EUR")`	*€1,234.50*, with the currency's own decimals

`md.number` rounds a half up, away from zero: `md.number(2.5, 0)` is *3*. The currency is a three-letter code, `"USD"`, `"EUR"`, `"GBP"`. Both follow the document's language: with `lang: de`, `md.number(1234.567, 1)` is *1.234,6*.

Sums, smallest and largest

These take a list of numbers, so a list of records goes through `map` first, as the harvest does: `md.data.harvest.map(h, h.kg)`.

Function	Gives
`md.sum(list)`	the sum; `0` for an empty list
`md.min(list)`, `md.max(list)`	the smallest, the largest; `null` for an empty list

An average is the sum over the count: `md.sum(…) / size(…)`. Check that a list isn't empty before dividing by its size, or writing its `md.max`: `null` can't be written with `string()`, and is

a diagnostic (`F024`).

Plurals

`md.plural(n, one, other)` picks a word by a number: `md.plural(1, "picking", "pickings")` is *picking*, and any other number gives *pickings*. Write the number beside it yourself, as the newsletter does.

Some languages have more forms than two. Give them by their names, `one`, `few`, `many` and `other`, and the document's language picks; `other` must be there:

```
---
md:
  lang: ru
plots: 5
---

На участке :value[5 грядок]{of=(string(md.doc.plots) + " " +
md.plural(md.doc.plots, {"one": "грядка", "few": "грядки", "many":
"грядок", "other": "грядки"}))}.
```

With 1 it writes *грядка*, with 3 *грядки*, with 5 *грядок*. The names are the CLDR plural categories, which also has `zero` and `two` for the languages that use them.

Lists

Chapter 5 chose items with `filter`, `exists`, `all` and `map`, written after a list. Two more of that kind:

Written	Gives
`list.exists_one(p, p.crop == "garlic")`	true when the condition is true for exactly one item
`"garlic" in md.doc.crops`	true when the list holds the value

And `size(list)` counts. The rest are functions that take a list:

Function	Gives
`md.sort(list)`	the values in order, smallest or earliest first

Function	Gives
`md.sortBy(list, "field")`	the records in order of one field: `md.sortBy(md.data.harvest, "kg")`
`md.reverse(list)`	the list turned around: largest or newest first, after a sort
`md.slice(list, start, end)`	the items from position `start` up to, not including, `end`: `md.slice(list, 0, 3)` is the first three, fewer if the list is shorter
`md.unique(list)`	each value once, in the order they first come
`md.groupBy(list, "field")`	the records grouped by one field's value, below
`md.list(list)`	the values written as a sentence: *beans, kale, and leeks*
`md.list(list, "or")`	the same with *or*: *beans, kale, or leeks*

A few things the newsletter relies on:

- **The first item.** `list[0]` is the first item, and an error on an empty list. `md.slice(list, 0, 1)` is a list of at most one item, so an `each` over it shows the item, or its `:: #empty` slot when there is none, as the next work day does.
- **Sorting.** Sorting keeps items that are equal in the order they came. Text sorts by its letters' codes, not by a dictionary: capitals before small letters, and letters with accents after *z*. Write the values you sort the same way, all small letters or all capitalized.
- **Sorting by a computed value.** `md.sortBy` takes a field's name, not a formula. To sort by something computed, `map` the records into ones with that field first: `md.sortBy(md.data.harvest.map(h, {"crop": h.crop, "grams": h.kg * 1000}), "grams")`.

Groups

`md.groupBy(md.data.harvest, "crop")` makes a record with one entry per crop, each holding that crop's rows, in the order the crops first come. Read one group by its name in brackets: `md.groupBy(md.data.harvest, "crop")["kale"]` is kale's two pickings.

`each` goes through a list, not a record, so the newsletter goes through the crops' names, `md.sort(md.unique(md.data.harvest.map(h, h.crop)))`, and reads each crop's group with `[crop]`: its sum, and its size. The same could be written with `filter`, `md.data.harvest.filter(h, h.crop == crop)`; the group says it once.

Text

Text has functions of its own, written after it:

Written	Gives
`"Plot 4".contains("4")`	true when the text holds the other
`"Plot 4".startsWith("Plot")`, `.endsWith("4")`	true when it starts or ends so
`"garlic".upperAscii()`, `"KALE".lowerAscii()`	*GARLIC*, *kale*: only the letters A to Z change
`" kale ".trim()`	*kale*, without the spaces around it
`"beans, kale".split(", ")`	a list, *beans* and *kale*
`md.doc.rotation.join(" → ")`	the list's items joined into one text
`"garlic".substring(0, 3)`	*gar*, from position 0 up to 3
`"garlic".replace("g", "G")`	*GarliG*: every match
`"garlic".indexOf("l")`	*3*, where it first is; -1 when it isn't

There is no matching by patterns: a function for that would work differently in different apps.

When a function is given the wrong thing

A function given something it can't use is a diagnostic, `F024`, that says what it takes: `md.date(d, "longer")` says the styles are `"short"`, `"medium"`, `"long"` and `"full"`, and `md.sum` over text says it needs numbers. Text from a CSV without a schema is the usual cause: chapter 6's schema makes its numbers and dates what they are.

Every function is in the spec's expression profile.

Next: boxes, rows, stacks and grids, the first chapter of Part III, where the newsletter's blocks go side by side.

9. Boxes, rows, stacks and grids

So far the newsletter's blocks come one under another, as in any Markdown page. This chapter puts them side by side: the notices in a stack, the work day's jobs in a row, and the plots as cards in a grid, as many across as the page has room for. Four built-in containers do it: `box`, `stack`, `row` and `grid`.

The newsletter, laid out

```
---
md:
  version: 1
title: Elm Street Garden
issue: November 2026
jobs: [Turn the compost, Mend the gate by plot 4, Plant the garlic]
plots:
  - {number: 1, gardener: Maya Okafor, crop: garlic}
  - {number: 2, gardener: Tomás Reyes, crop: kale}
  - {number: 3, gardener: "", crop: ""}
  - {number: 4, gardener: Priya Nair, crop: leeks}
  - {number: 5, gardener: "", crop: ""}
---

# {{md.doc.title}}

News from the plots, {{md.doc.issue}}.

::: stack {gap=s}

::: box {padding=m border-radius=m background=accent}

**The water goes off on November 15.** Empty your cans and hoses before
then.

::: /box

::: box {padding=m border-radius=m background=subtle}

**The tools go back in the shed** after the work day. Maya has the key.
```

```
::: /box

::: /stack

## Work day

Saturday, November 7, from nine to noon. On the list:

::: row {gap=s}

::: each {in=(md.doc.jobs) as=job}

::: box {padding="s m" border=subtle border-radius=full}

{{job}}

::: /box

::: /each

::: /row

## The plots

::: grid {min=12rem gap=m}

::: each {in=(md.doc.plots) as=p}

::: box {padding=m border=subtle border-radius=m background=(p.gardener ==
"" ? "accent" : "none")}

### Plot {{p.number}}

::: if {when=(p.gardener != "")}

{{p.gardener}}, growing {{p.crop}}.

:: #else

Free for next year. Ask Maya.

::: /if
```

```
::: /box

::: /each

::: /grid
```

Try it in the playground. The two notices sit one under the other with a small gap, and the three jobs are pills side by side. Drag the bar between the text and the page: with the page narrow, the pills wrap onto a second line and the plots are one card per row; as it widens, two, three, then all five cards fit across, and the free plots stand out in the accent colour.

The four layouts

Container	Lays out its blocks	Its own params
`box`	one under another, as the page does	`display` and the others below, which the three after it set for you
`stack`	one under another, with a gap between them	`gap`, `align-items`
`row`	side by side, wrapping onto the next line when there isn't room	`gap`, `align-items`, `justify-content`, `flex-wrap`
`grid`	in columns: as many as fit (`min`), or the columns you give	`min`, `gap`, `align-items`, `grid-template-columns`, `grid-template-rows`, `grid-template-areas`

Each of them is also a box, with the common params of chapter 3: `padding`, `border`, `background` and the others work on a row or a grid as on a box.

Items

A layout lays out its *items*: each block directly in it is one. A paragraph is an item, and so is a heading, a list, a table, or a container in it. In the newsletter, each notice is a box, so the stack has two items; the `each` in the row makes three boxes, so the row has three. `each` and `if` add their blocks to the layout around them, as if you had written them there.

To keep several blocks together as one item, put them in a box. Without the box, a card's heading and its line would be two items, side by side in a row or in cells of their own in a grid:

```
::: row {gap=m}

::: box {padding=m border=subtle}

### Plot 1

Maya Okafor, growing garlic.

::: /box

::: box {padding=m border=subtle}

### Plot 2

Tomás Reyes, growing kale.

::: /box

::: /row
```

`stack`

A stack lays its items one under another, as the page does, with `gap` between them. Its use is the gap: a page puts its own space around paragraphs and headings, but a box has none, so boxes one after another touch. In a stack they don't.

`align-items` says how wide its items are: `stretch`, the default, makes each as wide as the stack; `start`, `center` and `end` make each as wide as its content, at the start, the middle or the end.

`row`

A row lays its items side by side, each as wide as its content. When they don't fit, the row wraps onto the next line, as words do; with `flex-wrap=nowrap` it stays on one line, and its items shrink to fit.

- `align-items` lines items of different heights up: `stretch`, the default, makes them all as tall as the tallest; `start`, `center` and `end` line them up at the top, the middle or the bottom.
- `justify-content` says where the items go when there is room left on the line: `start` (the default), `center`, `end`, `between` for the space between them, and `around` for space around

each of them. `space-between` and `space-around`, CSS's names for the last two, work too.

A row of two items with `justify-content=between` puts one at each end, as a title and a date in a heading:

```
::: row {justify-content=between align-items=center}

**Elm Street Garden**

November 2026

::: /row
```

`grid`

A grid lays its items in columns. There are two ways to say which:

- **As many as fit.** `min=12rem` makes columns at least 12rem wide, as many as the grid has room for, and shares what is left between them. This is the grid for cards: a wide page has five across, a phone one. It is the grid in the newsletter.
- **The columns you give.** `grid-template-columns="2fr 1fr"` makes two columns, the first twice as wide as the second. `fr` is a share of the room left; a column can also be a length, `12rem`, or `auto`, as wide as its content. Each item takes the next cell, row by row.

A grid with neither has one column, so its items come one under another; chapter 10 gives it columns from a width up. `gap` is the space between the cells, and `align-items` lines up items in a row of cells as in a `row`. `grid-template-rows` and `grid-template-areas`, which place items in cells of their own, are in chapter 11.

Gaps and sizes

`gap` takes the same sizes as `padding`: `none`, `s`, `m`, `l`, or a length such as `1.5rem`. The words follow the app that shows the document, as chapter 3's do.

Any block takes a size:

Param	Values
`width`, `min-width`, `max-width`	a length, a percentage of the box around it, or `auto`
`height`	a length, a percentage, or `auto`
`aspect-ratio`	width over height, such as `16/9`, or `none`

A size includes the block's padding. A `width` on an item of a row makes it that wide instead of as wide as its content: `width=30%` in a row gives three to a line.

A table too wide for a phone

A table with many columns is wider than a phone. `scroll` lets it scroll sideways inside its own box, so the rest of the page stays put:

```
::: box {scroll}

| Plot | Gardener | 2023 | 2024 | 2025 | 2026 | Next year |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | Maya Okafor | beans | cabbages | roots | potatoes | beans |
| 2 | Tomás Reyes | cabbages | roots | potatoes | beans | cabbages |

::: /box
```

`box`, closer

`stack`, `row` and `grid` are boxes with their layout set: a stack is a box with `display=flex flex-direction=column`, a row one with `display=flex flex-direction=row flex-wrap=wrap`, a grid one with `display=grid`. You can write those on a `box` yourself, and the names come from CSS, so they mean what they mean there. The three names say what you mean in one word, and are what this book uses.

A box with none of them is `display=block`: its blocks come one under another, as on the page, and it is the box you put around anything to give it padding, a border or a background.

Next: widths and breakpoints, for layouts that change with the width: one column on a phone, two on a wider page.

10. Widths and breakpoints

A newsletter is read on phones as well as on wide screens, and a layout that suits one rarely suits the other. This chapter makes the newsletter change with the width: the news and the work day in one column on a phone and side by side on a wider page, more padding where there is room, and the plots as a short list on a phone and a table on a wide page.

The newsletter, at every width

```
---
md:
  version: 1
title: Elm Street Garden
issue: November 2026
news:
  - The water goes off on November 15. Empty your cans and hoses before
then.
  - The seed swap is on December 5, in the library's back room.
  - Tomás has spare kale seedlings, free to a good bed.
---

# {{md.doc.title}}

News from the plots, {{md.doc.issue}}.

::: grid {gap=m md:grid-template-columns="2fr 1fr" md:gap=l}

::: box

## News

::: each {in=(md.doc.news) as=item join=list}

- {{item}}

::: /each

::: /box
```

```
::: box {padding=m md:padding=l border-radius=m background=subtle}

## Work day

Saturday, November 7, from nine to noon. Bring gloves; we bring the tea.

::: /box

::: /grid

## The plots

::: box {hide=md}

- **Plot 1**: Maya Okafor, garlic.
- **Plot 2**: Tomás Reyes, kale.
- **Plot 4**: Priya Nair, leeks.

Plots 3 and 5 are free.

::: /box

::: box {show=md}

| Plot | Gardener | Growing | Since | Next year |
| ---: | --- | --- | --- | --- |
| 1 | Maya Okafor | garlic | 2019 | beans |
| 2 | Tomás Reyes | kale | 2023 | roots |
| 3 | free | | | |
| 4 | Priya Nair | leeks | 2021 | potatoes |
| 5 | free | | | |

::: /box
```

Try it in the playground. With the page narrow, the work day's box comes under the news, and the plots are a list. Drag the bar between the text and the page until the page is wider than 768 pixels (folding the text away is quickest): the work day moves beside the news, a third of the width, its padding grows, and the list gives way to the table.

Breakpoints

A *breakpoint* is a width at which a param takes another value. There are three, always the same:

Breakpoint	From
`sm`	480 pixels, a large phone
`md`	768 pixels, a tablet
`lg`	1024 pixels, a laptop

A param with a breakpoint's name and a colon in front applies from that width up: `md:padding=l` is `l` from 768 pixels up. The same param without one is the value below it. So you write the layout for a phone first, and then what changes on wider pages:

```
::: box {padding=s sm:padding=m lg:padding=l border=subtle}

Padding `s` on a small phone, `m` from 480 pixels, and `l` from 1024.
Between 768 and 1024 it stays `m`: each value holds until a wider one takes
over.

::: /box
```

In the newsletter, the grid has one column, as chapter 9's grid without columns does, until `md:grid-template-columns="2fr 1fr"` gives it two from `md` up. Its `gap` is `m`, and `l` from `md` up.

Most params take a breakpoint: the layout params of chapter 9, the common params of chapter 3, and those of chapter 11. A few don't, since they make no sense at one width and not another: `grid-area`, `scroll`, `fill`, `show` and `hide`, and `class`, `id`, `lang` and `dir`. A breakpoint on one of them is a diagnostic (`F008`), and so is a name other than `sm`, `md` and `lg`, such as `xl:gap=l`; the param is then left out.

Showing and hiding

`show` and `hide` take a breakpoint as their value:

Written	The block shows
`show=md`	from `md` up
`hide=md`	below `md`
`show=sm hide=lg`	from `sm` up to `lg`

The newsletter writes the plots twice, a list for phones (`hide=md`) and a table for wider pages (`show=md`), so the reader sees one of them at any width. Keep both saying the same thing:

which one a reader sees depends only on the width.

A hidden block is still on the page, only out of sight. A screen reader skips it too, so it doesn't read the plots twice. A plain Markdown viewer, such as GitHub's, shows both, one after the other.

Which width

Breakpoints follow the width of the box a block is in, not the width of the window. At the top of a document that is the page itself, which is as wide as the app makes it: the whole window, a column beside a list, the playground's page pane. A document shown in a narrow column lays out as on a phone, however wide the window is.

Inside a box, a block's breakpoints follow that box. The same two-column grid, in a box at most 24rem wide, never gets its second column, since 24rem is less than 768 pixels:

```
::: box {max-width=24rem}

::: grid {gap=m md:grid-template-columns="1fr 1fr"}

**News**

**Work day**

::: /grid

::: /box
```

So a part of a page lays out by the room it has, wherever you put it. A card in a grid follows the grid, not its own width: a card with `md:padding=l`, in a grid 1000 pixels wide, has `l`, though the card itself is only 300 pixels wide.

`each` and `if` aren't boxes, and neither are lists, quotes and table cells: a block in them follows the box around them.

A box that needs a width

A box whose blocks use breakpoints has its width from where it is, never from its content, since its content waits for its width. Where a box would be as wide as its content, as in a row (chapter 9), it would then shrink to nothing. So such a box needs a `width`, and without one it is a diagnostic (`W114`). This one, in a row, has the grid's breakpoints in it and no width:

```
::: row {gap=m}

::: box
::: grid {md:grid-template-columns="1fr 1fr"}
…
```

Give it one, and the grid in it follows that width:

```
::: row {gap=m}

::: box {width=60%}

::: grid {gap=s md:grid-template-columns="1fr 1fr"}

**News**

**Work day**

::: /grid

::: /box

**Contacts**

::: /row
```

The same holds in a stack where the box isn't stretched across it (an `align-items` or `align-self` other than `stretch`), in a grid where the box has a `justify-self`, and for a box placed by position (chapter 11) without both an `inset-inline-start` and an `inset-inline-end`. A box at the top, or in a plain box, stack or grid, is as wide as what holds it, and needs nothing.

Next: placing blocks, for blocks in cells of a grid with names, at a place in a box, or kept in view as the page scrolls.

11. Placing blocks

Chapters 9 and 10 let blocks flow into a layout, one after another. This chapter puts blocks in particular places: in cells of a grid with names, at a place in a box, as on a map, or kept in view while the page scrolls. The newsletter gets a map of the garden, with its plots where they are, and its notes beside the map on a wide page.

The newsletter, with a map

```
---
md:
  version: 1
title: Elm Street Garden
issue: November 2026
plots:
  - {number: 1, gardener: Maya Okafor, crop: garlic, x: "6%", y: "10%"}
  - {number: 2, gardener: Tomás Reyes, crop: kale, x: "38%", y: "10%"}
  - {number: 3, gardener: "", crop: "", x: "70%", y: "10%"}
  - {number: 4, gardener: Priya Nair, crop: leeks, x: "6%", y: "55%"}
  - {number: 5, gardener: "", crop: "", x: "38%", y: "55%"}
---

# {{md.doc.title}}

News from the plots, {{md.doc.issue}}.

::: grid {gap=m grid-template-areas="'map' 'plots' 'notes'" md:grid-
template-columns="2fr 1fr" md:grid-template-areas="'map notes' 'plots
notes'"}

::: box {grid-area=map aspect-ratio=3/2 border=subtle border-radius=m
background=subtle}

::: each {in=(md.doc.plots) as=p}

::: box {inset-inline-start=(p.x) inset-block-start=(p.y) width=26%
padding=s border-radius=s background=(p.gardener == "" ? "accent" :
"strong")}
```

```
**{{p.number}}**

::: /box

::: /each

::: box {inset-inline-end=4% inset-block-end=8% padding=s border=strong
border-radius=s}

Shed

::: /box

::: box {inset-inline-start=0 inset-block-end=0 padding=s}

Gate

::: /box

::: /box

::: stack {grid-area=plots gap=s}

## The plots

::: each {in=(md.doc.plots) as=p}

::: box {padding=s border=subtle border-radius=s}

::: if {when=(p.gardener != "")}

**Plot {{p.number}}**: {{p.gardener}}, growing {{p.crop}}.

:: #else

**Plot {{p.number}}** is free for next year.

::: /if

::: /box

::: /each

::: /stack
```

```
::: stack {grid-area=notes gap=s align-self=start md:sticky=top}

## Notes

::: box {padding=m border-radius=m background=accent}

**The water goes off on November 15.**

::: /box

::: box {padding=m border-radius=m background=subtle}

**Work day:** Saturday, November 7, from nine to noon. Mend the gate first.

::: /box

::: /stack

::: /grid
```

Try it in the playground. The map is a box three wide by two high, with the five plots on it in two rows, the free ones in the accent colour, the shed in its far corner and the gate at the bottom. With the page narrow, the map, the plots and the notes come one under another. From 768 pixels up (drag the bar between the text and the page, or fold the text away), the notes move beside the map and the plots, and stay in view as you scroll down the list of plots. Move a plot by changing its `x` or `y`.

Cells with names

`grid-template-areas` draws a grid's cells as words, a row in quotes for each row of the grid, a name for each cell:

```
grid-template-areas="'map notes' 'plots notes'"
```

This is two rows of two cells. The name `notes` is in both rows of the second column, so the area called `notes` takes both. `map` is the first row's first cell, `plots` the second row's. A block goes in an area with `grid-area`: `grid-area=map` puts the map box there. A `.` instead of a name is a cell left empty. Every row has as many names as the others; rows of different lengths are a diagnostic (`F009`). An area must also be a rectangle: the same name in cells that don't make one isn't reported, but the browser then ignores the grid's areas.

The columns' widths still come from `grid-template-columns` (here `2fr 1fr`), and the rows are as tall as their content, or what `grid-template-rows` says, written as columns are.

`grid-template-areas` takes breakpoints, and `grid-area` doesn't: a block always has the same name, and the grid says where each name goes at each width. So a grid that names its areas from `md` up names them below it too. In the newsletter, below `md` they are one column, `'map' 'plots' 'notes'`; from `md` up, two. Without areas at the narrowest width, the blocks' names would point at areas that aren't there, and the grid would lay them out in extra rows and columns of its own. Spell the names alike in both places: a name the grid doesn't have isn't a diagnostic.

On screen, and in reading order

Areas move blocks on screen, but never in the text. A screen reader, the keyboard's tab order, a copy and paste, and a plain Markdown viewer all follow the order you wrote. So write blocks in the order they should be read, and lay them out from there. Where blocks hold links or inputs, keep the order on screen close to the written one, so the keyboard doesn't jump about.

The same holds for `order`, which moves one block to the start or the end of its row, stack or grid: `order=first`, `order=last`, or `order=none` for where it was written. Here *Plant the garlic* shows first, though it is written last:

```
::: row {gap=s}

::: box {padding=s border=subtle}

Turn the compost

::: /box

::: box {padding=s border=subtle}

Mend the gate

::: /box

::: box {padding=s background=accent order=first}

Plant the garlic

::: /box
```

```
::: /row
```

Lining one block up

A layout lines up all its items with `align-items` (chapter 9). One item can differ:

Param	Values	Does
`align-self`	`start`, `center`, `end`, `stretch`	in a row or a grid, where the block is from top to bottom; in a stack, from side to side
`justify-self`	`start`, `center`, `end`	in a grid, where the block is in its cell from side to side; in a row, `end` pushes the block, and the blocks after it, to the end

In the newsletter, the notes are `align-self=start`, as tall as their content. A block in a grid is stretched to its area's height by default, and the notes' area is as tall as the map and the plots together.

`justify-self=end` in a row is the way to put a block at the end and the others at the start:

```
::: row {gap=s align-items=center}

**Elm Street Garden**

November 2026

::: box {justify-self=end padding="s m" border-radius=full
background=accent}

Sign up by November 30

::: /box

::: /row
```

Kept in view

`sticky=top` keeps a block at the top of the page while the page scrolls, as long as what holds it is still in view; `sticky=bottom` keeps it at the bottom, and `sticky=none` lets it scroll. A block

sticks within its own area: in the newsletter's grid, the notes stay in view while the reader scrolls past the plots, and leave with the grid.

The notes stick only from `md` up (`md:sticky=top`): below it they come after the plots, and there is nothing left to scroll past. They are `align-self=start` too, since a block as tall as its area has no room to move in it.

At a place in a box

The four *insets* put a block at a place in the box around it, as a note is pinned on a board. The block is then out of the flow: the other blocks lay out as if it weren't there, and it is drawn over them.

Param	Measured from the box's
`inset-inline-start`	start edge: the left, in a language written left to right
`inset-inline-end`	end edge: the right
`inset-block-start`	top
`inset-block-end`	bottom

Each takes a length, a percentage of the box's width (for the inline ones) or height (for the block ones), or `auto`, the default, which leaves it unset. The names say *start* and *end* rather than left and right, so a page in a language written right to left, such as Arabic, mirrors as it should: chapter 3's `dir=rtl`.

In the newsletter, each plot's label is placed with `inset-inline-start=(p.x)` and `inset-block-start=(p.y)`, from the plot's record, and the shed from the box's end and bottom. A placed block is as wide as its content, or as its `width` says: each plot is 26% of the map. With both `inset-inline-start` and `inset-inline-end`, it stretches between them. Among placed blocks, the later ones are on top.

The box a block is placed in is the nearest box around it: a `box`, `stack`, `row` or `grid`, or a component's box. `each` and `if` aren't boxes, nor are lists, quotes and table cells, so the plots' labels are placed in the map, through the `each`.

The box needs a height

Placed blocks give their box no height, since they are out of the flow. So a box that places blocks needs one: a `height`, an `aspect-ratio`, as the map has, or `fill` (below). Without one it is a diagnostic (`W119`):

```
::: box {padding=m}
::: box {inset-inline-end=0 inset-block-start=0}
New!
::: /box
::: /box
```

A block placed outside any box, at the top of the document, is `W119` too: it is placed against the page, whose height is the whole document's.

Placed from a width up

The insets take breakpoints. `md:inset-inline-start=60%` places a block from `md` up and leaves it in the flow below; `md:inset-inline-start=auto` takes a block placed below `md` back into the flow from there. The box then needs its height at every width where it places a block: `W119` is checked at each.

Filling the height

`fill` makes a block as tall as the room its box leaves, so a list in an app's side panel reaches the panel's bottom. It matters only where the document is shown in an area of a fixed height. In a page that grows with its content, as the playground's does, it does nothing. Like `scroll` (chapter 9), it is written alone, and takes no breakpoint.

Components without a box

Every component has a box, so a component in a grid is one item, as a box is. A component you write yourself (chapter 12) can instead have none, with `display=contents` on its definition: its blocks are then items of the layout it is in, one by one, as if you had written them there. Such a component takes no `padding`, `width` or other param for a box (`W120`), since it has none. `display=contents` goes only on a definition; on a box in a document, it is `F009`.

Next: definitions, for components of your own, written in CompMD.

12. Definitions

When the same shape comes back again and again, such as a card for each plot, you can write it once as a component of your own and use it like a built-in one. This chapter writes two: a card for a plot, used as a leaf, and a gardener's name, used inside a sentence. They are written in CompMD itself, so they need no code and work wherever the document is shown.

The newsletter, with components of its own

````
---
md:
  version: 1
title: Elm Street Garden
issue: November 2026
plots:
  - {number: 1, gardener: Maya Okafor, crop: garlic}
  - {number: 2, gardener: Tomás Reyes, crop: kale}
  - {number: 3, gardener: "", crop: ""}
  - {number: 4, gardener: Priya Nair, crop: leeks}
---

# {{md.doc.title}}

News from the plots, {{md.doc.issue}}.

```compmd component=plot-card {padding=s border=subtle border-radius=m
background=(gardener == "" ? "accent" : "none")}
params:
  number: number
  gardener: string
  crop: string
---
# Plot {{number}}

:value[Maya Okafor, growing garlic.]{of=(gardener == "" ? "Free for next
year." : gardener + ", growing " + crop + ".")}
```
````

````
```compmd component=gardener
params:
  name: string
  plot: number
---
**{{name}}** (plot {{plot}})
```

## The plots

::: grid {min=12rem gap=m}

::: each {in=(md.doc.plots) as=p}

:: plot-card[Plot {{p.number}}] {number=(p.number) gardener=(p.gardener)
crop=(p.crop)}

::: /each

::: /grid

## Thanks

To :gardener[Maya Okafor (plot 1)]{name="Maya Okafor" plot=1} for turning
the compost, and to :gardener[Priya Nair (plot 4)]{name="Priya Nair"
plot=4} for mending the gate.
````

Try it in the playground.

The page shows four cards, one per plot, each with its number as a heading and a line about it. Plot 3 has no gardener, so its card has the accent background and says *Free for next year.* The thanks name Maya and Priya in bold, with their plots.

A definition

A component is defined in a fenced code block, the kind you'd write for code, whose first line says it is a definition:

````
```compmd component=plot-card {padding=s border=subtle}
````
````

- `compmd` is the language the component is written in: CompMD itself. Components written in code use other languages, and are the Developer's Guide's.
- `component=plot-card` names it. The name is how you write it in the document: `:: plot-card[…] {…}`.
- The params in braces, which may be left out, style the component's box (below).

Inside the fence come three parts:

1. The **signature**: what the component takes. Here `params` lists three params and the type of each value. Chapter 15 has every type; `string` (text) and `number` are the ones you'll use most.
2. A line of three dashes, `---`.
3. The **body**: CompMD, as you'd write it anywhere in a document. It is what each use of the component shows.

The fence itself doesn't show on the page. A plain Markdown viewer, such as GitHub's, shows it as a code block, so a reader there can see how the component is made.

Params in the body

In the body, a param is a name of its own: `{{number}}` inserts the plot's number, as `{{md.doc.title}}` inserts the frontmatter's title. In an expression, write it without braces: `gardener == "" ? "accent" : "none"`.

A param whose name has a `-`, such as `last-year`, is written with `_` in the body: `{{last_year}}`, because expressions can't use `-` in names.

The body also sees the frontmatter of the document the definition is in, as `md.doc`, but not the reader's inputs (chapter 7): `md.input` in a body is a diagnostic (`F014`). A component gets an input's value as a param instead (chapter 15).

Using a component

You write your component as any component: as a leaf, or inline, or, once it has slots (chapter 13), as a container. Its params go in braces, with the values its signature says:

```
:: plot-card[Plot 1] {number=1 gardener="Maya Okafor" crop=garlic}
```

A bare value is read by its type: `number=1` is the number 1, because the signature says `number`. A value can be an expression too, as in the `each` above: `number=(p.number)`.

The brackets hold the fallback, as for any leaf: the text a plain viewer shows, and the page shows if the component can't be drawn (chapter 15). Write what the card says: *Plot 3*, or *Plot 1: Maya Okafor, garlic*.

An **inline** use, `:gardener[Maya Okafor (plot 1)]{…}`, puts the body's text in the sentence. That works when the body is one paragraph, as `gardener`'s is; a body of several blocks can't go in a sentence, and an inline use of it shows its fallback.

The processor checks every use against the signature, and says what's wrong:

Written	Diagnostic
`:: plot-crad[…]`	`F002`: no such component, *did you mean `plot-card`?*
`colour=red`	`F008`: `plot-card` has no param `colour`
`number=one`	`F009`: `number` takes a number
no `number`	`F010`: `plot-card` needs `number`, and shows its fallback
`::: plot-card` … `::: /plot-card`	`F018`: `plot-card` declares no slots, so it takes no body

The component's box

Each use of a component as a block is one box, as each `::: box` is, and the body goes in it. The params after the name on the fence style that box: they take the common params of chapter 3, and `box`'s own, such as `display` and `gap` (Part III). They can be expressions of the component's params: `background=(gardener == "" ? "accent" : "none")` makes a free plot's card stand out.

The params of a use style the same box, and win over the fence's. `:: plot-card[…] {… padding=m}` gives that card more room, and keeps the fence's border and background.

A component that should add no box of its own, such as a name inside a list of cards, has `display=contents` on its fence, and nothing else. Its blocks then sit directly in whatever holds it. Such a component has no box to style, so a box param on its use is ignored, with a warning (`W120`).

Headings in a body

The card's title is written `# Plot {{number}}`, yet on the page it is a third-level heading. Headings in a body are relative to where the component is used: `#` is one level below the

heading above the use, `##` two below, and so on. The cards are under *The plots*, a second-level heading, so their titles are third-level. Used under the document's title, they would be second-level. So the page's outline stays right wherever a component goes. Chapter 15 shows how a component keeps its headings as written.

Where definitions go

A definition goes anywhere at the top level of the document, not inside a container, and its name is known to the whole document, above and below it. These are near the top, so a reader of the text sees them before their uses; at the end, they would keep the newsletter's text together. Inside a container, such as a `box`, it isn't a definition, and the component's uses are `F002`.

When several documents use the same components, keep the definitions in a file of their own and import them: chapter 15 shows how.

Components that use components

A body can use any component the document can name: built-in ones, as `plot-card` uses `value`, ones from packs, and your own. So a `gardener-list` could use `gardener` for each name. What a component can't do is use itself, directly or through another component: that would never end, and is `F019`.

Names

A component's name is made of letters, digits, `-` and `_`, and starts with a letter. Like the built-in ones, names are usually lower case words joined by `-`: `plot-card`, `work-day`.

- Two definitions with one name are `F013`, and the first one is used.
- `if`, `each`, `value` and `box` are CompMD's own, and can't be defined (`F013`).
- A definition named like another built-in component, such as `row` or `grid`, replaces it in this document, with a warning (`W112`), so check that's what you mean.

Next: slots, for components with a body you write, in parts.

13. Slots

A component of your own can wrap content you write each time, as `box` does: a notice around its message, a work day around its plans. That content goes in the component's *slots*. This chapter gives the newsletter a notice with one slot and a work day with three, and shows how a component decides where each part of its body goes.

The newsletter, with slots

````
---
md:
  version: 1
title: Elm Street Garden
issue: November 2026
---

# {{md.doc.title}}

News from the plots, {{md.doc.issue}}.

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

{{md.slot}}
```

```compmd component=work-day {padding=m border=subtle border-radius=m}
params:
  date: date
  hours: string
slots: [default, jobs, bring]
---
# Work day: :value[Saturday, November 7, 2026]{of=(md.date(date, "full"))}

{{hours}}.
````
````

````
{{md.slot}}

## On the list

{{md.slot.jobs}}

::: box {padding=s border-radius=s background=subtle}

{{md.slot.bring}}

::: /box
```

::: notice {title="The water goes off on November 15"}

Empty your cans and hoses before then. The taps come back on in March.

::: /notice

::: work-day {date=2026-11-07 hours="From nine to noon"}

The last work day of the year, and the garlic goes in.

:: #bring

**Bring** gloves and a trowel; we bring the tea.

:: #jobs

- Turn the compost
- Mend the gate by plot 4
- Plant the garlic

::: /work-day
````

Try it in the playground.

The page shows the notice in a shaded box, then the work day: its date as a heading, *From nine to noon*, the line about the garlic, the heading *On the list* with the three jobs, and what to bring in a shaded box at the end, though the text gives it before the jobs.
````

Declaring slots

A component takes a body only when its signature lists slots, under `slots`:

```
slots: [default, jobs, bring]
```

`default` is the slot for the body itself, what you write after the opener. The other names are the parts you start with a slot line, `:: #jobs`. List `default` whenever the component takes a body of its own, as both of these do.

A component without `slots` takes no body, so it is used as a leaf or inline (chapter 12); a body given to it is `F018`.

Placing slots in the body

In the body, `{{md.slot}}` places the default slot's content, and `{{md.slot.jobs}}` the `jobs` slot's. Each slot's content is blocks, paragraphs and lists, so put it on a line of its own.

The body decides where each slot goes. The work day places `jobs` under a heading of its own, and `bring` in a shaded box at the end. A slot can be placed anywhere a block can: in a `box`, in an `if`, even twice.

A slot the use doesn't give is empty, and nothing shows where it is placed. What the body puts around it still does: a work day without `:: #jobs` would still show the heading *On the list*.

Filling slots

A use writes the default slot first, straight after the opener, then each named slot from its slot line to the next one, or to the closer:

````
---
title: Elm Street Garden
---

```compmd component=notice
params:
  title: string
slots: [default, footer]
---
# {{title}}
````
````

```
{{md.slot}}

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

```
::: notice {title="Plots for next year"}

Sign up by November 30.

:: #footer

Plots go to last year's gardeners first.

::: /notice
```

- The default slot has no slot line, so it always comes first.
- The named slots come in any order: the work day gives `bring` before `jobs`, and the page shows them where the body places them.
- A slot line starts at the container's own level, not indented under a list item, which would put it inside the item (chapter 5).

What can go wrong:

Written	Diagnostic
`:: #extra`, a slot the component doesn't declare	`F006`, which says the component has no slot `extra`
`:: #jobs` twice in one use	`F007`, and the second one's content joins the first
a slot line outside any container	`F015`, and the line is ignored

What slot content sees

The content of a slot is yours, not the component's: it is written in your document, and reads as it would anywhere in it. `{{md.doc.title}}` in a slot is your title, an `each`'s name around the use can be used inside it, and so can any component your document can name. The component's params are the component's: `{{hours}}` in a slot is a diagnostic (`F014`).

Headings in a slot keep the levels you wrote. Only the body's own headings follow where the component is used (chapter 12), since the body's writer can't know where it will go, and you do.

Slots and their names

A slot's name is a word, such as `jobs` or `footer`, or words joined by `-`, as params' are: `:: #what-to-bring`. In the body, a path's steps can't hold a `-`, so it is written with `_`: `{{md.slot.what_to_bring}}` places it. Two slots of one component whose names differ only in `-` and `_` are `F013`, since their paths would be the same.

The built-in containers have slots too: `if` has `else`, and `each` has `before` and `empty` (chapter 5). They are filled the same way.

A component with slots, as a leaf

A component with slots can still be used as a leaf: `:: notice[No news this month] {title="From the committee"}`. Its slots are then empty, and the page shows the box with its title. That's useful when the body has something to show without them.

Next: template slots, for a slot that shows once for each item of a list.

14. Template slots

Some components show a list: a bed with its plots, a calendar with its events. A *template slot* lets such a component take the list as a param and the way to show each item as a slot, written by you. This chapter gives the newsletter a `bed` component that shows each bed's plots, with a count and a message for a bed that has none.

The newsletter, by bed

```
---
md:
  version: 1
title: Elm Street Garden
issue: November 2026
beds: [North, South, East]
plots:
  - {number: 1, gardener: Maya Okafor, crop: garlic, bed: North}
  - {number: 2, gardener: Tomás Reyes, crop: kale, bed: North}
  - {number: 3, gardener: Jun Park, crop: beans, bed: South}
  - {number: 4, gardener: Priya Nair, crop: leeks, bed: South}
---

# {{md.doc.title}}

News from the plots, {{md.doc.issue}}.

```compmd component=bed {display=flex flex-direction=column gap=s padding=m
border=subtle border-radius=m}
params:
  title: string
  plots: array
  as: name
slots:
  default: {each: plots, as: as}
  empty: {}
---
# The {{title}} bed

:value[2 plots]{of=(string(size(plots)) + " " + md.plural(size(plots),
```
```

```
"plot", "plots"))}

{{md.slot}}

::: if {when=(size(plots) == 0)}

{{md.slot.empty}}

::: /if
```

```
## The beds

::: grid {min=14rem gap=m}

::: each {in=(md.doc.beds) as=b}

::: bed {title="{{b}}" plots=(md.doc.plots.filter(p, p.bed == b)) as=plot}

**Plot {{plot.number}}**: {{plot.gardener}}, growing {{plot.crop}}.

:: #empty

No plots here yet. Ask Maya for one.

::: /bed

::: /each

::: /grid
```

Try it in the playground.

The page shows three boxes side by side. *The North bed* says *2 plots* and has a line for each of Maya's and Tomás's plots; *The South bed* has Jun's and Priya's; *The East bed* says *0 plots* and *No plots here yet.* Add a plot with `bed: East`, and it moves into the third box.

A slot for each item

The signature says which slot is a template, and for which list:

```
params:
  title: string
```

```
  plots: array
  as: name
slots:
  default: {each: plots, as: as}
  empty: {}
```

- `plots` is an `array` param: a list, here of records.
- `as` is a `name` param: the name the use gives each item. Its value is a name, not text or an expression.
- Under `slots`, each slot now has its options, in braces. `{each: plots, as: as}` makes the default slot a template: it shows once for each item of `plots`, with the item under the name `as` holds. `empty: {}` is an ordinary slot, with no options.

The params can have other names: `{each: rows, as: call}` works with `rows: array` and `call: name`. Calling them `rows` and `as`, or after what they hold, keeps uses easy to read.

Writing the template

A use passes the list, and picks the name:

```
::: bed {title="North" plots=(md.doc.plots.filter(p, p.bed == "North"))
as=plot}

**Plot {{plot.number}}**: {{plot.gardener}}, growing {{plot.crop}}.

::: /bed
```

The body you write is the template, written once and shown once per plot, with `{{plot.number}}` and `{{plot.gardener}}` that plot's. Inside it, `md.loop` says which time round it is, as in `each` (chapter 5): `{{md.loop.number}} of {{md.loop.count}}`.

The name is yours: `as=plot` here, `as=p` or `as=entry` as well. It is one word, or words joined by `_`, as `each`'s `as` is. Some words are taken by expressions, such as `in` or `true`, and can't be used (`F013`). An expression, `as=(plot)`, is `F009`: the name is written as it is.

The name is known in the template only. In another slot, such as `empty`, `{{plot.number}}` is a diagnostic (`F014`), since there is no plot there.

A template slot that isn't the default one is filled from its slot line, as any named slot is: a slot declared `item: {each: rows, as: as}` takes its template after `:: #item`.

Where the items go

In the body, `{{md.slot}}` places every item's template, one after another, in the list's order. The body puts what goes once around them: the title above, and the message for an empty bed below. It can also use the list itself in expressions, as `size(plots)` counts it, both for the count and to show `empty` only when there are no plots.

The fence styles the bed's box as a column (`display=flex flex-direction=column`), with a gap between its blocks. So the heading, the count and each plot's line are spaced evenly.

A component written in CompMD places the items together, where `{{md.slot}}` is. A component written in code, such as a calendar or a board from a pack, can place each item where it belongs, in its day or its column. You write the template the same way for both.

Choosing the items

Which items a component shows is the use's to say, with an expression: `plots=(md.doc.plots.filter(p, p.bed == b))` gives the bed only its own plots. Sorting goes the same way, with `md.sortBy` (chapter 5).

A value worked out for each item goes in the list first, with `map`, which makes a new record per item:

```
plots=(md.doc.plots.map(p, {"number": p.number, "gardener": p.gardener ==
"" ? "nobody yet" : p.gardener}))
```

Template slot or `each`

`each` repeats any content once per item, and a component can go inside it: an `each` around a `plot-card` gives a card per plot, as in chapter 12. A template slot is for a component that is itself about the list:

Use	When
`each` around a component	each item is a component of its own, and what goes around the list is yours to write each time
a template slot	the component frames the list, with its heading, its count, its layout, or its message when the list is empty

The newsletter uses both: `each` makes a bed per name in `beds`, and each bed's template slot shows its plots. In fact `each` is itself a component with a template slot, `default: {each: in, as: as}`, and the slots `before` and `empty`.

A plain Markdown viewer shows a template slot's template once, as written, as it shows an `each`'s.

Next: signatures and fallbacks, for every type a param can have, components kept in a file of their own, and what shows when a component can't be drawn.

15. Signatures and fallbacks

A component's signature is what every use of it relies on: the params it takes, of which types, which ones it needs, and its slots. This chapter moves the newsletter's components into a file of their own, a kit any issue can import, gives them complete signatures, and shows what a page shows when a component can't be drawn.

The newsletter, with a kit

The kit is a CompMD document of its own, in a folder beside the newsletter:

````
---
md:
  version: 1
  export:
    names: [plot-card, notice]
garden: Elm Street Garden
---

# The garden's components

The components every issue of the {{md.doc.garden}} newsletter uses.

```compmd component=plot-card {padding=s border=subtle border-radius=m}
params:
  number: number
  gardener: string?
  crop: string = "something new"
  size: small | large = small
---
# Plot {{number}}, {{size}}

::: if {when=(gardener == null)}

Free for next year: ask at the {{md.doc.garden}} shed.

:: #else

**{{gardener}}**, growing {{crop}}.
````
````

````
::: /if
```

```compmd component=notice {padding=m border-radius=m background=(tone ==
"urgent" ? "accent" : "subtle")}
params:
  title: string
  tone:
    type: plain | urgent
    default: plain
    doc: An urgent notice has the accent background.
slots: [default]
---
# {{title}}

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

The newsletter imports what it uses from it:

```
---
md:
  version: 1
  import:
    - from: ./parts/garden-kit.comp.md
      names: [plot-card, notice]
title: Elm Street Garden
issue: November 2026
---

# {{md.doc.title}}

News from the plots, {{md.doc.issue}}.

::: notice {title="The water goes off on November 15" tone=urgent}

Empty your cans and hoses before then.

::: /notice

## The plots
```
```

```
::: grid {min=12rem gap=m}

:: plot-card[Plot 1, large: Maya Okafor, growing garlic] {number=1
gardener="Maya Okafor" crop=garlic size=large}

:: plot-card[Plot 2, small: Tomás Reyes, growing something new] {number=2
gardener="Tomás Reyes"}

:: plot-card[Plot 3, small: free for next year] {number=3}

::: /grid
```

Try it in the playground.

The page shows the notice with the accent background, then three cards: *Plot 1, large* for Maya's garlic, *Plot 2, small* for Tomás, growing *something new*, since the use gives no crop, and *Plot 3, small*, free for next year.

A file of components

Any CompMD document can offer its components to others. It lists them in its frontmatter, under `md.export`:

```
md:
  export:
    names: [plot-card, notice]
```

- **`names`** lists the components it offers. Those it doesn't list stay its own: a kit can have helpers its components use, which no other document sees.
- A name can be offered under another one, `{name: notice, as: garden-notice}`, and is then imported by that name.
- `names: "*"` offers every component the file defines.

The newsletter imports them under `md.import`, as it would a pack (chapter 16), with the file's path in `from`:

```
md:
  import:
    - from: ./parts/garden-kit.comp.md
      names: [plot-card, notice]
```

A path starts with `./`, and is read from the folder of the document it's written in. The kit is in a folder of its own, `parts`, so the playground lists only the newsletter as the example's document, with the kit as a file it reads. Next month's issue imports the same kit, and every issue's cards change together when the kit does.

What can go wrong:

Written	Diagnostic
a file that isn't there	`F026`, and the uses of its names show their fallbacks
a name the file doesn't export	`F021`, on that name
two files that import each other	`F020`, on the import that closes the circle

A pack (chapter 16) is a file like the kit, published for anyone to import, with its own name and version under `md.export`.

What a component sees, from a kit

A component reads the kit's frontmatter, not yours: `{{md.doc.garden}}` in `plot-card` is the kit's `garden`, wherever the card is used. So a kit means the same in every document that imports it.

Slot content is the other way round: it is written in your document, so it reads your frontmatter (chapter 13). The notice's message could say `{{md.doc.issue}}`, and it would be your issue.

Which name wins

A document can name a component from several places. When two have the same name, this decides:

Comes first	Where the name comes from
1	the document's own definitions
2	names an import lists
3	`"*"` imports, and the built-in components
4	components the app showing the document gives every document

- A definition and a listed import of the same name, or two listed imports of one name, are `F013`.

- A name from a `"*"` import that you also define or list is yours, without a diagnostic. So a kit or pack that adds a component later never changes what a name you chose means.
- The same name from two `"*"` imports, or from one and the built-in components, is `F023` when you use it. List the one you mean to settle it.
- A definition or listed import named like a built-in component, such as `grid`, replaces it, with `W112`. `if`, `each`, `value` and `box` can't be replaced (`F013`).

The signature

A signature is written in YAML, as the frontmatter is. It has up to three keys:

Key	Holds
`params`	each param's name and its declaration: its type, and its default if it has one
`slots`	the slots, a list of names (chapter 13), or names with options (chapter 14)
`headings`	`relative`, the default, or `absolute`

`headings: absolute` keeps the body's headings at the levels written, for a component such as a title page, whose `#` must stay a first-level heading wherever it is used. Without it, headings follow where the component is used (chapter 12).

Types

A declaration starts with the type of the value:

Type	Takes	Written in a use
`string`	text	`title="Work day"`, or `title=Saturday` for one word
`number`	a number	`number=4`, `ratio=0.5`
`boolean`	true or false	`open=true`, or `open` alone for true
`date`	a date, year, month and day	`day=2026-11-07`
`array`	a list	`plots=(md.doc.plots)`
`object`	a record	`editor=(md.doc.editor)`
an enum, `small | large`	one of the names listed	`size=large`

Type	Takes	Written in a use
`name`	a name, for a template slot (chapter 14)	`as=plot`

A bare value is read as its param's type: `number=4` is a number, `day=2026-11-07` a date, `size=large` checked against the enum's names. A value in quotes is always text, so `number="4"` is `F009`, as is `size=medium` or `open=yes`. An expression's value must have the param's type too, or it is `F024`.

The values of the layout params have types of their own, for a component that passes a param on to its box: `space` (`s`, `m`, `l` or a length), `length`, `color`, `surface` (a background), `border`, `radius`, `tracks` and `areas`. A component fenced `{display=flex gap=(gap)}` with `gap: space = s` lets each use choose its gap.

A type misspelt, such as `numbr`, is `F037`, which suggests the right one. The param then takes any value, so one mistake gives one diagnostic, not one for every use.

Required, optional and defaults

Declared	A use without it
`number: number`	is `F010`: the param is required, and the component shows its fallback
`gardener: string?`	gives no value: in the body, `gardener == null` is true, and `{{gardener}}` writes nothing
`crop: string = "something new"`	gives the default, *something new*
`size: small | large = small`	gives `small`

A default is written as a value in a use is: `= 3`, `= small`, `= "two words"` in quotes when it has a space.

Plot 3's card shows how they work together: its use gives only `number`, so `gardener` has no value, the body's `if` shows *Free for next year*, and `size` is `small`.

The long form

A declaration can also be a record, for what a line can't say well:

```
tone:
  type: plain | urgent
  default: plain
  doc: An urgent notice has the accent background.
```

`type` is the type, with a `?` if it's optional, `default` the default, and `doc` a sentence about the param, for the people who use the component and the tools they write with.

Inputs

The reader's inputs (chapter 7) are declared in the same way, under `md.inputs`: a name, a type and a default. A component can't read them itself; the document passes an input's value to it as a param, and the component shows it again whenever the reader changes it:

````
---
md:
  inputs:
    size: all | small | large = all
---

```compmd component=showing
params:
  size: all | small | large
---
Showing {{size}} plots.
```

:: select-input[Show: all plots] {bind=size}

:: showing[Showing all plots] {size=(md.input.size)}
````

Signatures have one more type, `input<…>`, for a component that changes an input itself, such as a chart whose bars the reader clicks. Only components written in code use it, so it is the Developer's Guide's.

Fallbacks

A component written in CompMD can be drawn by every app that shows CompMD, so it shows its fallback only when something is wrong, or in a plain Markdown viewer. What shows in its place:

Component	Its fallback
a leaf, or an inline component	the text in its brackets
a container	its slots, the default one first and then the named ones in the order written, as plain Markdown

So nothing you wrote is lost: a broken `work-day` still shows its jobs and what to bring, without its heading and its box.

A component shows its fallback when:

- a required param is missing (`F010`), or a required param's value is wrong (`F009`), so the param is left out;
- its name is unknown (`F002`), or it leads back to itself (`F019`);
- the file or pack it comes from can't be had (`F026`);
- it is used inline, and its body isn't one paragraph.

A param that isn't required and has a wrong value is left out instead, with its diagnostic, and the component is drawn with the param's default.

In a plain Markdown viewer, such as GitHub's, every leaf and inline component shows its fallback, every container its content, and every definition its fence, as code. Writing each fallback as what the component shows, *Plot 3, small: free for next year*, keeps the newsletter readable there too.

Next: Part V, using a pack, for components other people publish.

16. Using a pack

A *pack* is a set of components someone else wrote and published, for any document to use: diagrams, charts, maps, ratings. This chapter gives the newsletter a diagram of the garden's crop rotation, drawn by a pack from npm, and says what happens when a pack can't be had.

The newsletter, with a diagram

```
---
md:
  version: 1
  import:
    - from: "@comp-markdown/mermaid@0"
      names: [mermaid]
title: Elm Street Garden
issue: November 2026
rotation: [Beans, Cabbages, Roots, Potatoes]
---

# {{md.doc.title}}

News from the plots, {{md.doc.issue}}.

## Next year's beds

Each bed moves one step along the rotation every spring, so nothing grows
where it grew last year, and the soil gets a rest from each crop.

:: mermaid[Beans, then cabbages, then roots, then potatoes, then beans
again] {code=("graph LR; " + md.doc.rotation.join("  +
md.doc.rotation.join(" --> ") + " " --> " + md.doc.rotation[0])}

Your bed's crop next year is the one after this year's.
```

Try it in the playground. The first time, the diagram's text shows for a moment while the pack arrives; then the diagram is drawn. Add `Onions` to the rotation, and it grows a step.

Importing a pack

A document says which packs it uses in its frontmatter, under `md.import`, one entry per pack:

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

- **`from`** is the pack's name, `@comp-markdown/mermaid`, then `@` and the version you want. A pack's name always starts with `@` and has a slash in it: `@who/what`, the name it is published under.
- **`names`** lists the components you use from it. Each is then written as any component is, `::mermaid[…] {…}`, as if it were built in.

Two packs may each have a component of the same name. Give one another name with `as`, and write it by that name:

```
---
md:
  import:
    - from: "@comp-markdown/mermaid@0"
      names:
        - name: mermaid
          as: diagram
---
```

`names: "*"` takes every component the pack offers, under its own names. It's quicker, but a component you write and one the pack adds later may share a name: listing the names you use keeps your document meaning the same.

Which version

A pack publishes new versions as it improves, and the number after `@` says which ones your document takes:

Written	Takes
`@1`	the newest version whose number starts with 1 (1.0, 1.4.2, ...), never a 2
`@^0.1.1`	0.1.1, or a newer 0.1 version
`@0.1.1`	exactly 0.1.1

The newest version that matches is used, so a document written with `@1` gets the pack's fixes without a change. A pack raises its version's first number when it changes how it is used, so `@1` never takes a version 2 that might draw your document differently. Before version 1, a pack is still settling, and its second number plays that part: `@^0.1.1` stays with 0.1.

A document that must never change, such as a published report, *pins* its pack: an exact version, with the pack's fingerprint as `integrity`. If the pack were ever changed, the fingerprint wouldn't match, and the document would refuse it (`F035`) rather than show something you didn't write. You don't work fingerprints out yourself: tools write them, from the one npm publishes for each version.

```
---
md:
  import:
    - from: "@comp-markdown/mermaid@0.1.1"
      integrity: sha512-
EwevqkkUdxOsFLHV85W74wEL+LoyZhaZkmnj8bJax3zLG48uuiGrezFTixCdlS6VQC/GHVU8Ra0
vVXjeFftc7Q==
      names: [mermaid]
---
```

Where the pack comes from

Your document says which pack it wants; the app that shows it gets the pack. An app gives its documents the packs it was built with, and it may also fetch any pack published on npm the first time a document asks for it, as the playground does. Each app decides which packs its documents may use.

A pack the app can't give is a diagnostic, `F026`, that says why: it isn't published, no version matches the one you wrote, or the app doesn't allow it. The document still shows: each use of the pack's components shows its fallback, the text in brackets. That's why a fallback should say what the component shows (chapter 3): *Beans, then cabbages, then roots, then potatoes* reads well where no diagram can be drawn. A reader with a plain Markdown viewer sees the same text.

While a pack arrives

A pack fetched from npm takes a moment the first time; after that, the reader's browser keeps it. Until it arrives, the page shows everything else, and the pack's components show their fallbacks. When it's there, the page draws them, without a reload.

What a pack can do

A pack's components run in a *sandbox*: a box of their own, cut off from everything else. There they see only what your document gives them, their params and their slots, and draw in their place on the page. They can't reach the internet, read the page around them, or see who is reading.

So you can use a pack nobody you know has checked: the worst it can do is draw badly in its own box. An app may trust some packs with more, such as a map that needs to load its tiles, but only the app decides that, never the document: no line in a document can give a pack more than the app does.

What npm learns

An app with a server of its own fetches packs from there, and npm learns nothing about its readers. An app without one, such as the playground, lets the reader's browser fetch from npm. npm then learns the reader's address and the names and versions of the packs, as a site learns who loads its pictures; never the document, or what it says. Such an app says so.

Finding packs

Packs on npm carry the keyword `compmd-pack`, so searching npm for it lists them. A pack's page there, its README, says which components it has and what each one's params are.

While you write, the playground helps too: its Components tab lists every component your document can name, with where each comes from, and a param a component doesn't take is a diagnostic that says which ones it does (chapter 3).

What's in a pack

A pack is a CompMD document that *exports* components: it lists, in its own frontmatter, the components it offers. Some are written in CompMD itself, as you'll write your own in Part IV; others are written in code, such as this diagram, which draws with a library called Mermaid. To your document they're all the same: a name, params, and a fallback.

Next: code blocks a pack draws: a pack like this one can also draw a document's ```mermaid code blocks.

17. Code blocks a pack draws

Some packs draw code blocks: a document that imports `@comp-markdown/mermaid` gets every ```` ```mermaid ```` block drawn as a diagram, while the text stays an ordinary code block that any Markdown viewer can show. This chapter gives the newsletter a diagram of the compost, written as a code block, and shows when to write a block and when a leaf.

The newsletter, with a code block

````
---
md:
  version: 1
  import:
    - from: "@comp-markdown/mermaid@0"
      names: [mermaid]
title: Elm Street Garden
issue: November 2026
rotation: [Beans, Cabbages, Roots, Potatoes]
---

# {{md.doc.title}}

News from the plots, {{md.doc.issue}}.

## Where your scraps go

The green bin by the gate takes your kitchen scraps all winter. This is
their way back to the beds:

```mermaid {theme=forest alt="Kitchen scraps go in the green bin, then on
the compost heap; turned on work days, the compost goes on the beds in
spring"}
graph LR
  Scraps[Kitchen scraps]   Scraps[Kitchen scraps] --> Bin[Green bin by the
gate]
  Bin   Bin --> Heap[Compost heap]
  Heap   Heap -->|turned on work days| Compost
  Compost   Compost -->|in spring| Beds[The beds]
```
````

```
No meat, no bread, nothing cooked: they bring rats.

## Next year's beds

:: mermaid[Beans, then cabbages, then roots, then potatoes, then beans
again] {code=("graph LR; " + md.doc.rotation.join("  +
md.doc.rotation.join(" --> ") + " " --> " + md.doc.rotation[0])}
```

Try it in the playground. Change `theme=forest` to `theme=dark`, or add a step to the compost's way, and the diagram follows.

A code block, drawn

The compost's diagram is written in Mermaid, a language for diagrams that exists outside CompMD. You write it as you would in any Markdown file: three backticks, the language, the diagram's text, three backticks.

What makes it a drawing is the import. The `mermaid` component the newsletter imports for its leaf in chapter 16 also says, in its pack, that it draws the code blocks of the languages `mermaid` and `mmd`. Such a component is a *code renderer*. Once it is imported, every block of those languages in the document is drawn by it; nothing else in the document changes. Without the import, a ```` ```mermaid ```` block is a code block like any other, and no diagnostic says so, since most code blocks are just code.

A code block shows its text as you wrote it. `{{md.doc.title}}` inside it stays `{{md.doc.title}}`, as in any code block (chapter 4): a block's diagram is the same for every reader.

Params on the block

Params go on the block's first line, in braces after the language, written as a component's are (chapter 3):

Param	Takes
`theme`	the Mermaid pack's own param: `auto`, the default, follows the page, Mermaid's dark theme on a dark page; `default`, `dark`, `neutral` and `forest` are Mermaid's themes
`alt`	the drawing's text alternative, below; every drawn block takes it

Param	Takes
`width`, `padding`, `border`, …	the common params of chapter 3, for the box the drawing is in: `width=50%`
`.name`, `#name`	a class and an id, as on any component

The block's text is the component's `code` param, so `code` can't also be given in the braces: `code=…` there is `F017`, and ignored. A value the param doesn't take is `F009`, which says what it does take: write `theme=frest`, and the playground offers `forest`.

CompMD reads the braces only when a code renderer draws the block. On a block nothing draws, such as a ```` ```sh ```` block, the line stays as you wrote it, so what other tools put there is left alone.

`alt`, the text alternative

A drawing can't be read aloud. `alt` is what a screen reader says in its place, as a leaf's fallback is (chapter 3): write in it what the diagram shows, in a sentence. A drawn block without `alt` is `W101`, at its first line.

The diagram's text is no help here: `Bin --> Heap` means something to Mermaid, not to a listener. Write `alt` as you would tell someone what the picture shows.

A block or a leaf

The rotation's diagram, from chapter 16, is the same `mermaid` component as a leaf. Which to write depends on where the diagram's text comes from:

The diagram's text is	Write	Because
written out, the same for every reader	a code block	a reader of the plain file sees the diagram's source, and GitHub draws it
made from the frontmatter, data or inputs	a leaf, with the text as its `code` param	a code block is never filled in from values; an expression can be

The compost's way doesn't change, so it is a block. The rotation's beds come from the frontmatter, and its diagram grows a step when the list does, so it is a leaf. One import gives both: the component can't tell them apart, and draws them the same way. The leaf's fallback in brackets is its text alternative, as `alt` is the block's.

What other readers see

Shown by	The block	The leaf
an app with the pack	the diagram	the diagram
the same app while the pack arrives (chapter 16)	the code block, then the diagram	the fallback, then the diagram
an app that can't get the pack	the code block	the fallback
a plain Markdown viewer	the code block	the line, with its fallback in it
GitHub	a diagram GitHub draws itself from ```` ```mermaid ````	the line, with its fallback in it

A block's fallback is the code block itself: where it can't be drawn, it shows its text, as any Markdown viewer shows a code block. GitHub knows Mermaid but not CompMD, so it draws ```` ```mermaid ```` blocks in its own way, without the params.

When it isn't drawn

- **The pack can't be had.** The import is `F026`, which says why (chapter 16). The document still shows: the blocks stay code blocks, and the leaves show their fallbacks.
- **Mermaid can't read the diagram,** say a line ends in `-->` with nothing after it. The block shows its code, and the app reports what Mermaid found as `W118`; the playground shows it on the block. When the text is fixed, the diagram draws again.
- **Two packs draw the same language.** If two components you import both draw ```` ```mermaid ````, a ```` ```mermaid ```` block is `F023`, which names both. Import only one of them by name, or list it explicitly while the other comes with `names: "*"`.

Giving a code renderer another name with `as` (chapter 16) renames the leaf only: `:: diagram[…] {…}`. Blocks are chosen by their language, so ```` ```mermaid ```` blocks are still drawn.

A code renderer can also be written in CompMD itself, as a definition whose signature says which language it renders; the spec's section on code renderers has the details.

Next: components in code, in brief, for what a component written in JSX, Svelte or JavaScript looks like to the document that uses it.

18. Components in code, in brief

Some components are written in code: in JSX for React, in Svelte, or in plain JavaScript. A pack's diagram is one; you may also get one from a friend who programs, or write one yourself. This chapter shows what such a component looks like to the document that uses it: where its code goes, how the document knows its params, what shows when it can't run, and what the sandbox lets it do. The newsletter gets a bar for each crop of the harvest, from a file, and a map of the plots, from the document itself. Writing the code is the Developer's Guide's subject.

The newsletter, with components in code

```
---
md:
  version: 1
  import:
    - from: ./harvest.jsx
      names: [harvest]
title: Elm Street Garden
issue: November 2026
harvest:
  - {crop: Beans, kilos: 42}
  - {crop: Leeks, kilos: 25}
  - {crop: Kale, kilos: 18}
plots:
  - {number: 1, gardener: Maya Okafor}
  - {number: 2, gardener: Tomás Reyes}
  - {number: 3, gardener: ""}
  - {number: 4, gardener: Priya Nair}
  - {number: 5, gardener: ""}
---

# {{md.doc.title}}

News from the plots, {{md.doc.issue}}.

## This year's harvest
```

````
:: harvest[Beans 42 kg, leeks 25 kg, kale 18 kg] {rows=(md.doc.harvest)
unit=kg}

## The plots

:: plot-map[Plots 1, 2 and 4 are taken; plots 3 and 5 are free] {plots=
(md.doc.plots)}

```compmd component=harvest signature
params:
  rows: array
  unit: string = kg
```

```svelte component=plot-map runtime=svelte@5
<script>
  let { plots, compmd } = $props();
</script>

<div role="img" aria-label={compmd.fallback.text} style="display: flex;
gap: 0.5rem">
  {#each plots as plot}
    <span style="padding: 0.5rem 0.75rem; border: 1px solid; opacity:
{plot.gardener ? 1 : 0.4}">{plot.number}</span>
  {/each}
</div>
```

```compmd component=plot-map signature
params:
  plots: array
```
````

Try it in the playground. Give plot 5 a gardener, and its square on the map no longer looks faded.

The harvest's bars are drawn by a React component in a file beside the document:

```
export default function Harvest({ rows, unit, compmd }) {
  const most = Math.max(...rows.map((row) => row.kilos));
  return (
    <figure role="img" aria-label={compmd.fallback.text} style={{ margin: 0
}}>
      {rows.map((row) => (
```

```
        <div key={row.crop} style={{ display: "flex", gap: "0.5rem",
alignItems: "center" }}>
          <span style={{ width: "4rem" }}>{row.crop}</span>
          <span style={{ width: `${(row.kilos / most) * 60}%`, height:
"0.75rem", background: "currentColor" }} />
          <span>
            {row.kilos} {unit}
          </span>
        </div>
      ))}
    </figure>
  );
}
```

You don't need to read the code. What matters to the document is in the document: each component's name, its params, and its fallback.

The same as any component

To the text that uses it, a component in code is written as every other: `:: harvest[…] {…}` is a leaf with a fallback and params, as `:: mermaid[…] {…}` is in chapter 16. Its params can be expressions (chapter 4), it takes the common params for its box (chapter 3), and it can sit in an `each` or an `if` (chapter 5). A component in code can be a container too, with slots, when its code places them.

Where its code lives is the only difference, and there are two places: a file the document imports, or a fence in the document itself.

A component in a file

```
md:
  import:
    - from: ./harvest.jsx
      names: [harvest]
```

An import whose `from` is a file of code, beside the document or in a folder below it, gives one component, named after the file: `./harvest.jsx` gives `harvest`, and `./PlotMap.svelte` would give `PlotMap`. Names aren't changed between cases; give it another with `as`, as for a pack (chapter 16).

The file's ending says its language:

Ending	Written in	Runs on
`.jsx`, `.tsx`	JSX or TSX, for React	React 19
`.svelte`	Svelte	Svelte 5
`.js`, `.ts`	plain JavaScript or TypeScript	nothing more: plain code

A file keeps the code out of the text, so a reader of the plain file, on GitHub or anywhere else, sees only the leaf's line with its fallback.

A component in a fence

````
```svelte component=plot-map runtime=svelte@5
…the component's code…
```
````

A fenced code block whose first word is a language of code, followed by `component=` and a name, defines a component in the document. The page doesn't show the fence; a plain Markdown viewer shows it as code. The fence can go anywhere in the document, at its top level, and the name works everywhere in it.

`runtime=` says which framework the code is written for, and which version:

Language	`runtime`	When left out
`jsx`, `tsx`	`react@19` or `preact@10`	`react@19`
`svelte`	`svelte@5`	`svelte@5`
`js`, `ts`	none: plain code takes no `runtime`	

Writing `runtime` keeps the component meaning the same when a later version of CompMD moves the default to a new version of the framework. A runtime the app doesn't have, such as `react@18`, or any `runtime` on a `js` or `ts` fence, is `W102`: the component shows its fallback.

A fence in the document is handy while you try a component out, or for one only this document uses. One used by several documents is better in a file, or in a pack.

Its signature

CompMD can't read the code to learn which params a component takes, so a *signature fence* says it, after the language `compmd` and the word `signature`:

````
```compmd component=harvest signature
params:
  rows: array
  unit: string = kg
```
````

It lists the params with their types and defaults, and its slots, the same way as the signature of a definition written in CompMD (Part IV). A signature fence attaches to the component of its name, whether the code is in a fence of the document or in a file it imports. The page doesn't show it.

With a signature, the component's uses are checked like any other's: a param it doesn't take is `F008`, a value of the wrong type `F009`, a required param left out `F010`, and a default fills in what you leave out (`unit` is `kg` unless you say otherwise). Without one, the params go through unchecked, so a misspelt name is never caught; for a component in a fence, that is `W105`, which a signature fence answers. A pack always has signatures for its components, which is why the playground can say what a pack's component takes.

Its fallback

A component in code needs something to run it: an app that shows CompMD in a browser, with the right framework. Wherever that isn't there, the leaf's fallback shows instead:

Where	What shows
a plain Markdown viewer, GitHub	the leaf's line, with its fallback
an app without the framework or its version (`W102`)	the fallback
an app that makes something other than a web page, such as a PDF (`W102`)	the fallback
an app where the code fails to draw (`W118`) or can't load (`W121`)	the fallback
an app that runs it	the component

So the fallback should say what the component shows, as a sentence: *Beans 42 kg, leeks 25 kg, kale 18 kg* tells a reader everything the bars do. A component that draws a picture also gets the fallback as text for screen readers, as both components above use it.

What code in a document can do

A component in code runs in a *sandbox*, as a pack's does (chapter 16). How much it may do is its *trust level*, which only the app that shows the document sets, never the document:

Level	Reaches	For
strict sandbox	only its params, its slots and its own box	the default: code in documents, and every pack the app gives no other level
permissive sandbox	the same, plus the internet addresses the app lists for its pack, and what the app gives it	packs the app chooses, such as a map that loads its tiles
trusted	what the app's own code reaches	components built into the app itself

Code you write in a document, in a fence or a file beside it, always runs in the strict sandbox. For a writer, that means:

- **It sees only what you give it.** Its params and its slots, nothing else of the page or of the reader. Give it the values it needs as params, as the newsletter gives `harvest` its rows.
- **It has no internet.** It can't fetch a forecast or a picture. Data it needs comes through the document's data (Part II), into a param.
- **It can't open links or windows on its own.** The app may offer to open a link it shows, after showing the reader where it goes.
- **It can't harm the reader.** The worst it can do is draw badly in its own box, so a document with code in it is safe to open from anyone.

Sharing a component in code

A document that offers its components to others lists them under `md.export`, as a pack does. A component in code that is exported needs a signature fence, `F029` otherwise, since documents that import it must be checked where its code may not run. A component in a fence that is exported should also name its `runtime`: without one, it is `W117`, since its meaning would change when the default moves. Plain JavaScript takes no `runtime`, and has no `W117`.

Writing one

The code itself is the Developer's Guide's: what a component is given, and how it draws.

- Components and trust: the trust levels, and who decides them.
- Your app's own components: a component in React or Svelte, and its signature.
- Your first pack and signatures and params: a pack of components, with their types, defaults and slots.
- React, Svelte and libraries: the frameworks, and the libraries a component may use.
- The sandbox, closer: how the sandbox is built.

Next: the appendices, starting with the syntax at a glance, everything a writer types in one place.

19. The syntax at a glance

Everything a writer types in CompMD, on a few pages: the frontmatter, the three forms of a component, params, slots, values and expressions, `if` and `each`, definitions, imports and exports, and code blocks a pack draws. Each table links to the chapter that teaches it, and the spec has every rule.

The frontmatter

Everything CompMD reads is under one key, `md`. Every other key is yours, read as `md.doc.<key>` (chapter 4).

Key under `md`	Holds	Example
`version`	CompMD's version; `1` when left out	`version: 1`
`import`	the components the document uses from packs and files (below)	`- from: "@comp-markdown/mermaid@0"`
`export`	the components the document offers others (below)	`names: [notice]`
`data`	data from files and addresses, read as `md.data.<as>` (chapter 6)	`- from: ./plots.csv`
`inputs`	values the reader changes, read as `md.input.<name>` (chapter 7)	`size: small | large = small`
`lang`	the document's language, which formatting follows; `en` when left out	`lang: de`
`dir`	its text direction, `ltr`, `rtl` or `auto`; from `lang` when left out	`dir: rtl`
`tz`	the time zone of `md.today()` and `md.now()`	`tz: Europe/Berlin`
`syntax`	extra syntax the document asks for: `[wiki-links]` for `[[Page]]`	`syntax: [wiki-links]`

YAML to remember. `true` and `false` are the only yes and no; a date such as `2026-11-07` is text, which date functions read; quote what would read as a number, `"1.10"`, or as something else, `"9:30"`. No anchors (`&a`), aliases (`*a`), tags (`!!str`) or repeated keys (`F033`).

A data entry

Key	Holds
`from`	`./file.csv`, `.tsv`, `.yaml`, `.yml`, `.json`, `.comp.md` (its frontmatter), a file in an imported pack (`@scope/pack/file`), an `https://` address, or `host:` data the app gives; may hold `{{md.input.…}}` and `{{md.doc.…}}`
`as`	its name: `as: plots` is `md.data.plots`
`schema`	the type of each field of a list of records: `{plot: number, gardener: string}`; `{from: "Plot no.", type: number}` renames a column
`cache`	for an address: how long a copy is kept, `30s`, `10m`, `1h`, `1d`
`fallback`	for an address or `host:`: a local file to use when it fails
`delimiter`	for a CSV file: the character between fields, `,` by default

An input

`<name>: <type> = <default>`, every input with a default: `start: date = (md.today())`, `size: small | large = small`, `show_all: boolean = false`. An input's name is one word, or words joined by `_`.

Components

The number of colons tells the form (chapter 3):

Form	Written	Notes
Container	`::: name {params}`, a body, `::: /name`	the closer repeats the name; containers nest
Leaf	`:: name[fallback] {params}`	one line; the fallback is what shows where the component can't
Inline	`:name[fallback]{params}`	inside text; no space before `{`, and the brackets are needed

Form	Written	Notes
Continuation	`::: + params`	more params for the line above it, with no blank line between
Slot	`:: #name`	starts a named part of the container it is in

- Put a blank line before and after a block component's lines, so plain viewers keep the text around them as you wrote it (`W108`).
- A backslash makes a line or an inline component text: `\::: box`, `\:value[x]`.
- Component lines work in list items and quotes, indented as the item's text is; not in code or HTML.

Built in

Component	Does	Chapter
`box`	a box, with any of the common params	3
`stack`, `row`, `grid`	blocks top to bottom, side by side, or in a grid	9
`if`, `each`	shows a part when true; repeats a part for each item	5
`value`	computed text: `:value[fallback]{of=(…)}`	4
`date-input`, `number-input`, `text-input`, `select-input`, `toggle-input`	lets the reader change an input: `:: number-input[Plots: 4] {bind=plots}`	7

Params

Written	Is
`title="Plot 4"`	text in quotes; may hold spaces, `{{…}}`, and `\"` for a quote
`padding=m`	a bare value: a word or number, read as the param's type says

Written	Is
`count=3`	the number 3 where the param takes a number, the text `3` where it takes text
`scroll`	a flag: `true`
`tags=["a","b"]`, `meta={"id":1}`	a list or a record, written as JSON on one line
`background=(soon ? "accent" : "subtle")`	an expression, computed (chapter 4)
`.notice`	a class (may repeat)
`#greeting`	the id, for links: `[the greeting](#greeting)`
`md:padding=l`	from the width `sm` (480 px), `md` (768 px) or `lg` (1024 px) up (chapter 10)

- Params are separated by spaces. Each may be given once (`F017`), counting continuation lines.
- A bare value can't hold `(`, `)`, `[`, `]`, `{`, `}`, spaces or quotes: quote it, `"repeat(3, 1fr)"`, or make it an expression. `title=` with nothing after it is `F011`; `title=""` is empty text.
- `{{…}}` works in params only inside quotes: `title="{{md.doc.title}}"`, or `title=(md.doc.title)`.
- Every component takes `class`, `id`, `lang` and `dir`; every block takes the common params of its box (chapter 3): `width`, `height`, `min-width`, `max-width`, `padding` and its sides, `border`, `border-radius`, `background`, `grid-area`, `order`, `align-self`, `justify-self`, `aspect-ratio`, the `inset-*` positions, `scroll`, `fill`, `sticky`, `show` and `hide`.

Slots and template slots

Written	Is	Chapter
content before the first slot line	the default slot	13
`:: #else`, `:: #footer`	a named slot, up to the next slot line or the closer; once per container	13
`{{md.slot}}`, `{{md.slot.footer}}`	in a definition, where a slot's content goes	13

Written	Is	Chapter
`::: each {in=(…) as=job}` … `{{job}}`	a template slot: the body repeats once per item, the item named by `as`	14

A slot line starts at the container's own level, never indented under a list item in it.

Values and expressions

Written	Is	Chapter
`{{md.doc.issue}}`	a value from the frontmatter, inserted as text	4
`{{md.data.fees.small}}`, `{{md.input.size}}`	a value from data, and an input's value	6, 7
`{{md.lang}}`, `{{md.version}}`, `{{md.tz}}`	the document's language, version and time zone	4
`{{job}}`, `{{md.loop.number}}`	inside `each`: the item, and where the list is	5
`{{title}}`	inside a definition: a param	12
`\{{`	two braces as text	4
`:value[$25.00]{of=(md.currency(25, "USD"))}`	computed text, with a fallback	4
`(…)` in a param	an expression in CEL	4

`{{…}}` takes a path only: names joined by dots, a param named with `-` written with `_`. It works in text, headings, table cells, link addresses and quoted params, never in code. A list, or a record without a `name`, can't be inserted (`F024`).

In expressions: `+ - * /`, `== != < <= > >=`, `&& || !`, `a ? b : c`, text in double quotes, `string(…)`, `int(…)`, `double(…)`, `size(…)`, `x in list`, and the list macros `filter`, `map`, `exists`, `all`, `exists_one` (chapter 5). For text: `contains`, `startsWith`, `endsWith`, `lowerAscii`, `upperAscii`, `replace`, `split`, `join`, `substring`, `trim`, `charAt`, `indexOf`, `lastIndexOf`.

CompMD's functions	
Formatting	`md.date(d, style)`, `md.currency(n, code)`, `md.number(n, digits)`, `md.plural(n, one, other)`, `md.list(list)`
Dates	`md.today()`, `md.now()`, `md.addDays(d, n)`, `md.daysBetween(a, b)`, `md.startOf(d, unit)`, `md.dateOf(t)`
Lists	`md.sort`, `md.sortBy(list, field)`, `md.reverse`, `md.slice(list, start, end)`, `md.sum`, `md.min`, `md.max`, `md.unique`, `md.groupBy(list, field)`

Chapter 8 has each one, with examples.

`if` and `each`

	Params	Slots
`if`	`when`, true or false	the default, shown when true; `else`, shown when false
`each`	`in`, the list; `as`, the item's name; `join`: `none` (the default), `list` or `inline`; `sep` and `last`, between items with `join=inline`	the default, once per item; `before`, above the items when there are any; `empty`, when there are none

`md.loop` inside `each`: `index` (from 0), `number` (from 1), `count`, `first`, `last`.

Definitions

A component written in CompMD (chapter 12) is a fence: the language `compmd`, `component=` and its name, params for its box, a signature, `---`, and a body.

````
```compmd component=notice {padding=m border-radius=m background=accent}
params:
  title: string
  tone: info | warn = info
slots: [default]
---
**{{title}}** {{md.slot}}
```
````

In a signature	Is	Chapter
`title: string`	a required param	15
`note: string?`	an optional one, with no value when left out	15
`tone: info | warn = info`	a choice of words, with a default	15
`{type: object, default: {a: 1}, doc: …}`	the long form, for defaults that don't fit on a line	15
`slots: [default, footer]`	the slots it takes; without `slots`, it takes no body	13
`slots: {default: {each: rows, as: as}}`	a template slot over the param `rows`, its item named by the param `as`	14
`headings: absolute`	headings in the body keep their level, instead of fitting where it is used	12

Types: `string`, `number`, `boolean`, `date`, `array`, `object`, `name` (a name the caller picks, as `as` is), the styling types `length`, `space`, `color`, `tracks`, `areas`, `surface`, `border` and `radius`, a choice `a | b | c`, and `input<type>` for a param bound to an input.

A component in code (chapter 18) is a fence in its language, `jsx`, `tsx`, `svelte`, `js` or `ts`, with `component=` and an optional `runtime=` (`react@19`, `preact@10`, `svelte@5`), and its signature is a fence of its own: ```` ```compmd component=harvest signature ````.

Imports and exports

```
md:
  import:
    - from: "@comp-markdown/mermaid@0"
      names: [mermaid]
    - from: ./notices.comp.md
      names:
        - name: notice
          as: alert
    - from: ./harvest.jsx
      names: [harvest]
  export:
    pack: "@elm-street/garden"
    version: "1.0.0"
    names: [notice]
```

Key	Holds	Chapter
`from`	a pack, `@scope/name@version`, or a file: `./x.comp.md`, `.jsx`, `.tsx`, `.svelte`, `.js`, `.ts`	16, 18
the version	`@1`: the newest 1.x; `@^0.1.1`: 0.1.1 or a newer 0.1; `@0.1.1`: exactly that	16
`integrity`	the pack's fingerprint, with an exact version: `sha512-…`	16
`names`	the components used: `[a, b]`, `{name: a, as: b}` to rename, or `"*"` for all	16
`export.names`	the components offered: their names, `{name, as}`, or `"*"` for every definition	12
`export.pack`, `export.version`	the name and version a published pack has	12

`if`, `each`, `value` and `box` can't be defined, imported or renamed to (`F013`).

Code blocks a pack draws

Written	Is
`` ```mermaid ```` … ```` ``` ````	drawn by an imported code renderer for `mermaid`; otherwise a code block
`` ```mermaid {theme=dark alt="…"} ````	params after the language, in braces; `alt` is the text for screen readers
`:: mermaid[fallback] {code=(…)}`	the same component as a leaf, for a diagram made from values

A code block's text is never filled in from values. Chapter 17 has the rest.

Next: the diagnostics and their fixes.

20. The diagnostics and their fixes

When something in a document isn't right, CompMD says so with a *diagnostic*: a code, the place, and a message that says what it found. This appendix lists the diagnostics a writer meets, grouped by what you were doing when it came up, each with what it means and how to fix it. The spec's catalogue has every code, and the Developer's Guide's appendix the ones only an app or a pack's author causes.

Reading a diagnostic

- `F` **codes are errors:** the document says something CompMD can't do as written. `W` **codes are warnings:** the document works, but something is probably not what you meant.
- **The page still shows.** In the playground and in editors, CompMD drops the smallest broken piece, a param, a line or a record, shows the rest, and shows a component's fallback where it can't show the component. A tool that publishes may refuse a document with errors, so fix them all.
- **One mistake gives one diagnostic.** A param that is refused doesn't also give a diagnostic at each place it would have been used.
- **Some come with a fix.** For a misspelt name, the message says *did you mean …?*, and the playground offers a button, **Use** and the right name, that makes the change. Those are marked *Fix offered* below.

In the frontmatter

Code	Means	Fix
`F001`	`md.version` names a version this app doesn't read, such as `2`; the document is shown as plain Markdown	write `version: 1`, or leave it out
`W110`	`md.version` is a later minor version, such as `"1.3"`, than the app reads; what it doesn't know gives its own diagnostics	write the version the app reads, or use an app that reads the later one

Code	Means	Fix
`W111`	a key under `md` that CompMD doesn't know, such as `imports`, is ignored; the message names the nearest one. Also a name in `md.syntax` it doesn't know	correct the key: `import`. A misspelt key under `md` is a key that does nothing, so what it held, such as the inputs, is missing too
`F033`	the frontmatter isn't YAML as CompMD reads it: a mistake in the YAML, a key given twice, or an anchor (`&a`), alias (`*a`) or tag (`!!str`); the document is shown as plain Markdown	fix the YAML at the place the message gives; write a repeated value out again instead of an anchor

Writing components

Code	Means	Fix
`F016`	a line starts with `::` but is no component line: `:::box` (no space), `:: /box` (two colons for a closer), `::::` (four colons), or `::: @who/box` (a pack's name in the text); the line is shown as text	write the form the message gives: `::: box`, `::: /box`, `:: name[…]`, `:: #name`, `::: + …`. To write the line as text, start it with `\`
`F003`	a closer names no open container, such as `::: /bxo` for `::: box`; it closes the innermost one, which the message names	correct the closer's name
`F004`	a container isn't closed before the end of the document, or before the closer of a container around it; it runs to there	add its closer, `::: /name`, where its body ends
`F005`	a closer while no container is open; it is ignored	remove it, or add the opener it belongs to
`F032`	a closer in a list item or quote names a container opened outside it; it is ignored	put the closer at the same level as its opener, outside the list item or quote
`F012`	a `::: +` line that doesn't follow an opener, a leaf or another `::: +` line, often because of a blank line between them; it is ignored	move it directly under the line it adds to
`F015`	a slot line, `:: #name`, outside any container; it is ignored	move it into the container, or remove it

Code	Means	Fix
`F007`	a container has the same named slot twice; the second one's content joins the first	merge the two parts under one slot line
`F018`	a component that takes no body is written as a container, such as `::: mermaid` … `::: /mermaid`; the body shows after it, as ordinary content	write it as a leaf, `:: mermaid[…] {…}`
`W108`	a component line directly after a line of a paragraph; a plain Markdown viewer reads it as part of the paragraph	put a blank line before it
`W116`	a line starting with `::` inside an HTML block, which makes it part of the HTML	put a blank line between the HTML and the component line
`W107`	an inline component followed by a space, then braces: `:value[x] {of=(…)}`; the braces are shown as text	remove the space: `:value[x] {of=(…)}`
`W101`	a leaf without fallback text, `:: chart {…}`, which shows nothing in a plain viewer and has no text for screen readers; or a code block a pack draws without `alt`	write what it shows in the brackets, `:: chart[Sign-ups rose to 31] {…}`, or add `alt="…"` after the block's language (chapter 17)

Writing params

Code	Means	Fix
`F008`	a param the component doesn't take, such as `colour=red`; it is ignored. Also a breakpoint that isn't `sm`, `md` or `lg`, or a breakpoint on a param that takes none, such as `md:show`	*Fix offered* for a misspelt param: `pading` becomes `padding`. Otherwise remove it; a component's signature, or its pack's README, says which params it takes
`F009`	a value the param doesn't take, such as `border=subtel`, or `count=many` for a number; the param is treated as absent, so its default is used	*Fix offered* for a misspelt word or colour: `subtle`. Otherwise write a value of the kind the message names
`F010`	a required param is left out; the component shows its fallback	add it, as the message names it

Code	Means	Fix
`F011`	a param list CompMD can't read: a quote or bracket that doesn't close, a value with `(`, `{` or a space that isn't quoted, such as `grid-template-columns=repeat(3, 1fr)`, or `=` with nothing after it. On a block line, the params before it are kept; in an inline component, none are, and the list is shown as text	close the quote or bracket; quote the value, `"repeat(3, 1fr)"`, or make it an expression, `title=(md.doc.title)`; write `title=""` for empty text
`F017`	a param given twice, counting `::: +` lines, or `#name` and `id=` together, or `code` in the braces of a code block a pack draws; the first is used	remove the second
`F006`	a slot line the component doesn't declare, such as `:: #els` in an `if`; its content joins the default slot	*Fix offered*: `els` becomes `else`
`W114`	a box whose blocks change at a breakpoint gets its width from its content, in a row for example, and so has none	give it a `width` (chapter 10)
`W119`	a box that places blocks by position (`inset-*`) has no height to place them in, at some width	give it a `height`, an `aspect-ratio` or `fill` (chapter 11)
`W120`	a param for a box given to a component that has no box of its own (one defined `display=contents`); it is ignored	remove it, or put it on a `box` around the component

Values and expressions

Code	Means	Fix
`F014`	a path in `{{…}}` that doesn't exist, such as `{{md.doc.isue}}`; it is shown as written. Also prose that mentions double braces	correct the path; to write the braces as text, write `\{{`, or put them in a code span
`W106`	`{{` followed by something that isn't a path, such as `{{ 1 + 2 }}`; it is shown as text	for a computed value, use `:value[…]{of=(…)}` (chapter 4); for the braces as text, `\{{`

Code	Means	Fix
`F024`	a mistake in an expression: a misspelt name, a value of the wrong type (`"Plot " + 4`), or one that fails when computed; the message points to the place in the expression. Also a list, or a record without a `name`, put in `{{…}}`	correct the expression: `"Plot " + string(4)`; write a list with `md.list(…)` or `each` (chapter 5)
`W109`	a link or image address made with `{{…}}` that is neither a relative address nor one starting with `https:`, `http:` or `mailto:`; the text is shown without the link	check the value: an address from the frontmatter or data should start with `https://`
`F013`	a name declared twice in the document; or `if`, `each`, `value`, `box` or `md` defined, imported or renamed to; or a word expressions keep for themselves (`in`, `as`, `list`, `string`, …) as an `as` name, an input or a definition's param; or two params whose names differ only in `-` and `_`. The first declaration wins	choose another name: `as=job`, not `as=in`
`F025`	an `each` whose body doesn't have the shape its `join` needs: list items for `join=list`, one paragraph for `join=inline`; the items are added one after another	make the body one list item, or one paragraph, or remove `join`
`W103`	an `each` whose body is one list item, without `join`; each item becomes a list of its own	add `join=list`
`F034`	the page would have more than the app allows, at least 100,000 parts, usually from `each` inside `each` inside `each`; the `each` that crossed the limit shows its body once	filter the lists first, or show fewer levels

Data and inputs

Code	Means	Fix
`F026`	data that can't be loaded: a file that isn't there, an address that doesn't answer, a file CompMD can't read, a `host:` source the app doesn't give, a path that leaves the document's folder (`../`), or a CSV header with an empty or repeated name. The data is its `fallback` file if it has one, otherwise empty, so an `each` over it shows its `empty` slot	correct the path or the file at the place the message gives; give a remote source a `fallback`

Code	Means	Fix
`F027`	a record that doesn't match the `schema`, such as `twelve` where a number is declared, or a row of a CSV file with more or fewer cells than the header; the record is dropped	correct the record, or the schema's type; declare a column that may be empty as optional: `number?`
`F028`	data from an address the app isn't set up to fetch from; the data is its `fallback`, or empty	ask the app's owner to allow that address, or give the entry a `fallback`. An app that asks its readers shows the data once the reader allows it
`F031`	a `from` fills in something it may not: anything but `md.input` and `md.doc`, a value in the address's `https://` and host, or an input a component in code can change	fill in only the path or query of the address, from inputs and the frontmatter
`F030`	an input component's `bind` names an input that isn't declared under `md.inputs`, or one of a type it can't edit, such as a `number-input` bound to a choice of words	declare the input, correct its name, or use the input component for its type (chapter 7)

Definitions and components of your own

Code	Means	Fix
`F002`	a component name the document doesn't know, such as `:: bxo`; it shows its fallback. Also a signature fence with no component of its name to attach to	*Fix offered* for a misspelt name: `box`. Otherwise import the component, or define it
`W112`	a definition or import with the name of a built-in component, such as `row`, which this document's `row` now means	keep it, if that is what you want, or choose another name
`F019`	a definition that uses itself, directly or through others, such as a `card` whose body has a `card`; the use that closes the circle shows its fallback	remove the circle: a definition can't contain itself
`F036`	a signature fence with params in braces after it; they are ignored	put them on the component's own fence, which styles its box

Code	Means	Fix
`F037`	a type in a signature, an input or a schema that CompMD doesn't have, such as `numbr`; that param takes any value	*Fix offered*: `number`
`W105`	a component in code, in a fence, without a signature fence; its params go through unchecked	add a signature fence (chapter 18)
`W102`	a component in code that this app can't run: a `runtime` it doesn't have, such as `react@18`, any `runtime` on a `js` or `ts` fence, or an app that makes no web page; it shows its fallback	use a runtime the app has, or leave `runtime` out
`F029`	a component in code that the document exports, without a signature fence, which the documents that import it need, since they may be shown where its code can't run	add a signature fence
`W117`	a component in code that the document exports, without `runtime=`	add `runtime=`: `react@19`, `preact@10` or `svelte@5`

Packs and imports

Code	Means	Fix
`F026`	an import that can't be had: no pack of that name, no version that matches, a pack the app doesn't allow or that is too large, a file that isn't there, or a `from` that isn't a pack's name (`@who/what`) or a path. The import is dropped, and its components show their fallbacks	correct the name or the version (chapter 16); the message says which of these it is
`F021`	an imported name its source doesn't offer; a file of code offers one component, named after the file	correct the name, or see the pack's README for what it offers
`F020`	documents that import each other in a circle; the import that closes it is dropped	remove one of the imports
`F023`	a name two `names: "*"` imports both give, or one of them and a built-in component, is used; or a language two code renderers both draw. It shows its fallback	list the name you mean in one import's `names`

Code	Means	Fix
`F009`	`integrity` on an import whose version isn't exact, such as `@0`; it is ignored	pin an exact version, `@0.1.1`, or remove `integrity`
`F035`	the pack doesn't match the fingerprint in `integrity`; the import is dropped	check the version and its fingerprint; a tool should write it (chapter 16)
`W113`	a component from a pack the app itself provides, which other apps don't have; there, it shows its fallback	nothing, for a document only that app shows; for others, import a published pack instead
`W118`	a component in code failed to draw what it was given, such as a diagram Mermaid can't read; a leaf shows its fallback, and a code block its code	correct its params, or the block's text; the message says what the component found
`W121`	a component in code can't load: it needs a library or framework the app doesn't have, or its code doesn't compile; it shows its fallback	use the component in an app that has what it needs, or tell the component's author

Back to the contents