What is Rheo? ... 4
Who should use Rheo? ... 4
Why Rheo? ... 4
  Why do we need Rheo? ... 4
  The philosophy of Rheo ... 5
Installation ... 6
Firing up ... 7
Scaling up ... 8
Adding a config ... 9
Usage ... 10
What gets created ... 10
Compiling the scaffolded project ... 10
Extending the template ... 11
Frequently Asked Questions ... 11
  What is the difference between Typst and Rheo? ... 11
  How do I read EPUBs on my system? ... 12
  Who maintains Rheo? ... 12
  Does Rheo work with CI/CD and static site hosting? ... 12
  Can I contribute to Rheo? ... 12
Core ... 12
  Format plugins ... 12
  An open architecture ... 12
Rheo.toml ... 13
  Per-page footnotes ... 13
  Automatic package detection ... 14
  Fonts ... 14
Relative linking ... 14
  Handle assignment ... 14
  Escape form ... 15
  Migrating from older Rheo versions ... 15
`rheo-context` ... 15
  Fields ... 16
  Reading vertebra metadata ... 17
  Example: building a blog feed ... 17
Migrating projects ... 18
  Usage ... 18
  What migrate rewrites ... 18
  Removals in 0.6.0 ... 19
Build directory ... 20
Configuration ... 20
  CLI flag ... 20
  `rheo.toml` ... 20
Content directory ... 20
Configuration ... 20
  `rheo.toml` ... 20
Spines ... 21

Directory-scan default 21
`[spine] exclude` 21
`[spine] include` 21
`[[spine.section]]` 21
Per-format overrides 22
Assets 22
Copying files to all formats 22
Format-specific copies 22
Format-specific assets 23
Relation to Typst's bundle format 23
Marrow 23
Crafting marrow 23
Reading across the spine 24
Using beacons 24
Head control 24
`.rheo/head.html` for every page 25
Control assets 25
PDF 25
One PDF from many files 25
EPUB 25
How Rheo builds an EPUB 26
Detecting the EPUB target 27
HTML 27
From one file to a whole site 27
HTML Assets 28
CSS 28
Javascript 28
Multiple asset blocks 28
Packages 29
Using packages 29
Turning off automatic detection 29
Creating a Rheo-compatible package 29
Requiring a minimum Rheo version 30
Feeds 30
Importing the package 30
Configuring a feed 31
Feed fields 31
Multiple feeds and formats 31
Where entries come from 32
Entry fields 33
Sourcing from another package 34
Keeping page chrome out of your entries 35
Required fields and silent drops 35
Migrating from the retired `[html]` feed keys 35
Slides 36
Importing the package 36
Defining slides 36
Applying the template 37
Configuring the spine 37
PDF script output 38

Customising the RevealJS CSS 38
RevealJS CSS variables 38
Common overrides 38
Keeping your original paper styling 39
Bibliography 39

What is Rheo?

The simple answer is that Rheo (*ree-oh*) is document infrastructure that makes Typst a viable foundation for multi-format publishing. The less simple answer is that Rheo is a typesetting and static site engine that sits between the Typst language and the formats readers use — PDF, HTML, and EPUB — providing the compilation, linking, and asset management needed to produce all three from a single set of source files. This guide explains both *how* to use Rheo, and *why* it might be for you.

Rheo allows you to produce a website, a fixed-size document, and an adaptive document from the same Typst source. It allows you to do something similar to LaTeX—except that Typst is *much* simpler to write, and we can produce a greater number of formats with it. The documentation that you are reading now, for example, was typeset with Rheo. As a result, you can read it as:

- HTML - as a website for browsers.
- PDF - as a fixed-size document for printing.
- EPUB - as an adaptive document for e-readers.

Who should use Rheo?

If you write anything as simple as a blog or as complex as a dissertation or monograph in Typst, Rheo enables you to publish it in multiple formats. If you are willing to learn a little bit of syntax, you can turn a piece of writing into a website, an adaptive document, and/or a printable document.

Some of the things you can write and publish with Rheo include:

- A blog
- A paper
- A dissertation
- A book manuscript
- A novel
- A textbook
- Technical documentation

Rheo is for anyone who has ever spent regrettable hours formatting citations, fighting with LaTeX, experiencing the limitations of Markdown, or who wants to benefit from the richer writing experience that Typst makes possible (more on this in the next section). It is for students and teachers, humanists and scientists, bloggers and novelists.

If you have only ever used Microsoft Word to author text, or haven't heard the phrase 'markup language' before, we recommend first familiarizing yourself with Typst via the excellent tutorial. This should give you a good intuition for what Typst is—a markup language similar to but also more powerful than Markdown—and why you might want to use Rheo to typeset your documents.

Why Rheo?

Why do we need Rheo?

Rheo (*ree-oh*) is an open source typesetting and static site engine for Typst. It is a typesetting engine because it produces typeset digital documents such as PDF and EPUB, and a static site engine because it produces websites that don't require communication with a custom backend server, but rather are self-sufficient sets of files that can be natively opened in a browser (static sites). Most static site engines these days employ Markdown, a markup format that is approachable and pretty generic, allowing folks who are not familiar with or otherwise don't want to deal directly with the

required file formats of the web— HTML, CSS, and Javascript— to write blog posts and other content which can then be pumped into a static site.

As useful as it is, Markdown has its ambiguities. For one, there isn't a standardized syntax for citations or footnotes. Though extensions exist that can produce these, they are not supported in the core Markdown syntax, meaning that it's not *really* Markdown and can't be relied upon to work in all contexts that support Markdown. Markdown is great when using hypertext (hyperlinks, images, etc.). It's not so great when it comes to things like tables, figures, and math.

Typst is a markup language that integrates with plain text, like Markdown, making it easy to adopt and joyful to write. Unlike Markdown, however, it is also a Turing-complete programming language with a modern type system, meaning that it is possible (though not necessary) to express sophisticated conditional logic controlling where and how text is rendered. Typst has a concrete and concise syntax for footnotes and citations, and can express visual constructs such as tables, figures, colors, and mathematical formulas. It was developed as a modern alternative to LaTeX, Leslie Lamport's legendary 1980s addition to Donald Knuth's original '78 TeX typesetting system. For the past 40 years, LaTeX has been the most expressive way to produce PDF documents, rendering it the de facto standard for academic and scientific publication. In the past few years, Typst has become the most promising and powerful alternative to LaTeX due to its maintainers' effort to build out a reliable PDF compilation toolchain.

In 2025, Typst added experimental support for HTML compilation. Though there are still many features in Typst that will only produce meaningful output in the PDF toolchain, the HTML toolchain now supports all of the essential features for academic documents in the humanities: text decoration, headings, hyperlinks, footnotes, and citations.[1] This makes it an extremely good replacement for Markdown as a markup language in a static site engine, and so: enter Rheo.

Rheo is a CLI (command line interface) that produces PDF, HTML, and EPUB simultaneously from a folder of Typst documents. It is a static site engine because it can produce a fully valid website: all it needs is a folder containing valid Typst. Rheo also provides mechanisms to combine multiple Typst files into a unified EPUB or PDF, making it a tool that improves the experience of writing books, dissertations, or any other long-form text in Typst. On the other side of the same coin, Rheo allows you to produce an offline version of a website such as a blog written in Typst through its PDF/EPUB toolchain.

Rheo allows you to compile multiple Typst files that link to each other into a single output, adding what is needed (relative linking) in order to make Typst an ideal markup language for writing static sites. Typst is the most elegant and flexible way to typeset PDF documents today; Rheo extends Typst's capabilities, allowing you to additionally typeset EPUBs and generate static sites from the same source.[2] Naturally, this blog post was written in Typst, and this site was made with Rheo. If you're already convinced, feel free to jump ahead to Getting Started to download Rheo on your system and start writing.

The philosophy of Rheo

Rheo is a prefix or combining form in English that originates from the Greek word *rheos* (ῥέος), meaning flow, stream, or current. Rheo flows Typst documents into a number of concurrent output formats in PDF, HTML, and EPUB. But other meanings lurk beneath the surface of this basic idea. Sarah Pourciau has argued that the oceanic is a deep-rooted metaphor in computing, as all computation at some level seeks solid space in a sea of digital noise (Pourciau 2022). From Alan

[1] I qualify this with 'in the humanities' as scientific papers often require tables, figures, and mathematical markup. These features are on Typst's roadmap for html, but are not yet available at time of writing (January 2026).

[2] EPUB is on Typst's roadmap, but is not yet natively supported.

Turing's partial solution to David Hilbert's *Entscheidungsproblem* in the universal machine, to Claude Shannon's information theory, to Leslie Lamport's ordering of events in a distributed system, the key issue at hand is how to carve out clarity from uncertainty and confusion. Writing has played a magisterial role in calming the storm of imprecise thought. Long before computation arrived on the scene, the written word has served as the steward of reason, in the Western world and beyond, from Mesopotamian cuneiform to Twitter. *Nota bene* ('Take note'): that writing can also herald chaos and confusion doesn't invalidate its capacity for spreading sensibility.

Rheo is a tool that facilitates the production and publication of documents following from the original vision of the Internet as a mechanism for lively and reasonably unfettered academic exchange, rather than the densely commercial space of platform capitalism that it has become. It should not be so difficult, given the extraordinary capacities of software and hardware today, to make a piece of writing publicly available in a plain and pleasant format. That there exist digital humanities initiatives measured in months and years to bring books to the web as basic websites is a clear sign that something has gone awry.[3]

Rheo aims to enable the publication of more books, blog posts, and papers without the necessary capitalist ceremony of creating an account on Substack, Medium, or Squarespace. A website without any interactive elements such as forms, online marketplaces, or comments should be simple to set up, as it isn't rocket science in 2025 thanks to all the hard work that folks have put into Internet protocols and web standards. It *should* be simple to create a PDF or EPUB for sharing with colleagues or collaborators—as simple as it is to send an email.

This is the vision of the world to which we at the free computing lab aspire, and in search of which we have built Rheo. Beyond making publishing simpler, we believe that better document infrastructure lays the groundwork for richer ways of *reading*. When documents are produced in structured, portable formats like EPUB and HTML, they become augmentable. Researchers and developers can build reading experiences on top of them, such as contextual definitions, explorable explanations, and interactive figures. (This is exactly what we are trying to do with EPUB through Bene.)

Rheo is the first installment in a larger set of writing tools we aim to build, which will include processes for collaboratively drafting documents, constructing and working with digital libraries, and more.

Installation

The easiest way to install Rheo is using cargo-binstall. Install it, and then run:

```
cargo binstall rheo
```

Alternatively, you can build Rheo from crates.io, the Rust language's package manager. If you don't already have Rust/cargo, you will need to install those first. Open your terminal, and run the following command:

```
cargo install rheo --locked
```

[3] There are many exciting and experimental ways of presenting text and other content in the digital humanities. But we think that it should be easier than it currently is to publish and distribute books digitally, simply and straightforwardly.

Rheo is packaged as a standalone binary, and doesn't require any version of Typst on your system. (Note that even if you already have Typst on your system, Rheo will use its own embedded version of the compiler.) Refer to Rheo's source code for more information and installation options.

Because that embedded compiler is its own thing, Rheo also does its own font discovery rather than deferring to whatever fonts a system-installed Typst might already know about. Alongside whatever's on your system, it picks up a `fonts` directory at your project root automatically; point it elsewhere with the repeatable `--font-dir` flag on `compile` and `watch`, or list directories permanently under `font_dirs` in `rheo.toml`.

Firing up

With Rheo we can produce a static site, a PDF, and an EPUB from a Typst document. Let's create a directory with a single Typst file in it:

```
mkdir project_uno
touch project_uno/index.typ
```

As one of Rheo's outputs is a static site, the landing page will default to one named 'index'. (If this file doesn't exist, Rheo will present you with a basic listing of all the other files in the site.) Let's put some Typst in the `index.typ` file:

`project_uno/index.typ`

```
= Project uno

Project uno is a writing project.
```

Rheo aims to keep out of your way as much as possible, and doesn't require that you add any special syntax or metadata to your files to work. This single Typst file is all we need to get started. Provided you've already installed Rheo on your system, we can compile the project. You can tell Rheo to compile a folder by pointing the `compile` command at it:

```
rheo compile project_uno
```

This command produces a `build` subdirectory inside the base directory which contains a PDF, an EPUB, and a static site (HTML and CSS). Your project folder should now look something like this:

```
.
├── build
│   ├── epub
│   │   └── project_uno.epub
│   ├── html
│   │   ├── index.html
│   │   └── style.css
│   └── pdf
│       └── index.pdf
└── index.typ
```

It's a little tiring to have to run the compile command every time we make a change, though. Let's spin up a development server to see live changes across all output formats as we edit the source:

```
rheo watch project_uno --open
```

The `--open` flag here indicates that we'd like to open the output using our system's default applications. Provided you have an EPUB reader on your system (if you don't, we recommend installing Bene), you should now have a PDF, an EPUB, and a website in front of you. As simple as that!

Scaling up

Let's kill that process (with Ctrl-C). Rheo compiles documents from across your project directory towards EPUB, PDF, and HTML simultaneously, whereas the Typst compiler typically takes just one Typst file and produces one kind of output.[4] Let's add a couple of files to our project and link between them:

`project_uno/about.typ`

```
= About

Project uno is an incredible writing project that will transform the way we
understand the world.
If you want to be involved, see the #link(<contact>)[Contact page].
```

`project_uno/contact.typ`

```
#let email = "myemail@mydomain.com"
= Contact

To learn more about project uno, email me at #link("mailto:" + email)[#email]
```

And let's also link to the two new pages on the index page:

`project_uno/index.typ`

```
= Project uno

Project uno is a writing project.

- #link(<about>)[About]
- #link(<contact>)[Contact]
```

Now let's run Rheo again, but this time let's only build the HTML and EPUB outputs:

```
rheo watch project_uno --html --epub --open
```

Note how the relative links are working across both the HTML and the EPUB. Relative linking is one of the key features in Rheo that enables you to build richer static sites and EPUBs beyond using just Typst. All of Typst's other features such as variables are fair game, too, as Rheo just uses Typst's compiler under the hood.

[4] Typst allows you to break up your projects using modules, but still requires one entrypoint. Rheo, by contrast, enables multiple entrypoint files, corresponding to multiple standalone pages in a static site.

Adding a config

One issue with the EPUB that is currently being produced is that the `index.typ` section shows up last, after `about.typ` and `contact.typ`. This is because Rheo builds a project's spine by scanning `content_dir` and ordering files alphabetically by default, and `index` sorts after `about` and `contact`. This is probably not what we want, as the index page acts as a sort of table of contents in our writing project currently.

Since the spine follows file names, we can fix the order by renaming the files so the alphabetical scan already matches our intended reading order:

```
mv project_uno/index.typ project_uno/00-index.typ
mv project_uno/about.typ project_uno/01-about.typ
mv project_uno/contact.typ project_uno/02-contact.typ
```

Renaming a file changes its handle too — Rheo derives a handle from the file's name — so update the cross-links to match:

`project_uno/00-index.typ`

```
= Project uno

Project uno is a writing project.

- #link(<01-about>)[About]
- #link(<02-contact>)[Contact]
```

`project_uno/01-about.typ`

```
= About

Project uno is an incredible writing project that will transform the way we
understand the world.
If you want to be involved, see the #link(<02-contact>)[Contact page].
```

Now let's add a rheo.toml config at the base of the project directory to give the EPUB and PDF outputs a proper title:

`project_uno/rheo.toml`

```
version = "0.4.0"

[epub.spine]
title = "Project Uno"

[pdf.spine]
title = "Project Uno"
```

Let's run the `watch` command again, this time with all outputs like the first time:

```
rheo watch project_uno --open
```

Great! Renaming the files fixed the order for both the EPUB and the PDF in one go — the PDF output, like EPUB, combines every source file in its spine into a single document, following the same directory-scan order.

Before we run this again, let's also clean the outputs in the build directory:

```
rheo clean project_uno
rheo watch project_uno --open
```

Now we have a fully featured writing project, with nice-looking and orderly outputs in PDF, EPUB, and in HTML!

The `init` command scaffolds a fully functional Rheo project that demonstrates the most common features for Rheo projects that produce HTML, PDF, and/or EPUB.

Usage

```
rheo init my_project
```

Where `my_project` is a path to either a new directory that will be created, or an existing empty directory in which the Rheo project will be scaffolded. (Hidden files and directories like `.git` or `.jj` are ignored, so you can initialize a Rheo project in a fresh repository.)

What gets created

Running `rheo init my_project` produces the following structure:

```
my_project/
├── content
│   ├── about.typ
│   ├── img
│   │   └── header.svg
│   ├── index.typ
│   └── references.bib
├── index.js
├── rheo.toml
└── style.css
```

- **`rheo.toml`** – the project configuration, with a content directory set to `content/`, and spines for both PDF and EPUB.
- **`style.css`** – a custom stylesheet for the HTML output.
- **`index.js`** – a script entrypoint for the HTML output.
- **`content/index.typ`** – the landing page, which defines a reusable `template` show rule and demonstrates conditional rendering for HTML vs PDF.
- **`content/about.typ`** – a second page that imports the template and links back to the index.
- **`content/references.bib`** – a sample bibliography file, referenced from `index.typ` using Typst's `#bibliography` function.
- **`content/img/header.svg`** – a header image used across formats.

Compiling the scaffolded project

Once scaffolded, you can compile or watch the project as usual:

```
rheo compile my_project
```

Or spin up a development server:

```
rheo watch my_project --open
```

Extending the template

The scaffolded `index.typ` includes a `template` function that applies a header across formats using Typst's `target()` function:

```
#let template(current-page: none, doc) = {
  context if target() == "html" {
    html.elem("div", attrs: (class: "header"))[
      #image("img/header.svg")
    ]
    html.elem("hr")
  } else if target() == "paged" {
    image("img/header.svg")
  } else {}
  doc
}
```

Alongside Typst's native `"html"` and `"paged"`, Rheo synthesizes a `"epub"` target so you can branch on EPUB output specifically – see the EPUB page for details.

Other pages in the project import and apply this template with a show rule:

```
#import "index.typ": template
#show: template.with(current-page: "about")
```

This pattern — defining a show rule in one file and importing it in others — is the same one used to build this documentation site. You can extend it to include navigation, footers, or any other shared layout you need.

Frequently Asked Questions

What is the difference between Typst and Rheo?

Typst is a markup/programming language that provides its own toolchain which includes a CLI. You can use the Typst CLI to compile one Typst document to one kind of output file:

```
typst compile source.typ # compile to PDF
typst compile --features html --format html source.typ # compile to HTML
```

Rheo compiles a *project folder* to three outputs—PDF, HTML, and EPUB—concurrently. It allows you to configure how certain source files should be merged (to produce a 'combined' EPUB or PDF file, for example, via spines), and also allows you to enrich certain outputs (such as HTML via custom CSS) with non-Typst content. Rheo supports EPUB natively, which is not currently supported by the upstream Typst CLI (though it is on the roadmap).

Since 0.2.0, Rheo has a plugin architecture, which means that it can be extended to compile to additional formats. We have built Rheo as infrastructure for augmented reading research in Typst – see the Core page for more on how the plugin architecture enables this.

How do I read EPUBs on my system?

Mileage varies greatly on EPUB reading niceness across systems. We have written more about this and the other sharp corners associated with the EPUB format here. If you don't have a good EPUB reading experience currently, we recommend using Bene, an EPUB reading system that we are developing.

Who maintains Rheo?

Rheo is developed by the Free Computing Lab, an academic research consortium that researches the nature of computing freedom. If you're interested to learn more or get involved, you can join our Zulip.

Does Rheo work with CI/CD and static site hosting?

Yes. Since Rheo is distributed as a self-contained binary, it integrates cleanly with CI environments without any additional toolchain setup. Rheo has been tested with GitHub Pages, Cloudflare Pages, and Netlify, and should work with any similar platform.

For a concrete example of a build script that uses Rheo to publish a website, see this build.sh from the Free Computing Lab site.

Can I contribute to Rheo?

Yes! Rheo is written in Rust and developed in public through GitHub. You can track development and submit issues or requests for features through that platform. While in principle we welcome community pull requests, it's best to join our Zulip and ask about it first, to confirm that your work will not go to waste.

Core

Rheo has a plugin architecture. Its core engine embeds the Typst compiler and provides shared infrastructure such as spine resolution, asset management, relative link rewriting, and a development server with live reloading. On top of this core, each output format is implemented as a *format plugin*.

Format plugins

Typst natively supports PDF and, experimentally, HTML. Rheo defines a `FormatPlugin` trait that can be implemented to produce additional formats beyond those that Typst natively supports. For example, Rheo supports EPUB as a format plugin, enabling the production of a standards-compliant EPUB archive from source Typst.

Each format plugin declares:

- A **compile function** specifying how build output files should be produced from the source spine.
- Any format-specific assets it needs – such as the HTML plugin's CSS stylesheet and JavaScript entrypoint.
- Optional template files and configuration that are scaffolded when you run `rheo init`.

The core engine discovers source files, resolves the spine for each format, copies assets, and invokes each plugin's compilation in turn.

An open architecture

Our hope is that this plugin architecture will inspire innovation at the format level. Typst is a remarkably expressive document language, and we believe its utility extends well beyond the formats it natively targets.

We have built Rheo as infrastructure for augmented reading research. Reading augmentations – features like hyperlink previews, contextual definitions, interactive visualizations, and explorable explanations – can be implemented either as Typst library functions (at the language level) or as Rheo format plugins (at the compiler level). The Rheo packages repository gives various examples of how custom Typst functions coupled with format-specific assets (such as JS and CSS in the case of HTML) can produce augmentations, both in the browser and in the other formats Rheo supports.

Rheo.toml

The directory that holds your Typst source is called the **project directory**, and you can compile it like so:

```
rheo compile path/to/projectdirectory
```

In general, there are two ways to configure Rheo:

1. By passing **flags** directly to the CLI command.
2. By specifying configuration in a `rheo.toml` file at the root of the project directory.

If you compile a Rheo project directory without a `rheo.toml` file, the following default settings will be applied to compile your project.

```toml
version = "0.4.0"
content_dir = "./"
build_dir = "build"
formats = ["pdf", "html", "epub"]

[epub.spine]
title = "[project directory name]"
```

Without a spine config, every Typst file under `content_dir` is included, ordered by the directory-scan default.

To point Rheo to a rheo.toml file that is not at the root of the project directory, specify it directly via the CLI:

```
rheo compile path/to/project --config path/to/config
```

Per-page footnotes

In HTML and EPUB output, Rheo produces one output file per page. By default it resets the footnote counter to `1` at the start of every page, so each page numbers its footnotes independently. To let footnotes accumulate continuously across the whole bundle instead, set `reset_footnotes = false` under `[html]` or `[epub]`:

```toml
[html]
reset_footnotes = false   # let footnotes accumulate across pages (default: true)
```

The key is per-format: set it on `[html]` and `[epub]` independently. It defaults to `true`, and as with other settings the CLI takes precedence over `rheo.toml`, which takes precedence over the default (there is no CLI flag for it). PDF is unaffected — it combines your project into a single document, so its footnotes are always continuous regardless of this key.

Automatic package detection

Rheo scans your content files' `#import` statements and auto-detects assets or marrow declared by any `@rheo`-aware package you use, per format. Set `auto_detect_packages = false` on a format's own table to turn that off for that format alone:

```toml
[html]
auto_detect_packages = false   # default: true
```

See Packages for more information.

Fonts

Because Rheo embeds its own copy of the Typst compiler, it resolves fonts itself rather than deferring to any `typst` binary or font cache already on your system. Alongside your system fonts, it looks for a `fonts` directory at the root of your project by default and loads anything it finds there. Set the top-level `font_dirs` key to search different or additional directories instead — paths are resolved relative to the project root unless absolute, and setting this key switches off the automatic `fonts` autoscan, so include `"fonts"` explicitly if you still want it searched:

```toml
font_dirs = ["fonts", "custom/typefaces"]
```

The repeatable `--font-dir` CLI flag, available on `compile` and `watch`, appends further directories on top of whatever `font_dirs` (or the autoscan) already resolved.

Relative linking

Rheo allows you to write documents in plain Typst without requiring any additional syntax or metadata. Because Rheo can combine multiple files into unified outputs, however, we need a way to reference other files in the same Rheo project.

Rheo assigns every source document a *handle* — a short label derived from its path — and you link between documents using standard Typst label syntax:

```typst
#link(<intro>)[Introduction]
#link(<chapters:intro>)[Chapter introduction]
```

When you compile a project with Rheo, these links are resolved and transformed according to the output format.

- In **HTML**, relative links become `<a>` tags that point to the relevant html page.
- In **PDF**, relative links become internal links to the relevant section in the single combined output PDF.
- In **EPUB**, relative links become links to the relevant sections in the EPUB.

Handle assignment

Every `.typ` file in your content directory gets a canonical handle derived from its path relative to that directory.

- **Root-level files** get a bare handle: `content/intro.typ` ⟶ `<intro>`.
- **Nested files** get a path-qualified handle using `:` as the path separator: `content/chapters/intro.typ` ⟶ `<chapters:intro>`, `content/a/b/notes.typ` ⟶ `<a:b:notes>`.

`:` and `.` are valid Typst label characters; `/` is not, which is why `:` is used as the separator.

Escape form

The alias `<handle.typ>` is always available for every vertebra, regardless of other labels in the project:

```
#link(<intro.typ>)[Introduction]         // root file
#link(<chapters:intro.typ>)[Chapter intro] // nested file
```

Use the escape form when the canonical handle would be ambiguous or when you want to be explicit about which file you are linking to.

If you have authored a label in your source that matches the canonical handle for a file, Rheo silently skips injecting the automatic label for that file. The file remains reachable via its escape form.

```
// chapters/intro.typ
= Chapter Introduction <chapters:intro>  // user-authored; Rheo skips auto-injection

// elsewhere
#link(<chapters:intro>)[...]    // resolves to the user label
#link(<chapters:intro.typ>)[...] // escape form also works
```

If any source file defines a label that matches the escape form of another file (e.g. `<chapters:intro.typ>`), Rheo aborts the build with an error naming the offending file and label.

Migrating from older Rheo versions

Before 0.4.0, relative links were written by pointing at the target's file path directly:

```
#link("./another-section.typ")[Another section]
```

This syntax still works, but the handle form above is now preferred. See Migrating projects for how to upgrade an existing project automatically.

`rheo-context`

When Rheo compiles a project, it prepends a small binding to the top of every vertebra:

```
#let rheo-context() = (
  handle: "chapters:intro",
  metadata-of: rheo-metadata,
  ..sys.inputs.rheo-context,
)
```

The `rheo-context` dictionary exposes Rheo's view of the project to your Typst code: which file this is, every file in the spine alongside it, and a way of reading any of their resolved document metadata live. Most of this is global context that can also be read in the format-global `sys.inputs.rheo-context`, but `handle` and `metadata-of` are file-specific. (Rheo injects a distinct binding into each file, so every file sees *its own* `handle`.)

`rheo-context()` is a zero-arg function that returns a dictionary. It is **not** a Typst `context` value, and you can read most of it with no `#context` keyword. The one exception is `metadata-of` (see below).

```
This page's handle is #rheo-context().handle.
```

Fields

The shape of `rheo-context` is designed to be extensible.

Field	Description
`handle`	This file's handle — its `:`-separated identifier, the same handle used for cross-file links.
`spine`	A tree (an array of top-level nodes) mirroring the project's directory and section structure. Each node is a dictionary with four fields: • `title` — the node's title, always path-derived (the file or directory name, prettified), regardless of any `#set document(title: ...)` the vertebra itself authors. • `handle` — the vertebra's handle, or `none` for a group node (a directory or section with no landing file). • `path` — the vertebra's path relative to the project root, or `none` for a group node. • `children` — an array of child nodes, recursing to arbitrary depth (empty for a node with no descendants). You can use a `title` or `handle` to derive the link to a vertebra. Walk `children` to build nested navigation.
`spine-flat`	A flat list of every **clickable** vertebra in spine order (group nodes are omitted). Each entry is a dictionary with three fields: `handle`, `path`, and a `title` that is always path-derived.
`metadata-of`	A function value, `(handle) => dict`, reading another vertebra's `#set document(...)` fields live off the compiled bundle. See below for more information.
`target`	The output format Rheo is compiling for, such as `"html"` or `"epub"`. It is **absent** for PDF, where Typst's native `target()` returns `"paged"`. In authored files, prefer Typst's own `target()` (which Rheo polyfills to return this value) over reading this field directly, as `target()` works everywhere, e.g. `#if target() == "epub" [ ... ]`.
`ext`	The output file extension for the format being compiled, such as `"html"` for HTML or `"xhtml"` for EPUB. Like `target`, it is format-global (the same for every vertebra) and **absent** for PDF, where there are no per-page files. This is the value Rheo uses to build cross-vertebra links.
`rheo-version`	The compiling Rheo binary's own semver string, e.g. `"0.6.0"`. Unlike `target`/`ext`, it is **always** present, on every format, including PDF. A package's own Typst code can read it to enforce a minimum Rheo and fail with a clear message of its own choosing, rather than breaking obscurely against a build-time surface an older Rheo simply doesn't have. Its **absence** is the signal to treat as "older than the release that added this field. (A package can also declare a floor in its own manifest: see Packages.)
`reset-footnotes`	The resolved per-format footnote-reset toggle, as a plain `bool`. Unlike `target`/`ext`, it is **always** present, on every format including PDF, because it's a resolved default rather than something format-gated. This only ever actually takes effect on HTML/EPUB regardless of its value, as a combined PDF has no per-page boundary to reset at. Set `reset_footnotes = false` under `[html]`/

Field	Description
	`[epub]` in `rheo.toml` (default `true`) if you'd rather footnotes accumulate across the whole bundle instead of restarting on every page.

Reading vertebra metadata

Every vertebra compiled under a per-page layout such as HTML or EPUB publishes a small, hidden beacon after its own body as Typst metadata. You can use the `metadata-of` function in `rheo-context` to query this metadata, meaning that you can query attributes such as the document title and date of other vertebrae from any point of the spine, which is useful for building blog feeds or other site listings. Because this is a live query using Typst introspection, it needs the `#context` keyword:

```
// This vertebra's own metadata, looked up by its own handle.
#context {
  let me = (rheo-context().metadata-of)(rheo-context().handle)
  if "date" in me [
    Published #me.date.display("[year]-[month]-[day]").
  ]
}
```

The are a few important nuances to the `metadata-of` function that you should keep in mind if you intend to use it:

- The combined PDF has no per-vertebra metadata at all, as PDF's default layout puts every vertebra into one shared `#document(...)` block (`metadata-of` returns an empty dictionary for every handle).
- A title or other value set inside a bounded code block is invisible to the beacon, and thus also to `metadata-of`. If another vertebra or a `@handle` anchor needs to see it, set the metadata either explicitly at the vertebra's top level in a `#document` block, or through a `#show:` template.
- `datetime.today()` resolves to whatever day the build actually happened on. If a vertebra's date feeds something published downstream such as an Atom feed built with `@rheo/feeds`, using `datetime.today()` date will churn on every rebuild. Use a literal `datetime(year:, month:, day:)` for anything that needs to hold still.
- The beacon's own label carries a reserved `rheo-meta:` prefix. Authoring a label starting with this prefix will result in a build error.

Example: building a blog feed

You can build an index of all pages in your project by mapping `spine-flat` through `metadata-of`. Note that we need the `#context` keyword because we are using the `metadata-of` function:

```
#context {
  let meta-of = rheo-context().metadata-of
  for e in rheo-context().spine-flat {
    let m = meta-of(e.handle)
    let when = m.at("date", default: none)
    if when != none [
      - #link(label(e.handle))[#m.at("title", default: e.title)] ---
#when.display()
    ]
  }
}
```

You can also use `rheo-context().spine` to keep structure when building, for example, a table of contents:

```
#let toc(nodes) = {
  for node in nodes [
    - #if node.handle != none [
        #link(label(node.handle))[#node.title]
      ] else [
        #node.title
      ]
      #if node.children.len() > 0 [
        #toc(node.children)
      ]
  ]
}
#toc(rheo-context().spine)
```

Migrating projects

The `migrate` command upgrades an existing Rheo project to the current version. It reads the `version` field from `rheo.toml`, determines what migrations apply for the gap between that version and the current CLI, and reports or applies them.

Usage

```
rheo migrate path/to/project          # dry run — reports changes, writes nothing
rheo migrate path/to/project --apply  # applies changes and bumps rheo.toml
version
```

Always run the dry run first to review what will be rewritten, then pass `--apply` to write the changes.

`migrate` is best-effort: it applies the mechanical rewrites it knows about, but it does not guarantee that your project will build or behave correctly on the new version. After migrating, rebuild and check the output yourself, and consult the changelog for breaking changes that require manual attention.

What migrate rewrites

`migrate` groups its rewrites by the project version it's migrating from:

From version	What `migrate` rewrites

`< 0.4.0`	• **Link syntax:** cross-file links written as file paths are rewritten to the current handle form:
	`// before`
	`#link("./another-section.typ")[Another section]`
	`// after`
	`#link(<another-section>)[Another section]`

`< 0.5.0`	• **Output format:** direct `rheo-target` references are rewritten onto `rheo-context.target`:

From version	What `migrate` rewrites

	``
	// before
	#if "rheo-context" in sys.inputs and "target" in
	sys.inputs.rheo-context { sys.inputs.rheo-context.target }
	// after
	#if "rheo-context" in sys.inputs and "target" in
	sys.inputs.rheo-context { sys.inputs.rheo-context.target }
	``
	It also rewrites the old `target()` helper to Typst's `target()`. Files already using `target()` need no change.
	• **Spine config:** a `[spine] vertebrae` inclusion-filter glob list is converted to an equivalent `[spine] exclude`, so files the old list never named don't silently start being published under the directory-scan default.

`< 0.5.1`	• **`rheo-context` binding:** the injected per-vertebra binding changed from a bare dictionary to a zero-arg function `rheo-context()`. So existing `rheo-context.field` code keeps working, `migrate` prepends a one-line compatibility shim to each file that reads the binding: ``` #let rheo-context = rheo-context() ``` The shim calls the injected function once and rebinds the name to its dictionary. `migrate` does **not** rewrite individual `rheo-context.field` references — the shim leaves them working untouched.
`< 0.6.0`	• **Feed removal:** Atom feed generation left Rheo in 0.6.0, and nothing about the change is mechanical enough to rewrite. `migrate` instead reports every retired `[html]` feed key and `#let rheo-feed-*` / `rheo-author` binding it finds, each with its file and line — see what 0.6.0 removed below for the full account and where the replacement lives.
any outdated version	Bumps the `version` field in `rheo.toml` to match the current CLI version (`--apply` only).

Removals in 0.6.0

- `[html] feed_base_url`, `feed_author`, `feed_title`
- `[[html.feed_include]]`
- the entire `#let rheo-<key>` variable convention that fed the old generator — `rheo-feed-title`, `rheo-feed-updated`, `rheo-feed-exclude`
- `rheo-author`, whose replacement is `#set document(author: ...)`

Since 0.6.0, a Rheo build will warn when it finds a retired key:

```
`feed_base_url` in [html] is retired and has no effect — Atom feed generation
moved to the Typst package @rheo/feeds — see https://rheo.ohrg.org/feeds
```

A top-level `#let rheo-anything = (1, 2)` used to be a compile error, because the old harvester only accepted a string or boolean literal on the right-hand side. From 0.6.0 it's just an ordinary Typst binding that Rheo doesn't consider.

Build directory

Rheo produces outputs in a simple directory structure with one subdirectory for each kind of output. By default, Rheo produces all outputs (PDF, HTML, and EPUB) in a `build` directory inside the project directory:

```
build/
├── epub
│   └── blog_post.epub
├── html
│   ├── portable_epubs.html
│   └── style.css
└── pdf
    └── portable_epubs.pdf
```

The build directory path is calculated *relative to the content directory*. This is important, as if you change the content directory, then your build directory path will become relative to that directory.

Configuration

CLI flag

You can specify a build directory with either the `compile` or `watch` commands:

```
rheo compile path/to/project --build-dir path/to/build
```

`rheo.toml`

The build directory is specified at the top level of the `rheo.toml`:

```
build_dir = "custom_build_directory"
```

Content directory

By default, Rheo will search your entire project directory for Typst documents. You can, however, indicate a specific subdirectory that Rheo should use if you prefer. This can be helpful for structuring projects, as it allows you, for example, to keep a separate `drafts` folder that Rheo will not compile.

The default content directory is the same path as the Rheo project directory. It is important to note that if you specify a custom content directory, all other configuration such as `build_dir` and `spine` globs will operate *relative to the content directory*.

Configuration

`rheo.toml`

A custom content directory can be specified at the top level of the `rheo.toml`. The path is calculated relative to the project directory:

```
content_dir = "pages"
```

Spines

A spine in Rheo is the backbone or 'table of contents' of Typst source files that should be compiled to an output format. It takes its name from the EPUB specification, in which the spine articulates—or *reticulates*— the set and order of chapters included.

Directory-scan default

With no configuration at all, the spine is built from `content_dir`'s own directory structure:

- Every `.typ` file is included, ordered alphabetically within each directory level.
- A subdirectory with a landing file — `index.typ`, or a file named after the directory itself (e.g. `chapters/chapters.typ`) — becomes a group with its own page and handle.
- A subdirectory **without** a landing file becomes a group node with no page or handle of its own, its title derived from the directory name.
- A leading numeric prefix on a directory or section name orders it among its siblings but is stripped from the displayed title: `01-intro/` → "Intro". The raw name, prefix included, is kept in the handle. Files are not prettified this way — a file's title comes from its own content, not its filename.

This default is enough for most projects: name your files and folders in the order you want them read, and the spine follows.

`[spine] exclude`

To omit files or folders from the scan without moving them out of `content_dir`, list glob patterns (relative to `content_dir`) under `[spine] exclude`:

```
[spine]
exclude = ["drafts/**", "TODO.typ"]
```

Excluded paths are dropped from every format's spine.

`[spine] include`

`[spine] include` is an ordered list of glob patterns that becomes that spine's definitive order, replacing the alphabetical directory scan outright. Patterns are matched in the order you list them. Within a single pattern, matches are sorted lexicographically. *A file matched by none of the listed patterns is dropped from the spine entirely*, thus `include` makes a separate `exclude` for the same files unnecessary.

```
[spine]
include = ["index.typ", "install.typ", "ideas.typ", "flights.typ"]
```

The `include` attribute is one way of restructuring the order in which vertebrae appear in the spine. For finer-grained control, read on to the next section.

`[[spine.section]]`

Using `[[spine.section]]` allows you to group files under virtual directories for the purpose of order and structure in the Rheo spine without actually moving the files on disk. This is useful for reshaping the spine's structure independently of your folder layout:

```
[[spine.section]]
name = "chapters"
include = ["ch-*.typ"]
```

- Files matching `include` get pulled under a virtual `chapters` group, gaining a namespaced handle exactly as if they lived in a `chapters/` directory: `ch-1.typ` → `<chapters:ch-1>`.
- A section's `title` defaults to a prettified version of `name`, and can be overridden explicitly.
- Sections can be nested via `[[spine.section.section]]` to arbitrary depth.
- When `include` lists more than one glob, matches are gathered in glob order and lexicographically within each glob — so listing globs in the order you want lets you control ordering explicitly, not just alphabetically.

Per-format overrides

A format-specific table via `[pdf.spine]`, `[html.spine]`, or `[epub.spine]` will override the global `[spine]` one field at a time.

```toml
[spine]
exclude = ["drafts/**"]

[pdf.spine]
title = "My Book"   # combined-output title, PDF/EPUB only
```

Here PDF still inherits the global `exclude`, even though it only sets its own `title`.

Each format uses its spine slightly differently. See the PDF, HTML, and EPUB format pages for details on how spines are configured and what defaults are applied for each output.

Assets

Assets are non-Typst files that your project needs in its output – things like images, fonts, or data files. Rheo provides a `copy` attribute that lets you specify glob patterns for files that should be copied into your build output.

Copying files to all formats

You can specify a top-level `copy` in your `rheo.toml` to copy files into every format's output directory:

```toml
copy = ["assets/**/*.png", "fonts/**"]
```

Paths are relative to the project root (not the content directory), and directory structure is preserved in the output. Only files are copied; parent directories are created automatically.

For example, if your project looks like this:

```
my-project/
├── rheo.toml
├── assets/
│   └── logo.png
└── chapter.typ
```

Then `assets/logo.png` will be copied into each format's build directory at the same relative path.

Format-specific copies

If you only need certain files for a specific format, you can scope the `copy` to that format's assets section:

```toml
[html.assets]
copy = ["images/**", "fonts/**"]
```

This copies the matched files only into the HTML output directory — handy for web fonts that PDF and EPUB have no use for.

You can combine global and format-specific copies; both will be applied when building a given format.

Format-specific assets

Format plugins can expose additional asset configuration beyond `copy`. For HTML, this includes CSS and JavaScript entrypoints and support for multiple asset blocks — see Custom JS/CSS.

Relation to Typst's bundle format

Typst has an experimental bundle format that handles assets programmatically inside a Typst bundle to declare files that should be written alongside the document output. Rheo's `copy` serves the same purpose but declaratively, at the project level via `rheo.toml`. Rather than writing `asset()` calls in every source file, you declare glob patterns once and Rheo copies the matched files into every (or a specific) format's output directory.

For programmatic control over assets in Typst, you can use the native `#asset` syntax in your project's marrow.

Marrow

Rheo projects treat `.marrow.typ` as a special file that allows you to programmatically generate extra files beyond the one-page-per-vertebra default. In contrast to vertebrae, which are treated as normal Typst compiled to PDF or HTML, marrow is Typst run at the bundle root, meaning that you can write Typst in it to generate new documents and assets. You can use marrow to programmatically derive artifacts such as feeds, sitemaps, search indexes, and generated pages.

Crafting marrow

Create a file named `.marrow.typ` at the top of your content directory, next to your ordinary pages. Inside this file (and only there), `document()` and `asset()` are in scope:

```typst
#document("extra/hello.html", format: "html", title: [Extra])[Hello from the
bundle root.]
#asset("extra/hello.txt", "root-level asset")
```

Compiling this produces `extra/hello.html` and `extra/hello.txt` in your build output, sitting alongside whatever your ordinary vertebrae produce. For more information on what is valid Typst in marrow, see the Typst bundle documentation.

If you'd rather call the file something else, set the top-level `marrow` key in `rheo.toml`, resolved against `content_dir` in `rheo.toml`:

```toml
marrow = "bundle-root.typ"
```

Note that marrow will never produce a `.marrow.html` of its own, and it never appears in the spine, the sidebar navigation, or a template's prev/next pager.

Marrow runs once per **per-page** output format on every compile and every dev-server rebuild. The combined PDF target skips it entirely, because `document()` and `asset()` both hard-error there. Typst reserves them for the bundle target, and a combined PDF has no bundle to speak of.

A page marrow mints is still a first-class Rheo page in every other sense. It receives the same head-asset injection as an ordinary vertebra, and any root-level `#set`/`#show` rule from your template still applies inside it.

Reading across the spine

A Rheo bundle compiles in a single Typst pass, and Typst's own bundle mechanism shares one introspector across every file in it. That means marrow can read `state(...)` and query labels registered by ordinary vertebrae elsewhere in the project. Marrow thus lets a package or project turn data scattered across many pages into pages of its own, without a stub `.typ` file per entry:

```
// across various vertebrae in the spine...
#state("notes", ()).update(old => old + (my-note,))

// in .marrow.typ
#context {
  for n in state("notes", ()).final() {
    document("notes/" + n.name + ".html", format: "html", title: [Note])[#n.body]
  }
}
```

Typst has no facility for enumerating files on disk, so marrow can never discover content by scanning your project directory itself—only by reading `state`, labels, or `sys.inputs.rheo-context`, all of which have to have been populated by something else in the bundle first.

Using beacons

Marrow runs inside the same Typst compile as every ordinary vertebra, but it runs before any of their HTML exists. This is a problem if you want to interpolate a page's *compiled* content into some asset that you program using marrow, such as an Atom entry's `<content>` or a search index's stored body. Rheo therefore allows you to specify a placeholder element called a **beacon**, which is resolved only once every ordinary page has actually compiled in a post-compilation pass:

```
<rheo-content page="notes/etal.html" select="main" as="escaped"/>
```

You can specify a beacon in marrow like so:

```
#asset(
  "excerpt.xml",
  "<entry><body><rheo-content page=\"notes/etal.html\"/></body></entry>",
)
```

Head control

A `<rheo-head>` element written anywhere in a page's body has its children hoisted into that page's `<head>` when the page compiles, and the wrapper itself disappears from the output:

```
#html.elem("rheo-head", html.elem("link", attrs: (rel: "canonical", href:
"https://example.com/a.html")))
```

The <rheo-head> element could sit at the top of the body, the bottom, or halfway through your prose. Multiple <rheo-head>s on one page will be hoisted in the order they appear.

`.rheo/head.html` for every page

For something every page's head ought to carry, you can specify a `.rheo/head.html` from your project's marrow. This is an HTML fragment with no wrapping (<html>/<head>/<body>), and its top-level elements are appended to **every** compiled page's <head>, after whatever that page's own <rheo-head> wrappers already hoisted in:

```
#asset(
  ".rheo/head.html",
  "<link rel=\"alternate\" type=\"application/atom+xml\" href=\"https://example.
com/feed.xml\" title=\"My Site\">",
)
```

Control assets

The `.rheo/` output prefix is reserved. An asset created in this folder should be considered a message from the bundle to Rheo itself. It will be consumed while compiling, but never written to a format's build output.

This matters most when a package ships a `.rheo/*` convention your installed Rheo predates: an unrecognised member simply does nothing yet, and it has not leaked into your output somewhere you haven't checked. Everything under `.rheo/` stays out of an EPUB container exactly as it stays out of an HTML build's output directory, dropped before either builder sees it. `@rheo/feeds` uses it, for example, to drop one autodiscovery link onto every page of your site in a single call.

PDF

Typst, the programming language and compilation toolchain that underwrites Rheo, natively and fully supports PDF.

One PDF from many files

Where the Typst CLI compiles a single file, Rheo 'reticulates' every source file in your project into one unified PDF. This lets you keep each section of your book in its own file — so that a section can double as its own web page in the HTML output — while still producing a single document for print.

The PDF format always produces a single combined document: every source file in its spine is merged into one output PDF, in directory-scan order. Give the combined document a title, and narrow which files are included with `exclude`:

```
[pdf.spine]
title = "My book"
exclude = ["drafts/**"]
```

Because the source files are combined, relative links between them are automatically resolved to internal document links that point to the relevant section in the output PDF.

Rheo also copies any assets matched by your `copy` patterns into the PDF build directory.

EPUB

EPUB is, in our view, the most promising document format for the future of digital reading. It combines the portability of PDF – a document encapsulated in a single file that renders reliably –

with the flexibility of HTML – dynamic layout that adapts to different screens, and the ability to support interactivity and extensions.[5]

Typst does not yet support EPUB. The Rheo EPUB format bridges the gap, letting you compile EPUB documents straight from a Rheo project directory. As EPUB export is on Typst's roadmap, we will track this feature closely in the upstream and look to integrate with it when it lands in the future.

How Rheo builds an EPUB

Unlike the PDF and HTML formats, EPUB always produces a single merged output from your project. Rheo handles the whole packaging pipeline: it converts your Typst source files to XHTML, generates a table of contents from your document headings, and bundles everything into an EPUB archive that validates against the spec.

Math renders the same way it does in HTML output — straight to MathML, both inline and display — and Rheo goes one step further here than it does for HTML: any XHTML file that ends up containing MathML gets marked with the `mathml` property in the EPUB manifest, as the EPUB 3 spec requires for reading systems to render it correctly.

The spine determines which files are included and in what order. An EPUB must have a spine in order to be valid; if you don't specify one, Rheo infers a default from the directory-scan order. Give the EPUB a title and narrow the included files with `exclude`:

```toml
[epub.spine]
title = "My book"
exclude = ["drafts/**"]
```

The `title` field sets the EPUB's metadata title. Rheo also generates a unique identifier for the document and populates other metadata fields such as language and publication date, though both the identifier and the date are yours to set explicitly under `[epub]` if the auto-generated versions don't suit — a stable identifier matters if you're distributing revisions of the same book and want reading systems to recognise them as the same work across updates:

```toml
[epub]
identifier = "urn:isbn:9780000000000"
date = 2024-01-24T00:00:00Z
```

Leave either out and Rheo falls back to its defaults: a freshly generated `urn:uuid:...` for the identifier, and no `dc:date` element at all for the date.[6]

Relative links between source files are resolved to internal links that navigate between sections in the EPUB.

Although the EPUB is a single merged archive, it is bundled from one output page per source file, and by default Rheo resets the footnote counter to `1` at the start of every page, so each section numbers its footnotes independently. Set `reset_footnotes = false` under `[epub]` to let footnotes accumulate continuously across the whole book instead:

```toml
[epub]
reset_footnotes = false
```

---

[5] For more on why we believe EPUB deserves wider adoption, see Portable EPUBs.

[6] The date has to be a full TOML offset date-time, not a bare date — Rheo parses it as RFC 3339, and a plain `2024-01-24` with no time or offset attached fails that parse and is silently dropped rather than erroring.

Detecting the EPUB target

EPUB compiles through Typst's HTML target, so Typst's native `target()` returns `"html"` for both HTML and EPUB output – the two are indistinguishable to standard Typst. Rheo synthesizes an extra value so you can tell them apart: inside an EPUB build, `target()` returns `"epub"`.

```
#context if target() == "epub" {
  // EPUB-only markup
} else if target() == "html" {
  // plain HTML output
}
```

Rheo polyfills `target()` to report one of three values: `"paged"` (PDF), `"html"` (HTML), and `"epub"` (EPUB).

HTML

Typst experimentally supports HTML. This means that not all Typst syntax will translate to a meaningful HTML structure. The most common features in everyday prose are all supported, however, such as text markup, links, headings, footnotes, citations, and mathematics. For more information on which features are currently supported in Typst's HTML export, refer to the HTML export tracking issue.

Math is worth calling out on its own, because it's one of the areas where Typst's HTML export is already fully mature rather than experimental. Both inline equations and display blocks compile straight to MathML, so an integral or a matrix keeps its own semantics in the DOM instead of becoming a rasterised image or a pile of hand-positioned `<span>`s. Rheo does no translation work here itself; it just carries Typst's MathML through unchanged to both HTML and EPUB output.

From one file to a whole site

Where the Typst CLI produces a single HTML file, Rheo turns your project into a static site. Each Typst source file becomes its own HTML page, and relative links between files are rewritten from `.typ` to `.html` so that navigation works in the browser.

Rheo also provides a development server with live reloading when you use the `watch` command, so you can see changes in the browser as you edit your source files:

```
rheo watch my-project --html --open
```

The HTML format supports configurable CSS and JavaScript entrypoints, as well as asset copying for additional files your site needs. These entrypoints are also the mechanism through which reading augmentations can be delivered – the tooltip example, for instance, uses a custom Typst function and a JavaScript entrypoint to provide inline tooltips in the HTML output.[7]

You can customize which files are included in the HTML output using a spine. By default, Rheo compiles every Typst file under `content_dir`; to exclude certain files from the site, narrow it with `[html.spine] exclude`:

```
[html.spine]
exclude = ["drafts/**"]
```

[7] Note that there is an experimental bundle format in upstream Typst which allows you to achieve similar results. We are tracking this closely, but for the time being have not incorporated this feature/format into Rheo on account of some critical deficiencies such as the inability to have multiple bibliographies in the same source file.

Rheo sites can also carry a feed — Atom, RSS, or JSON Feed — generated by the `@rheo/feeds` Typst package rather than a core config key.

Because each source file becomes its own page, Rheo resets the footnote counter to `1` at the start of every HTML page by default, so each page numbers its footnotes independently. Set `reset_footnotes = false` under `[html]` to let footnotes accumulate continuously across the whole site instead:

```
[html]
reset_footnotes = false
```

HTML Assets

The HTML format plugin extends Rheo's generic asset system with CSS and JavaScript entrypoints, configurable via `[html.assets]` in `rheo.toml`.

CSS

When Rheo generates HTML, it injects a default stylesheet that gives your site a clean, mobile-friendly look out of the box. 'Screening the subject' is a website generated with the default Rheo stylesheet for reference.

You can fully customize the stylesheet by adding a `style.css` at the root of your project directory. Note that if your project contains a custom `style.css`, *none* of the styles in the default stylesheet will be applied. If you want to build on the default styles, copy and paste the default stylesheet into the `style.css` file in your project directory.

You can customize the entrypoint for the CSS included in the HTML build in `rheo.toml`. Note that this path is relative to the Rheo root (not the content directory):

```
[html.assets]
css_stylesheet = "./index.css"
```

Javascript

Rheo also makes it easy to reference JS in your HTML site by ferrying the code in `index.js` through to your HTML build. Take a look at the tooltip_html example project, which leverages the `index.js` entrypoint using a modern JS bundling toolchain to provide a package through which you can specify tooltips in the HTML build as a custom function in Typst.

You can customize the entrypoint for the JS included in the HTML build in `rheo.toml`. Note that this path is relative to the Rheo root (not the content directory):

```
[html.assets]
js_scripts = "./entrypoint.js"
```

Multiple asset blocks

`[html.assets]` can be written as an array of blocks using TOML's double-bracket syntax (`[[html.assets]]`). Each block is a distinct asset set with its own `js_scripts`, `css_stylesheets`, and `copy`, and an optional `dest` subfolder within the HTML build directory.

```
[[html.assets]]
dest = "tooltip"
js_scripts = "tooltip/index.js"
```

```toml
css_stylesheets = "tooltip/index.css"

[[html.assets]]
dest = "annotations"
js_scripts = "annotations/index.js"
```

Multiple blocks are what you want when a project pulls in more than one independent set of scripts or stylesheets — say, assets bundled from several packages. Each block's files are written under its `dest` subdirectory, or directly into the HTML root if `dest` is omitted.

For automatic injection of asset blocks from Typst packages, see Packages.

Packages

A Rheo package is a standard Typst Universe package that ships its own web assets — CSS, JavaScript, or files to copy — alongside its Typst code.

Using packages

To use a Rheo-compatible package, import it in your Typst source files as you normally would:

```
#import "@rheo/tooltip:0.1.0": tooltip
```

That's all! Just like the Typst CLI, Rheo will prewarm the `@rheo` namespace so that all of the existing packages are available. When Rheo compiles your project, it reads every `#import` statement in your content files, identifies any Typst Universe and Rheo packages, and makes them available in your project.

A Rheo package is essentially a Typst package that can also provide assets and marrow. This means that a Rheo package can provide a Typst interface to custom functionality in any given format supported by Rheo, and/or abstract the same source Typst across multiple formats to gracefully degrade an interactive modal from HTML to PDF, for example.

See the slides package for an example exposing RevealJS by way of a Typst API to downstream projects.

Turning off automatic detection

Rheo handles asset and marrow import for every `#import` in your content files by default. If you don't want this auto-detection of Rheo packages, you can turn it off in your `rheo.toml`:

```toml
auto_detect_packages = false
```

Creating a Rheo-compatible package

If you are authoring a Typst Universe package and want to ship assets that integrate with Rheo's HTML output, add a `[tool.rheo.html]` section to your package's `typst.toml`:

```toml
[package]
name = "rheo-tooltip"
version = "0.1.0"
entrypoint = "lib.typ"

[tool.rheo.html]
```

```toml
js_scripts = "dist/index.js"
css_stylesheets = "dist/index.css"
```

The `[tool.rheo.html]` section accepts the same fields as `[[html.assets]]`:

- `js_scripts` — path to a JavaScript file to inject into HTML pages.
- `css_stylesheets` — path to a CSS file to inject into HTML pages.
- `copy` — glob patterns for additional files to copy into the HTML build.

Paths are relative to the package root and are resolved against the package's location in the local Typst package cache. When a user imports your package and builds their Rheo project, your assets are injected without any extra steps on their part.

Requiring a minimum Rheo version

A package can also declare the oldest Rheo it works with, in a `[tool.rheo]` table alongside (or instead of) the format-specific asset section above:

```toml
[tool.rheo]
min_version = "0.6.0"
```

Rheo will only check a package if this attribute exists. Where a project imports a package whose declared floor is above the running Rheo, the build will fail before any Typst compile even starts.

```
@acme/widgets:1.0 needs rheo >= 0.8.0, but this is rheo 0.6.0
@acme/charts:2.0 needs rheo >= 0.9.0, but this is rheo 0.6.0
Upgrade rheo: https://rheo.ohrg.org
```

A package that isn't resolved locally, has no manifest, or simply never sets `min_version` will never throw this kind of error. `min_version` is only checked from Rheo `>=0.6.0`.

Feeds

The `@rheo/feeds` package mints a syndication feed — Atom 1.0, RSS 2.0, or JSON Feed 1.1 — from the pages a project already has. A feed is a file a reader's newsreader polls on its own schedule, so new pages surface without anyone visiting your site to go and check. Describe one feed in Typst, point it at a slice of your project, and the package writes the file at the end of the build and puts an autodiscovery link in every page's `<head>`.

Rheo used to generate the Atom half of this itself, from a handful of `[html]` keys in `rheo.toml` read by a Rust plugin built into the engine. Since 0.6.0 that plugin is retired and feeds come from the package instead, built on the marrow, beacon, and control asset primitives. The wire format a subscriber's reader parses is unchanged; only the place you configure it has moved.

The package needs Rheo 0.6.0 or later, because those three build-time surfaces — a per-vertebra metadata beacon, page transclusion, and an autodiscovery control asset — don't exist before it. It asserts that floor itself rather than failing quietly on an older Rheo, which matters here because all three surfaces fail silently: on 0.5.2 the same project compiles clean and simply ships no feed.

Importing the package

Import the package in whichever vertebra you want to configure your feeds from:

```
#import "@rheo/feeds:0.1.0": feed, configure, spine
```

Configuring a feed

Describe one feed, and hand it to `configure`:

```
#configure(feeds: (
  feed(
    title: "My Site",
    base-url: "https://example.com",
    sources: (spine(),),
  ),
))
```

Call `configure` from any vertebra you like. It only appends onto a document-wide state, and the package's own build step reads that state once, at the end of the compile, and mints `feed.xml`. `spine()` reproduces the retired generator's default behaviour: every vertebra in the project becomes a candidate entry, so this block is the whole migration for a site that only ever pointed the old generator at a base URL.

Feed fields

`feed(...)` validates its arguments and returns the config dictionary that every source is later handed:

Field	Default	Description
`title`	required	The feed's own title, a non-empty string. It doubles as the `title=` on each autodiscovery link.
`base-url`	required	An absolute URL, scheme included. It prefixes an entry's `page` into an absolute URL, and is emitted as the feed's own `rel="alternate"` link — the site a reader visits from a subscription, as distinct from `rel="self"`, the feed file itself.
`sources`	required	An array of at least one source function. See Where entries come from.
`path`	`"feed.xml"`	Where the feed is written, relative to the HTML build directory.
`author`	`"Rheo"`	The feed-level author, inherited by any entry that names none of its own.
`subtitle`	`none`	An optional feed subtitle.
`format`	`"atom"`	The serializer: `"atom"`, `"rss"`, or `"json"`.
`content`	`"html"`	What an entry's body resolves to. `"html"` and `"xhtml"` splice in the entry's own page via transclusion `none` omits the body entirely, leaving a `summary` to stand in for it.
`limit`	`none`	A positive integer capping how many entries the feed carries, or `none` for all of them.

Multiple feeds and formats

A feed is just a value, so you can register more than one in the same `configure` call, each over its own slice of the site and with its own title, author, and output path:

```
#let posts = spine(filter: e => e.handle.starts-with("posts:"))
#let essays = spine(filter: e => e.handle.starts-with("essays:"))

#configure(feeds: (
```

```
  feed(path: "feed.xml", title: "My Site --- Posts", base-url: "https://example.
com", sources: (posts,)),
  feed(
    path: "essays.xml",
    title: "My Site --- Essays",
    author: "Someone Else",
    base-url: "https://example.com",
    sources: (essays,),
  ),
))
```

The same applies over **format**, so identical sources can be registered at three paths and ship as Atom, RSS, and JSON Feed from one call:

```
#configure(feeds: (
  feed(path: "feed.xml", title: "My Site", base-url: "https://example.com",
sources: (posts,)),
  feed(path: "rss.xml", title: "My Site", base-url: "https://example.com",
sources: (posts,), format: "rss"),
  feed(
    path: "feed.json",
    title: "My Site",
    base-url: "https://example.com",
    sources: (posts,),
    format: "json",
    content: none,
  ),
))
```

Every page's `<head>` then carries one autodiscovery link per feed, each with its own type, so whichever format a visitor's reader prefers, it finds it.

The entry model is shared across the three, but the formats are not equivalent, and where they diverge the package maps rather than drops:

Format	What differs
`"atom"`	The reference mapping, and the only one that takes `content: "xhtml"`. An entry with both `published` and `updated` gets both elements.
`"rss"`	RSS has one date per item, so `pubDate` takes `published` where there is one and `updated` otherwise, and the channel's `lastBuildDate` carries the newest `updated`. An author is emitted per item as `<dc:creator>` rather than `<author>`, which RSS requires to hold an email address.
`"json"`	Summary-only, and it needs `content: none`. There is no JSON-safe way yet to splice a page's compiled HTML into a JSON string, which is a Rheo limitation rather than a `@rheo/feeds` one.

Where entries come from

A feed's `sources` field is an array of plain functions, `cfg => (entries,)`, where `cfg` is the feed config above. A built-in source takes its own options and **returns** the source, so it reads as an ordinary call at the call site — `spine(filter: ...)`, not `spine.with(filter: ...)`, which yields a function that then refuses the positional `cfg` and fails with `error: unexpected argument`.

The package ships two:

Source	What it reads
`spine(filter:, select:)`	The project's own spine: every vertebra is a candidate entry unless `filter` says otherwise. The predicate runs against the vertebra's `handle`, `path`, and `title`, not its position in the directory tree, so a filtered source can reach arbitrarily deep — a post nested three directories down still matches a `handle.starts-with("posts:")` filter the same as one at the top level. Each entry's title, dates, and categories come straight from that vertebra's own `#set document(...)` call. `select`, where given, is passed through to every entry this source produces.
`items(filter:, label-name:)`	Any `#metadata((...)) <feeds:item>` beacon emitted anywhere in the project, by hand or via the package's own `item(...)` helper, with the metadata value shaped as an entry. Rheo compiles a project in one pass, so `items()` sees every beacon in the bundle rather than only those in the calling vertebra. Reach for this where the data has no accessor of its own to call, such as a hand-authored page with no registry behind it.

Entry fields

A source fills in as much of an entry as it has. Only `title` is mandatory on the way in; the package resolves the rest, and checks every type it is given, so a wrong value is named against this table rather than surfacing from somewhere inside the XML serializer.

Field	Default	Description
`title`	required	A string or content value.
`page`	one of `page`/`url`	The entry's page, relative to the HTML build directory, joined onto the feed's `base-url`.
`url`	one of `page`/`url`	An absolute URL, where you would rather give one directly than have it built. A page-relative path belongs in `page`.
`published`	one of `published`/`updated`	A real `datetime(year: ..., month: ..., day: ...)`, never a string.
`updated`	one of `published`/`updated`	Falls back to `published` where only that is given.
`id`	the entry's `url`	A stable, globally unique string. It need not be a URL: where it isn't, an RSS `guid` is emitted with `isPermaLink="false"` so a reader doesn't try to follow it.
`select`	`none`	A region selector for this entry's body, passed through to the same mechanism described under page chrome.
`summary`	`none`	A plain-text summary, emitted alongside the body rather than instead of it.
`categories`	`()`	An array of strings, even for a single tag: `("note",)` and never `"note"`.

Field	Default	Description
`author`	the feed's `author`	A plain name.

Sourcing from another package

Because a source is only a function, a package that already exposes its own data can be read directly. `@rheo/rookery`'s labelled notes are the case to reach for first: `ideas(tags:)` hands back an array of every matching note, synchronously, so a source can call it rather than going by way of a beacon. Rookery's own `href` field is relative to wherever a note happens to be read from, and a source runs at the bundle root, so the leading `../` run has to be stripped first:

```
#import "@rheo/feeds:0.1.0": feed, configure
#import "@rheo/rookery:0.5.0": ideas

#let root-relative(h) = {
  let out = h
  while out.starts-with("../") { out = out.slice(3) }
  out
}

#let from-ideas(tags: none) = cfg => (
  ideas(tags: tags)
    .filter(e => e.href != none and e.updated != none)
    .map(e => (
      id: e.id,
      title: e.text,
      page: root-relative(e.href),
      updated: e.updated,
      published: e.minted,
      summary: e.body,
      categories: e.tags,
    ))
)

#configure(feeds: (
  feed(
    path: "notes.xml",
    title: "My Site --- Notes",
    base-url: "https://example.com",
    content: none,
    sources: (from-ideas(tags: "note"),),
  ),
))
```

Neither package imports the other: `from-ideas` is a plain function reshaping rookery's own row shape (`href`, `text`, `minted`) into the entry shape `@rheo/feeds` expects (`page`, `title`, `published`). Reshaping one vocabulary into the other is the project's job, and getting it wrong surfaces at build time as `dictionary does not contain key "page"` — an error naming the key you asked for rather than the one that exists, so read the accessor's own row shape rather than assuming it matches the entry table above. The parentheses wrapping the `.filter(...).map(...)` chain are load-bearing, not decorative[8]. `content: none` is required here rather than a preference — a rookery note's page is

[8]Spread across lines without them, the chained calls fall out of the `#let`'s single expression and back into markup, which typesets as a paragraph of code rather than binding a function.

minted rather than compiled, so it has no rendered body to splice into an entry, and these entries carry rookery's plain-text `body` as a summary instead.

Anything with its own array-returning accessor becomes a source the same way, in a few lines that reshape its fields into the entry table above.

Keeping page chrome out of your entries

Where a feed's `content` is `"html"` (the default) or `"xhtml"`, an entry's body is spliced in from the page itself, and you want the article rather than the header and navigation sitting around it. This is Rheo's own default for splicing one page's content into another, and a feed entry is one consumer of it among others. Wrap the article in `<main>` and keep the chrome outside it:

```
#html.elem("main", doc)
```

Where your template already reserves `<main>` for something else, mark the region you want with the `rheo-feed-content` class:

```
#html.elem("div", attrs: (class: "rheo-feed-content"), doc)
```

Rheo checks four things in order before giving up: `<main>`, then a generic `.rheo-content` class meant for uses beyond feeds, then this specific `rheo-feed-content` one (kept working as a compatibility alias), then the whole `<body>`. An entry's own `select` field, where a source sets one, is passed straight through to the same mechanism, letting a source point at a differently-marked region per entry rather than relying on the page-wide default.

Required fields and silent drops

`title` is required, with no fallback chain. The retired generator fell back from an explicit title to the HTML spine's own title to the project's directory name; `feed(...)` refuses to build without one.

An entry with no date is dropped rather than dated some other way. Atom requires `<updated>`, and Typst has no way to read a compiled file's modification time the way the retired generator did, so a `spine()` entry with no `#set document(date: ...)` — and no `published` or `updated` from anywhere else — never becomes a feed entry at all. This doubles as the replacement for the old `rheo-feed-exclude`: an undated cover page or index simply never becomes a candidate.

`<published>` is a genuine addition rather than parity. The retired generator only ever wrote `<updated>`, whereas an entry with both `published` and `updated` set now gets both elements, with their own distinct values when the two differ.

One trap carries over unchanged: `datetime.today()` resolves to whatever day the build happens to run on, so a page dated with it produces an entry that looks freshly updated on every deploy, forever, to every subscriber. Give a post's `#set document(date: ...)` a literal `datetime(year: ..., month: ..., day: ...)` instead.

Within one date, entries keep their source's order. Spine dates are day-granular — one `#set document(date: ...)` per vertebra — so two posts published on one day are common, and the feed lists them in the order the source produced them. Where that matters, give the two posts distinct dates rather than relying on source order.

Migrating from the retired `[html]` feed keys

Every `[html]` key and `rheo-*` variable the old generator read has a replacement here. See Migrating projects for what 0.6.0 removed and the warnings a build now emits for a retired key.

Retired `rheo.toml` key	Here
`[html] feed_base_url`	the feed config's `base-url`
`[html] feed_title`	`title`
`[html] feed_author`	`author`
`[[html.feed_include]]`	no longer needed — a source already lists exactly what it lists

Retired per-vertebra variable	Here
`#let rheo-feed-title = "..."`	the entry's `title` — usually `#set document(title: ...)`, which `spine()` reads back
`#let rheo-feed-updated = "..."`	`#set document(date: datetime(...))`, or a composed override over a source's output
`#let rheo-feed-exclude = true`	omit the page from every source's selection, or leave it undated — either drops it from the feed

Slides

The `@rheo/slides` package turns one Typst source file into two outputs simultaneously: a printable PDF script and an interactive RevealJS presentation in the browser. Write a talk's script in Typst, insert slides into sections of it (also written in Typst), and you get a slide deck alongside a script annotated with markers noting when to advance.

Importing the package

Import the package at the top of your slides source file:

```
#import "@rheo/slides:0.1.0": template, slide
```

Defining slides

Use the `slide` function to mark each slide. Each call becomes one RevealJS `<section>` in the browser and a `SLIDE` marker in the PDF:

```
#slide(title: [Introduction])[
  Lorem ipsum dolor sit amet, consectetur adipiscing elit.
  Slides can contain any Typst content: lists, figures, math, citations.
]

#slide(title: [Methods])[
  - Step one: collect data
  - Step two: process data
  - Step three: analyse results
]
```

A slide's title is sticky rather than per-slide. The package injects a title bar into the deck and repopulates it on every slide change from the current section's own title, so a slide that names no title keeps showing the last one that did — which is what you want for a run of slides belonging to one part of a talk. Pass `title: none` to clear the bar for a slide that should carry no heading at all.

Argument	Default	Description
`title`	`auto`	The slide's title, shown in the deck's title bar. Left at `auto` the previous slide's title carries over; `none` clears the bar.
`transition`	`auto`	A per-slide override of the deck's transition, emitted as `data-transition` on that section. Left at `auto` (or set to `none`) the slide inherits whatever the deck is using. Any other value must be one of the transition names below, or the build fails naming it.
`inline`	`false`	Whether the slide's body is also typeset in the PDF, beneath its marker. See PDF script output.

Applying the template

Wrap your document with the `template` show rule to activate the layout:

```
#show: template.with(
  theme: "white",
  transition: "slide",
  first-slide: [
    = My Presentation

    Author Name
  ],
)
```

Argument	Default	Description
`theme`	`"black"`	One of RevealJS's built-in themes, emitted as `data-theme` and used to pick the stylesheet the package injects at runtime. The accepted names are `beige`, `black`, `black-contrast`, `blood`, `dracula`, `league`, `moon`, `night`, `serif`, `simple`, `sky`, `solarized`, `white`, and `white-contrast`; anything else fails the build rather than silently falling back.
`transition`	`none`	The deck-wide transition: `none`, `fade`, `slide`, `convex`, `concave`, or `zoom`. Left unset, RevealJS's own default applies.
`first-slide`	`none`	Arbitrary Typst content, rendered as the opening slide.
`title`	`none`	The deck's title, seeding the title bar for every slide after the cover until a `slide(title: ...)` replaces it. Where `first-slide` is omitted, the opening slide becomes a level-1 heading containing this instead.

One of `first-slide` or `title` is required, and the template asserts as much rather than producing a deck with no cover.

Configuring the spine

A single-file project needs no spine config at all — the directory-scan default already includes `paper.typ`. Give the PDF a title, if you'd like one:

```
[pdf.spine]
title = "My Presentation"
```

PDF script output

The PDF is a script for the person giving the talk, not a printed copy of the deck. Each `slide` call renders as a small red `SLIDE` marker in the flow of the prose, telling you where to advance; the slide's own body is left out, on the reasoning that a slide's content is a prompt for what you are about to say rather than part of the script. Pass `inline: true` on a slide whose body you do want typeset beneath its marker.

This makes the PDF a working script, and it is why the same file can carry prose the deck never shows: anything outside a `#slide[...]` call is script-only, since the deck is assembled from the slide sections alone.

Customising the RevealJS CSS

The `@rheo/slides` package injects its own base stylesheet automatically. To layer project-specific overrides on top, attach a CSS file via `[[html.assets]]` in `rheo.toml`:

```
[[html.assets]]
css_stylesheet = "style.css"
```

Your `style.css` loads after the package base styles, so any rule you write wins.

RevealJS CSS variables

RevealJS themes expose CSS custom properties you can reference anywhere in your stylesheet. Writing overrides against these rather than against literal colours keeps them theme-agnostic, so switching `theme` doesn't strand your stylesheet on the old palette.

Property	What it holds
`--r-main-color`	Foreground text colour.
`--r-background-color`	Slide background colour.
`--r-main-font-size`	Base font size, which slide-relative `em` sizes resolve against.
`--r-heading-color`	Heading colour.
`--r-link-color`	Link colour.

```
.reveal .slides figcaption {
  border-top: 1px solid color-mix(in srgb, var(--r-main-color) 25%, var(--r-
background-color));
}
```

Common overrides

Title slide heading colour — target the first slide's heading:

```
.reveal .slides section:first-child h2 {
  color: #e7ad52;
}
```

Font sizes — scale slide body and captions independently:

```
.reveal .slides > section    { font-size: 0.8em; }
.reveal .slides blockquote   { font-size: 0.9em; }
```

```
.reveal .slides figure       { font-size: 1.5em; }
.reveal .slides figcaption   { font-size: 0.4em; }
```

The title bar — the package appends it to `.reveal` as `.slide-title-bar`, so it is styled and positioned like any other element:

```
.reveal .slide-title-bar { font-size: 0.5em; opacity: 0.6; }
```

Keeping your original paper styling

Typst's conditional rendering lets you keep a publishable, typeset paper *as well as* its script and slides in the same document. The trick is to overload the `slide` function so that it returns an empty block when you are not building the presentational format, which means one source file renders through Rheo to both:

- a publishable PDF, HTML, or EPUB, and
- a presentation — a PDF script with slide markers, and a RevealJS slide deck.

Define a global boolean and shadow `slide` at the top of your document:

```
#let is-presentation = false

#let slide = if is-presentation { slide } else { (..args) => [] }
```

When `is-presentation` is `false`, every `#slide[...]` call produces no content, so the document compiles as a clean paper with none of the slide scaffolding visible. Set it to `true` (and apply the `template` show rule) to build the presentation instead. The same boolean can drive other conditional styling — a paper might use double spacing while the script uses single spacing:

```
#set par(leading: if is-presentation { 0.65em } else { 1.3em })
```

Any show or set rule that differs between outputs can be gated on `is-presentation` in the same way.

Bibliography

Pourciau, Sarah. 2022. "On the Digital Ocean." *Critical Inquiry* 48 (2): 233–61. https://doi.org/10.1086/717319.