



Enhancing Typst

An Interactive Tour and Collaborative Editor

12.06.2026

Authors:

David Bachmann · Dominik Bühler

Advisor:

Thomas Bocek

Co-advisor:

Remo Frey

Industry Expert:

Roman Blum

Bachelor Thesis · Spring 2026

ABSTRACT

Typst is a modern, programmable typesetting system and a growing alternative to LaTeX. Despite its momentum, its young ecosystem still lacks two things: a guided, hands-on way to learn the language, and an open way for multiple people to work on the same document, as the existing real-time editor is closed-source and available only as a hosted cloud service. This thesis sets out to close both gaps with web-based tooling.

The work delivers two complementary web applications built on a shared foundation. The first, *Tour of Typst*, is an interactive learning platform that teaches the language step by step through curated exercises with a live editor and preview, and deploys as a fully static site. The second is a real-time collaborative editor that lets multiple authors work on the same Typst project simultaneously, with authentication, project management, and shared assets. Being open-source, it can be self-hosted as well as run in the cloud. Both applications are powered by `typst-web`, a reusable in-browser Typst editing core that runs the compiler directly in the browser via WebAssembly and adds editor features such as auto-completion and hover definitions.

This report documents the requirements, design, implementation, and quality assurance of all three components, and evaluates the result against the project goals and a post-build user survey. Together, the tooling produced in this thesis lowers the barrier to entry for Typst, making it more approachable for newcomers and easier for everyone to use directly from the browser.

ACKNOWLEDGEMENTS

We would like to thank our advisor, Thomas Bocek, for his time and guidance throughout this thesis. We are especially grateful for his quick responses whenever we had questions, which kept us moving and made the collaboration both productive and enjoyable.

CONTENTS

1	Introduction	1
1.1	Context	1
1.2	Goals	1
1.3	Scope and Delimitations	2
1.4	Thesis Structure	2
2	State of the Art	3
2.1	The Onboarding Problem	3
2.2	Interactive Learning and the Tour	3
2.2.1	Pre-Build User Survey	4
2.3	Collaborative Editing and the Editor	6
2.4	A Shared Foundation	6
3	Requirements	7
3.1	Tour of Typst	7
3.1.1	Functional Requirements	7
3.1.2	Non-Functional Requirements	9
3.2	Collaborative Editor	9
3.2.1	Functional Requirements	10
3.2.2	Non-Functional Requirements	11
4	Design	13
4.1	System Overview	13
4.2	typst-web	14
4.2.1	Architecture	14
4.3	Tour of Typst	15
4.3.1	Architecture	15
4.3.2	Content organization	15
4.3.3	Navigation	16
4.3.4	Lesson experience	16
4.3.5	Solutions and progress	17
4.4	Collaborative Editor	18
4.4.1	Architecture	18
4.4.2	Access control	18
4.4.3	Data model	19
4.4.4	Login	20
4.4.5	Dashboard	20

4.4.6	Project creation	21
4.4.7	Editor layout	22
4.4.8	Presence	23
5	Implementation	24
5.1	Quality Assurance	24
5.1.1	Development Environment	24
5.1.2	Guidelines	25
5.1.3	Static Analysis and Code Style	25
5.1.4	Dependency Management	26
5.1.5	Testing	27
5.1.6	Continuous Integration	28
5.2	Deployment & Publishing	29
5.2.1	typst-web	29
5.2.2	Interactive Tour	30
5.2.3	Collaborative Editor	30
5.3	typst-web	31
5.3.1	The engine in WebAssembly	31
5.3.2	The TypstProject service	39
5.3.3	The CodeMirror extension	46
5.3.4	The Yjs adapter	51
5.4	Tour of Typst	53
5.4.1	Content as data	54
5.4.2	The lesson workspace	56
5.4.3	Solutions and progress	58
5.5	Collaborative Editor	59
5.5.1	Monorepo Layout	59
5.5.2	Backend	59
5.5.3	Invitations	64
5.5.4	Binary asset uploads	64
5.5.5	Frontend	65
5.5.6	Error Handling and Logging	69
5.5.7	End-to-end Type Safety	70
6	Evaluation	71
6.1	Post-Build Usability Survey	71
6.1.1	Overview	71
6.1.2	Quantitative Results	71

6.1.3	Qualitative Feedback	73
6.1.4	Conclusion	74
6.2	Requirements Verification	74
6.2.1	Tour of Typst	74
6.2.2	Collaborative Editor	76
6.3	Results	77
6.3.1	Interactive Tour	78
6.3.2	Collaborative Editor	79
7	Conclusion & Future Work	84
7.1	Conclusion	84
7.2	Future Work	85
7.2.1	Shared Library	85
7.2.2	Interactive Tour	85
7.2.3	Collaborative Editor	86
7.2.4	Usability Evaluation	86
A	Figures	87
B	Tables	89
C	Listings	90
D	Bibliography	92

1 INTRODUCTION

1.1 CONTEXT

Typst is a modern, programmable typesetting system and a growing alternative to LaTeX. Its ecosystem is still young, though. There is no guided way to learn it, and collaborating on a document means either self-hosting the tooling or paying for a closed-source service. This thesis builds web tooling to close both gaps. [Section 2](#) examines the problem and the existing landscape in detail. This chapter states the resulting goals and scope.

1.2 GOALS

This thesis addresses both gaps through two complementary web applications and a shared library that powers them. The first application is an interactive learning platform, *Tour of Typst*, that guides users through Typst step by step with an embedded compiler that provides a live preview ([Section 2.2](#)). The second is a collaborative editor that lets multiple authors work on the same Typst document in real time ([Section 2.3](#)). Both build on a shared in-browser Typst engine, the `typst-web` library ([Section 4.2](#)). [Table 1](#) summarizes the three objectives.

OBJECTIVE	DESCRIPTION
Interactive Tour	A browser-based, hands-on tutorial that teaches Typst through curated exercises beside a live editor and preview, deployable as a fully static site.
Collaborative Editor	A real-time, multi-user web editor for Typst projects, an open-source alternative to the closed-source <code>typst.app</code> [1] , with authentication, project management, and shared assets.
<code>typst-web</code> library	A reusable in-browser Typst editing core (compilation, IDE features, and formatting) that both applications build on, published on npm for the wider community.

Table 1: Primary objectives of the thesis

1.3 SCOPE AND DELIMITATIONS

The work covers the design, implementation, and evaluation of the two applications and the shared library described above. The following are explicitly out of scope:

- **No changes to the Typst language or compiler.** The projects build on the existing Typst crates and packages rather than extending the language itself.
- **Browser-only.** Both tools target the web. Native desktop and mobile clients are not pursued.
- **Single-instance deployment.** The Collaborative Editor is delivered as one self-hosted instance, not a multi-tenant, horizontally scaled service.
- **Foundational coverage.** The Tour teaches the core of Typst rather than every package and advanced feature.

1.4 THESIS STRUCTURE

The remaining chapters are organized by engineering phase, and each phase covers the shared `typst-web` foundation first and then the two applications, rather than repeating that arc separately per tool:

- **State of the Art** develops the motivation against the existing landscape of interactive learning and collaborative editing.
- **Requirements** specifies the functional and non-functional requirements of each application.
- **Design** presents the system architecture, foundation first.
- **Implementation** details how each part is built, including the shared quality-assurance baseline (tooling, testing, continuous integration) and how each tool is deployed and published.
- **Evaluation** reports the post-build user survey and the results.
- The closing **Conclusion** reflects on the process and outlines future work, and the project-management retrospective is collected in the appendix.

The architecture is introduced at progressively finer levels of detail across these phases, drawing on the C4 model's levels of abstraction [2]. Requirements opens each application with a system context, the system as a single box among its users and external dependencies. Design zooms in to the containers, the applications, the shared `typst-web` library, and the backing services. Implementation zooms in once more to the components and code that realize them. The reader can therefore follow one architecture from the outside in, rather than meeting it all at once.

2 STATE OF THE ART

2.1 THE ONBOARDING PROBLEM

Typst is a modern typesetting system that pairs a clean markup syntax with a real scripting language, but that power comes with a learning curve. The official documentation is thorough and well-organized as a reference, yet it follows a traditional text-first model. To try any example, a newcomer must switch between the documentation and a separate editor, either the official Typst web app or a local installation. The web app is a capable editor but offers no guided learning path, and community resources such as video tutorials and forum posts vary widely in quality and ask the reader to consume rather than practice. The result is a fragmented first experience, with nowhere to read an explanation and immediately act on it in one place.

2.2 INTERACTIVE LEARNING AND THE TOUR

Interactive, hands-on onboarding has become the dominant approach for introducing developers to new ecosystems. Research shows that active engagement, specifically constructing knowledge through direct experimentation, leads to significantly better learning outcomes than passive consumption [3]. Platforms such as *Tour of Go* [4] and *Tour of Rust* [5] put this into practice, letting learners experiment directly in the browser without any local setup, with a compiler or interpreter embedded alongside curated exercises.

No dedicated, structured learning platform for Typst currently exists that combines a step-by-step course with an in-browser, live-compiling environment. Closing that gap is the initial idea of this thesis, an Interactive Tour that teaches Typst through curated exercises next to a live editor, so that reading and doing happen in the same place. The pre-build survey (Section 2.2.1) confirms both the interest in Typst and the demand for exactly this kind of guided entry point.

2.2.1 PRE-BUILD USER SURVEY

2.2.1.1 OVERVIEW

This survey probed current document-writing habits and willingness to adopt Typst for a thesis. Although traditional tools dominate past experience, respondents show a clear appetite for Typst in an academic context.

2.2.1.2 CURRENT TOOL EXPERIENCE VS. THESIS PREFERENCE

Past experience is dominated by traditional word processors: 93% of respondents have used Microsoft Word, followed by Markdown (73%) and LaTeX (60%), with Typst the lowest at 27%.

Asked which tool they would actually choose for a thesis, the order reverses. Word drops to 27% while Typst climbs to 80%, even ahead of LaTeX (67%).

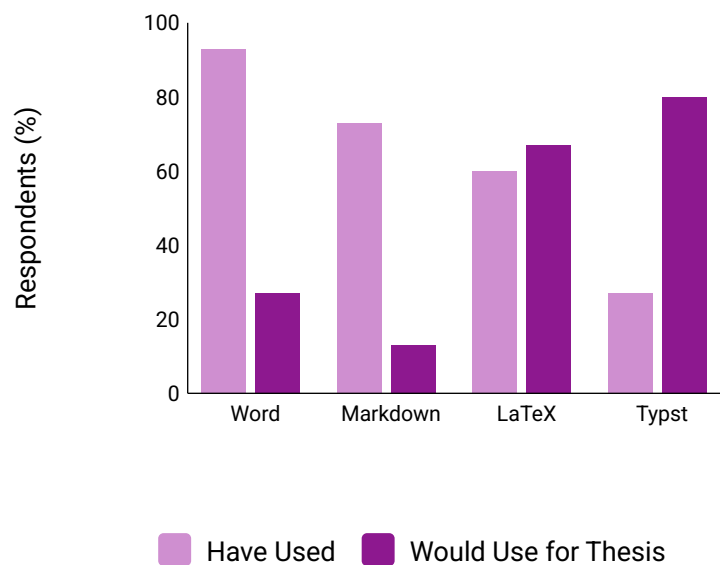


Figure 1: Prior tool experience versus desired tool for thesis writing

2.2.1.3 MATHEMATICAL REQUIREMENTS

Math support matters, but its depth varies: 60% need only simple formulas and variables, 7% are power users who rely on advanced constructs like matrices, and the remaining 33% write mostly prose. Advanced math is therefore a minority need, though not a negligible one.

2.2.1.4 FAMILIARITY AND LEARNING INTEREST

Interest far outruns proficiency. Nearly 93% want to learn Typst, but only 7% are comfortable with scripting and custom functions and 27% have built a basic document. The

largest group (40%) has seen examples without writing any markup, and 27% had not heard of Typst before.

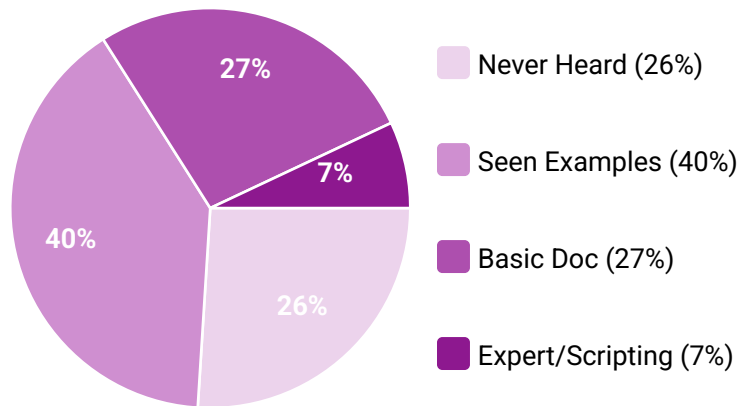


Figure 2: Current familiarity levels with Typst

2.2.1.5 ADOPTION REQUIREMENTS

Adoption depends on the right support. 53% would try Typst immediately, while the other 47% want resources first: 27% want both an OST-specific template and a guided tour, and 20% the template alone. Neither “tour only” nor “wouldn’t use Typst” drew any support. A template is therefore the decisive factor, with a guided tour as valuable reinforcement.

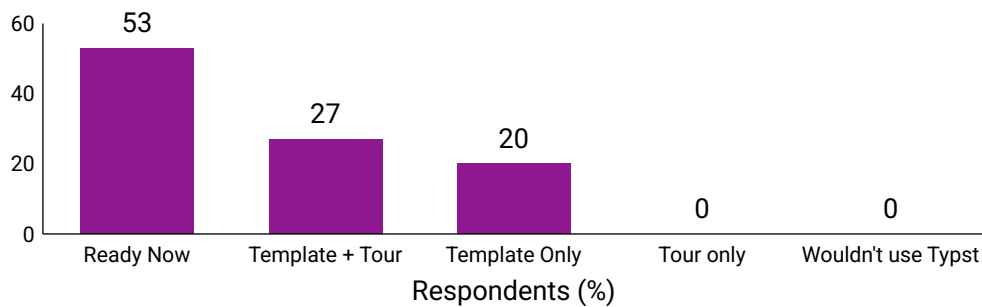


Figure 3: Support requirements for adopting Typst in a thesis project.

2.2.1.6 CONCLUSION

The survey points to clear demand for an official OST Typst template paired with a short guided introduction. Together they provide the low-friction entry point that the majority of interested students need.

2.3 COLLABORATIVE EDITING AND THE EDITOR

Learning to write Typst is only the first step. Once authors are productive, they need to work together, and that is where the next gap appears.

Writing Typst can be daunting for people used to a word processor like Microsoft Word, since they must learn a new way of typesetting, how to compile a document, and a new editor. LaTeX faces the same hurdle, and Overleaf [6] answers it with a collaborative online editor that removes the need to install the TexLive suite locally and makes group work straightforward.

Typst has an equivalent in `typst.app` [1], but it is closed source and paywalls certain features. The open-source, AGPL-licensed TeXlyre [7] covers both Typst and LaTeX, yet its interface is noticeably rougher than `typst.app`. This motivates the second tool of the thesis: a Collaborative Editor that brings real-time, multi-user editing to an open-source tool with a user experience on par with the commercial option.

2.4 A SHARED FOUNDATION

The Tour of Typst (Section 2.2) and the Collaborative Editor (Section 2.3) both need a fully in-browser Typst editor: highlighting, diagnostics, completion, hover, formatting, and a fast compile-and-render loop with no server round-trip. When this project started, no library tied those pieces together for the web. The Typst compiler, the `tinymist` language server, and the `typstyle` [8] formatter all had WASM builds, but shipped as separate tools.

The `typst-web` library [9] fills that gap. It exposes a single WASM build of the Typst engine behind a framework-agnostic API, drawing language intelligence from Typst’s own `typst-ide` rather than `tinymist`, and adds CodeMirror 6 extensions on top. It is published on npm under the `@vedivad/` scope so other web projects can adopt it directly. Its architecture is described in Section 4.2.

3 REQUIREMENTS

This chapter specifies the requirements for the two applications. Both build on the same in-browser editing core from the shared `typst-web` library (Section 4.2): client-side WASM compilation, a live preview, language intelligence (completion, hover, diagnostics), and `typstyle` formatting. The library has no requirements of its own. Its scope derives from the applications, so the sections below state only what each adds.

3.1 TOUR OF TYPST

The Interactive Tour is a single-user application that runs entirely in the browser, and its system context (Figure 4) sets the boundary that the requirements below specify. It has no backend of its own, as the lessons ship as a static site whose only external dependency is the Typst Universe, from which package imports are resolved while a lesson compiles. The requirements are therefore completely client-side, covering in-browser editing and compilation, lesson navigation, and local persistence, with no server behaviour to define.

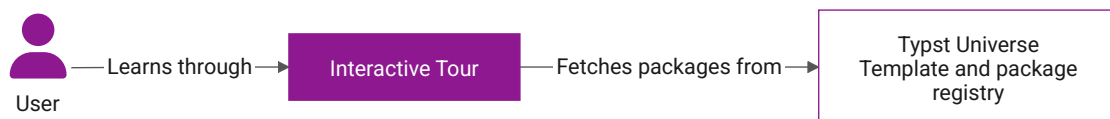


Figure 4: System context of the Interactive Tour

3.1.1 FUNCTIONAL REQUIREMENTS

ID	REQUIREMENT	PRIORITY
Content Rendering		
T-FR-1.1	The system renders tutorial instructions from local Markdown files.	Must
T-FR-1.2	The system provides a split-pane interface with draggable, resizable panes.	Must
Tour Navigation		
T-FR-2.1	The system presents the lessons as an ordered sequence of parts and chapters.	Must

Requirements

ID	REQUIREMENT	PRIORITY
T-FR-2.2	The system lets the user move between lessons and address each lesson by a stable URL, so a lesson can be linked and reloaded directly.	Should
Editing Core		
T-FR-3.1	The Tour embeds the typst-web editing core (compilation, live preview, and language intelligence) described at the start of this chapter, exposed as the editor and preview panes of the lesson view.	Must
Exercise Management		
T-FR-4.1	The system includes a “Show Solution” action that toggles between the user’s code and the solution.	Should
T-FR-4.2	The system runs <code>typstyle</code> via WASM to format the editor content when the “Format” action is triggered.	Should
T-FR-4.3	The system lets the user reset a lesson, or all lessons, to its starting template.	Should
State Persistence		
T-FR-5.1	The system persists the user’s per-lesson editor edits and selected language to <code>LocalStorage</code> .	Must
T-FR-5.2	The system restores a lesson’s saved edits when it is reopened.	Must
Localization		
T-FR-6.1	The system supports multiple languages.	Should
T-FR-6.2	The system lets the user switch language and remembers the choice across visits.	Should

Table 2: Functional requirements of the Interactive Tour

3.1.2 NON-FUNCTIONAL REQUIREMENTS

ID	REQUIREMENT	PRIORITY
Performance		
T-NFR-1.1	The document preview updates within 100 ms of the debounce timeout on standard desktop hardware.	Should
T-NFR-1.2	The system lazy-loads WASM modules so the tutorial text remains non-blocking during initialization.	Should
Portability & Deployment		
T-NFR-2.1	The application is a fully static bundle capable of being hosted on GitHub Pages.	Must
T-NFR-2.2	All computational logic (compilation, formatting, linting) occurs strictly within the client's browser context.	Must
User Experience		
T-NFR-3.1	The interface is fully functional without server-side dependencies or user authentication.	Must
Availability		
T-NFR-4.1	The application is accessible 24/7 via a public URL without requiring any login credentials.	Must

Table 3: Non-functional requirements of the Interactive Tour

3.2 COLLABORATIVE EDITOR

The Collaborative Editor lets several users write a Typst document together in real time, and its system context (Figure 5) frames the requirements that follow. Beyond the in-browser editing core, it spans a client, a server, and persistent storage, and it depends on two external services. A third-party identity provider (GitLab, GitHub, or Google) for authentication and the Typst Universe for templates and packages. The requirements below accordingly reach past editing to authentication, real-time synchronization, access control, and storage across that boundary.

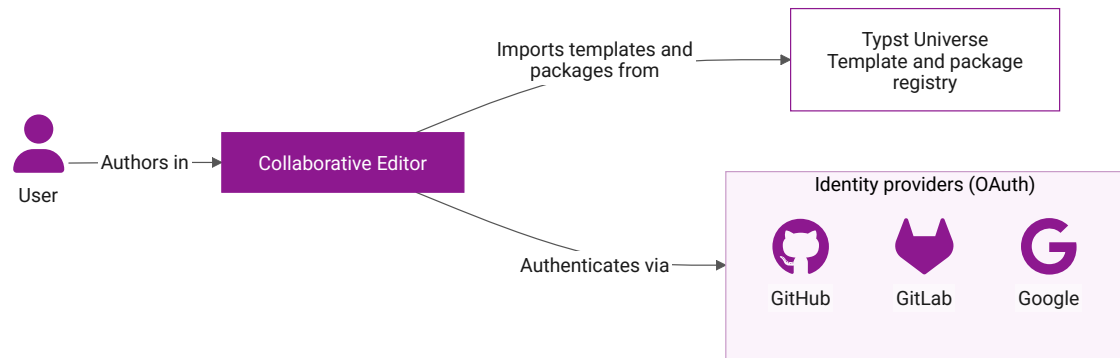


Figure 5: System context of the Collaborative Editor

3.2.1 FUNCTIONAL REQUIREMENTS

ID	REQUIREMENT	PRIORITY
Authentication		
C-FR-1.1	The system supports OAuth login via GitLab, GitHub, and Google.	Must
Real-Time Collaboration		
C-FR-2.1	The system synchronizes concurrent edits across all connected clients using CRDT-based conflict resolution provided by Yjs [10].	Must
C-FR-2.2	Each collaborator's cursor position and display name is visible to all other users currently editing the same document.	Should
C-FR-2.3	If a collaborator deletes a file that another user currently has open, the affected user is automatically redirected to the project's main file and notified that the file was deleted by a collaborator.	Should
C-FR-2.4	The same redirect and notification behavior applies when a folder containing the currently open file is deleted.	Should
Invitation & Access Control		
C-FR-3.1	Authenticated users can generate a shareable invite link for a project, configurable as either read-only or edit access.	Must
C-FR-3.2	Invite links expire within the span of a week and can be revoked by the user who created them at any time.	Should

ID	REQUIREMENT	PRIORITY
C-FR-3.3	The owner of a project can edit access permissions for existing members, including the ability to revoke access for specific users.	Should
Project Dashboard		
C-FR-4.1	Authenticated users are presented with a dashboard that lists all their projects, each accompanied by a rendered preview of the document's first page.	Must
Editing Core		
C-FR-5.1	The editor embeds the typst-web editing core (compilation, live preview, and language intelligence) described at the start of this chapter, operating on multi-file projects.	Must
Project Templates		
C-FR-6.1	When creating a new project, the user can choose from a catalog of templates sourced from the Typst Universe.	Should
C-FR-6.2	Selecting a template populates the new project with its text files, binary assets, and compile entry, so the project compiles without further setup.	Should
Binary Assets		
C-FR-7.1	Users with edit access can upload binary files (images, fonts, data) into a project, where they become referenceable from Typst source by path.	Must
C-FR-7.2	Asset references are part of the synchronized project state, so that all collaborators see and can use them without re-uploading.	Must
C-FR-7.3	The size of uploaded assets is limited to 10 MB, with a higher 30 MB limit for fonts to accommodate large font files.	Should

Table 4: Functional requirements of the Collaborative Editor

3.2.2 NON-FUNCTIONAL REQUIREMENTS

ID	REQUIREMENT	PRIORITY
Real-Time Performance		

Requirements

ID	REQUIREMENT	PRIORITY
C-NFR-1.1	Remote changes from collaborators are applied and visible to all connected clients with a maximum end-to-end latency of 200 ms, measured under a network connection with at least 1 Mbps bandwidth and a round-trip time of no more than 100 ms to the server.	Must
Fault Tolerance		
C-NFR-2.1	Temporary network interruptions are handled transparently by the Yjs synchronization layer. No data is lost during a brief disconnection and the editor state reconciles automatically upon reconnection.	Must
Security		
C-NFR-3.1	All OAuth flows include a <code>state</code> parameter to mitigate Cross-Site Request Forgery (CSRF) attacks.	Must
C-NFR-3.2	Invite link tokens are generated using a cryptographically secure random source (minimum 128 bits of entropy) and are long enough to be practically unguessable.	Must
C-NFR-3.3	The Hocuspocus server validates a user's session before allowing them to join a document room.	Must
Usability		
C-NFR-4.1	A new user can start editing a shared document within three clicks, without requiring any onboarding or tutorial.	Should
Tech Stack Constraints		
C-NFR-5.1	The backend is implemented with Elysia on Bun, using Hocuspocus as the Yjs server provider and Drizzle ORM for database access against PostgreSQL.	Must
C-NFR-5.2	The frontend is built with SolidJS and uses <code>y-codemirror.next</code> as the collaborative editor component.	Must
C-NFR-5.3	Client-server communication uses Elysia Eden for end-to-end type safety.	Must

Table 5: Non-functional requirements of the Collaborative Editor

4 DESIGN

4.1 SYSTEM OVERVIEW

This thesis delivers two user-facing applications on one shared foundation. The first is the Tour of Typst ([Section 4.3](#)), an interactive tour that introduces the Typst language step by step, and the second is the Collaborative Editor ([Section 4.4](#)). Both applications need the same thing at their core, namely a full Typst editing experience that runs entirely in the browser, with syntax highlighting, diagnostics, completion, hover, formatting, and a fast compile-and-render loop. Rather than build that core twice, it is factored out into a reusable library, `typst-web`, that compiles the Typst engine to WebAssembly and exposes it behind a single editor-agnostic API.

The result is a layered design, shown in [Figure 6](#). At its core is the `typsten` module, which bundles the Typst engine, its `typst-ide` language intelligence, and the `typstyle` formatter, all compiled into a single WebAssembly binary. The `typst-web` library (the foundation described in [Section 4.2](#)) wraps that core in a small set of `@vedivad/*` packages, and the two applications depend on those packages rather than on the engine directly.

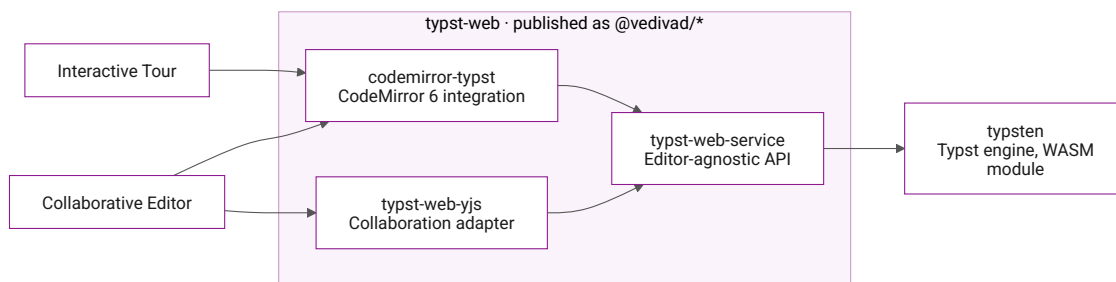


Figure 6: Layered architecture: `typsten` core, `typst-web`, and the two apps

The applications build on this shared core and diverge only where their purposes differ. The Tour of Typst is a fully static single-page application that pairs lessons with the embedded editor and needs no server at all. The Collaborative Editor is a three-tier application that adds real-time multi-user synchronization, authentication, and persistence around the same editing core. The sections that follow describe the foundation first, then each application in turn, so that the shared compilation and editing machinery is explained once and referenced thereafter.

4.2 TYPST-WEB

4.2.1 ARCHITECTURE

A Bun-managed workspace of three npm packages, each scoped to a single concern (Figure 7). The framework-agnostic core stays decoupled from the CodeMirror integration and the optional Yjs adapter stays out of the core.

- `@vedivad/typst-web-service`: Bundles the full Typst stack (compiler, `typst-ide`, `typstyle`) into a single WASM module, called `typsten`. Then puts it behind a `TypstProject` API, so nothing else in the stack touches WASM or workers directly. `typsten` compiles the real Typst engine to WebAssembly, so the browser renders exactly what native Typst would, and its editor intelligence stays in sync with upstream Typst automatically.
- `@vedivad/codemirror-typst`: Connects a CodeMirror instance to the engine, giving the user completions, diagnostics, and hover information as they type.
- `@vedivad/typst-web-yjs`: The collaboration layer. When active, the Yjs document replaces CodeMirror as the source of truth, so peer edits flow in through Yjs and the engine compiles from the shared document state instead.

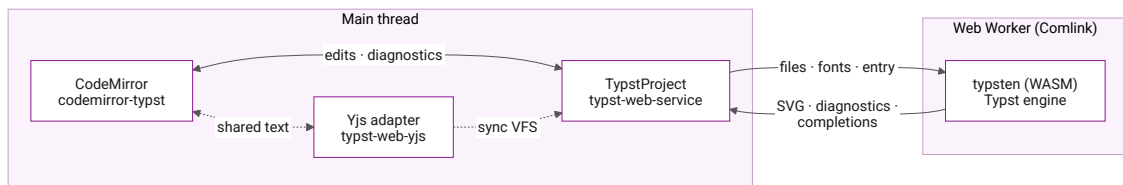


Figure 7: High-level architecture of `typst-web`

The `typsten` module runs in a Web Worker, reached from the main thread over Comlink [11]. Moving compilation off the main thread keeps a heavy compile from blocking the UI, and the worker boundary doubles as an I/O boundary, since all file access stays on the main thread. There the `TypstProject` owns the virtual file system (VFS) and mirrors the current files, fonts, and entry point into the worker, so `typsten` only ever compiles from that in-memory copy. The WASM module therefore performs no I/O of its own. It holds the mirrored files, fonts, and entry point as internal state, but compilation is deterministic over that state: the same inputs always produce the same output.

Both downstream projects in this thesis consume this library. The Tour of Typst loads it directly to drive its in-browser editor, and the Collaborative Editor consumes it through the Yjs adapter so peer changes flow into the local `TypstProject`.

4.3 TOUR OF TYPST

4.3.1 ARCHITECTURE

The defining constraint of the Tour of Typst is that it has no backend. Everything a learner needs ships inside the bundle, and all Typst compilation runs client-side via the `typst-web` library (Section 4.2). This keeps the application deployable as static files and removes any per-user server state. Figure 8 shows the resulting structure.

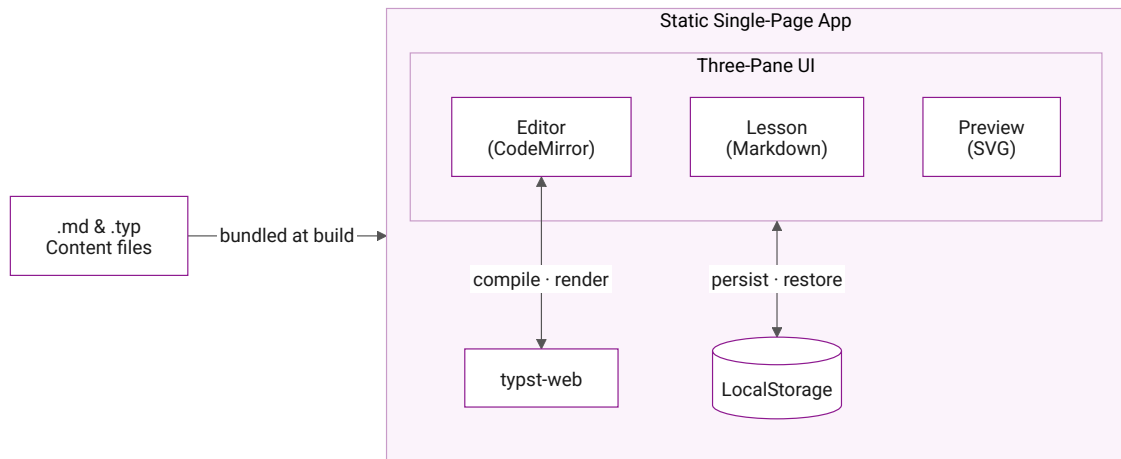


Figure 8: High-level architecture of the Tour of Typst

4.3.2 CONTENT ORGANIZATION

Lessons are treated as data rather than code. Each lesson is a directory holding its prose as Markdown (`index.md`), the editor starter code (`template.typ`), an optional reference solution (`solution.typ`), and any YAML data files it needs. The data files are loaded into the editor’s virtual file system beside the starter code, so a lesson can build a data-driven document.

```

content/<locale>/<part>/<chapter>/
├── index.md
├── template.typ
├── solution.typ
└── data.yaml
  
```

Listing 1: Content directory layout of a single lesson

The whole tour is bundled at build time, so nothing is fetched at runtime. Chapters are grouped into ordered parts. Each locale is an independent tree rather than a translation overlay, and the set of locales is discovered from those trees. A switcher lets the learner change language at any time, with the choice remembered across visits.

4.3.3 NAVIGATION

The two-level tree of parts and chapters (Section 4.3.2) is flattened into a single ordered sequence of lessons. The parts remain for orientation, but the reader moves along one linear path through that sequence, which is what the previous and next controls follow (Figure 9).

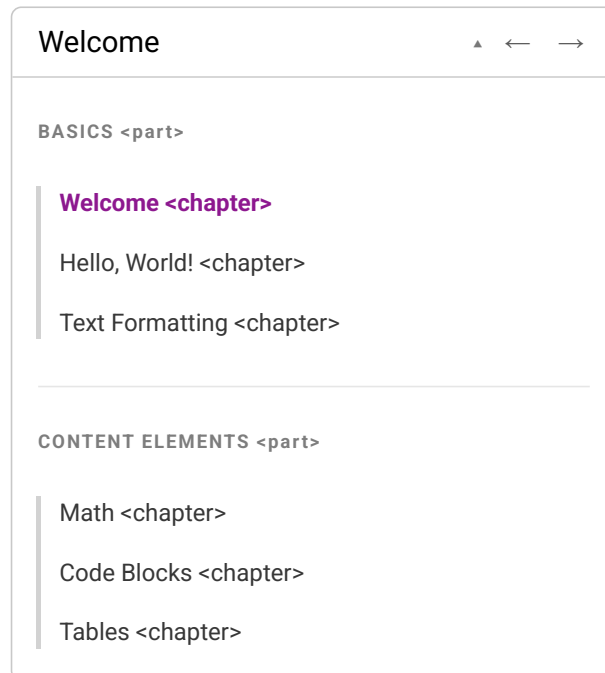


Figure 9: Mockup of the chapter navigation

Each chapter has a stable identity of its own, independent of where it sits in that order. A lesson can therefore be linked or bookmarked and still resolve after the tour grows or its chapters are reordered, which matters for a body of content meant to expand over time.

Language is deliberately kept out of a lesson's address. The active locale is a remembered setting rather than part of the link, so a shared link stays meaningful regardless of the recipient's language and opens the same lesson in whichever locale they are reading.

4.3.4 LESSON EXPERIENCE

Each lesson is presented in a three-panel "read, try, see" layout (Figure 10): the prose on the left, a live editor and an SVG preview on the right. The editor is seeded from `template.typ`, and the preview recompiles as the learner types.

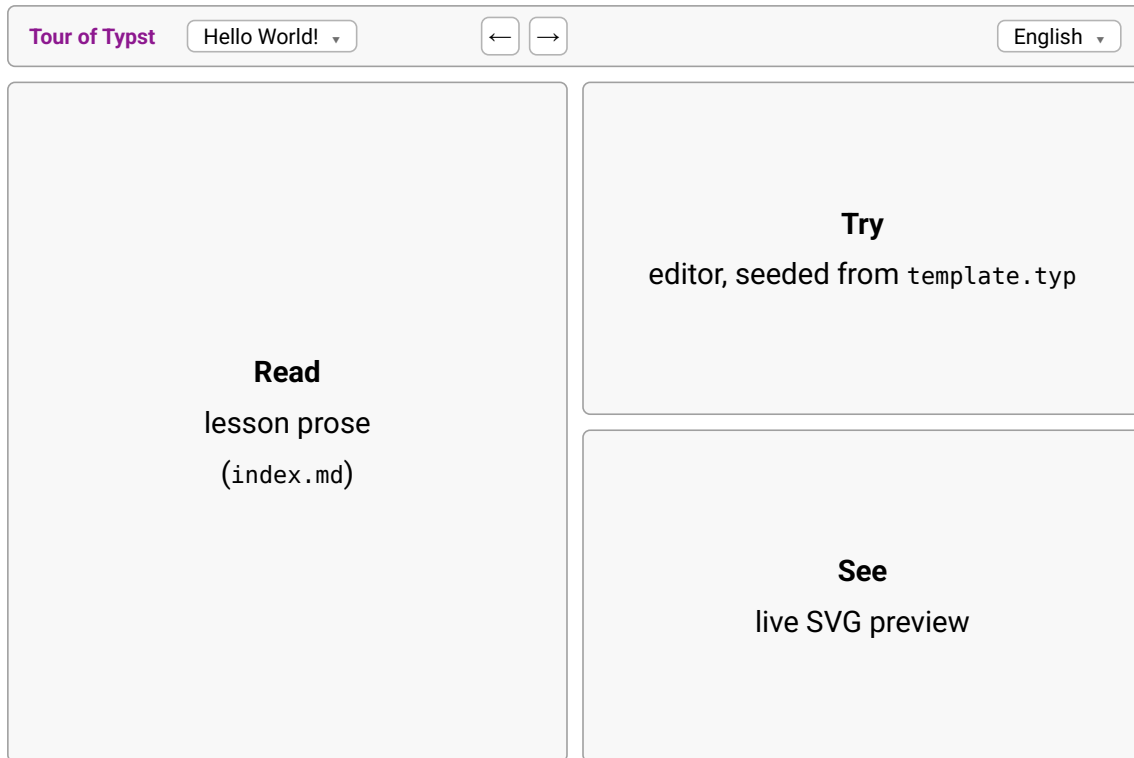


Figure 10: Mockup of the three-panel “read, try, see” lesson layout

Because it is the full `typst-web` editor (Section 4.2), learners get real completions, diagnostics, and hover while they type, not a cut-down sandbox. The editing surface is deliberately narrow, though: each lesson is a single Typst file, with any data files mounted read-only, so there is no file tree to manage as there is in the Collaborative Editor (Section 4.4).

4.3.5 SOLUTIONS AND PROGRESS

The reference solution stays hidden until the learner asks for it. Revealing it shows `solution.typ` with the lines that differ from the starter template highlighted through a line diff, so the changes the lesson introduces stands out.

The Tour never grades an attempt, it only offers the solution for comparison. The learner moves through the ordered chapters one at a time, and because each chapter’s edits are saved to `LocalStorage`, that work survives reloads and navigation. A learner can reset the current lesson to its starter template at any time, or clear every saved edit at once.

4.4 COLLABORATIVE EDITOR

4.4.1 ARCHITECTURE

Figure 11 shows the high-level structure. Unlike the Tour of Typst, the Collaborative Editor needs a persistent server, because projects, collaborator presence, and binary assets must outlive any browser session, and document state must converge across concurrent clients. The stack has three tiers, namely a SolidJS [12] frontend, an Elysia [13] backend on Bun [14], and a PostgreSQL [15] database.

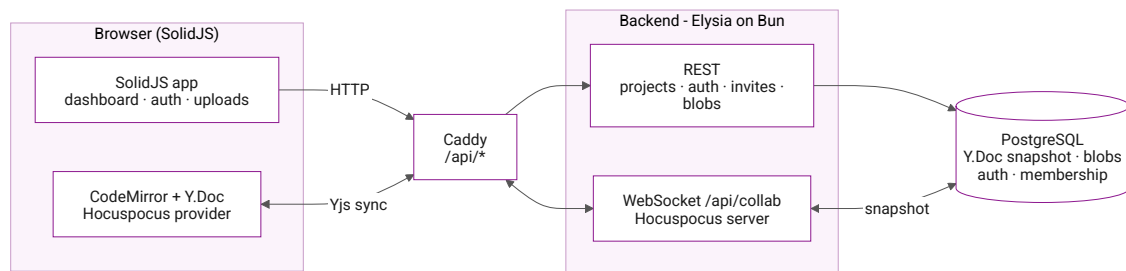


Figure 11: High-level architecture of the Collaborative Editor

The frontend is a single-page application that reaches the backend over two channels. Project lifecycle, authentication, membership, and asset uploads use a conventional REST API, while the collaborative document itself rides a persistent WebSocket. A single Elysia process on Bun serves both, and because they share the `/api/*` namespace, one Caddy reverse proxy rule fronts them in production.

4.4.2 ACCESS CONTROL

Identity is delegated entirely to third-party providers. A user signs in through OAuth with GitLab, GitHub, or Google, so the editor stores no passwords of its own. Within a project, an authenticated user then holds one of three roles. An owner is the user who created the project, an editor may change its content, and a viewer may only read it.

Making the Yjs document the shared source of truth has a direct consequence for authorization. Read-only access cannot be enforced on the REST API alone, since a viewer with an open collaboration socket could otherwise mutate the document and have those edits converge to everyone else. Access is therefore checked on both channels. When a client opens its Hocuspocus connection, the server resolves the caller's project membership and marks the connection read-only for viewers, so the collaboration layer rejects their writes at the source. Editors and owners receive write access. The same membership gates the REST routes, so neither path can be used to bypass the other.

4.4.3 DATA MODEL

The defining property of the data model is that the Yjs [10] document is the source of truth for project content, not the database. Figure 12 shows how the document and the schema divide responsibility.

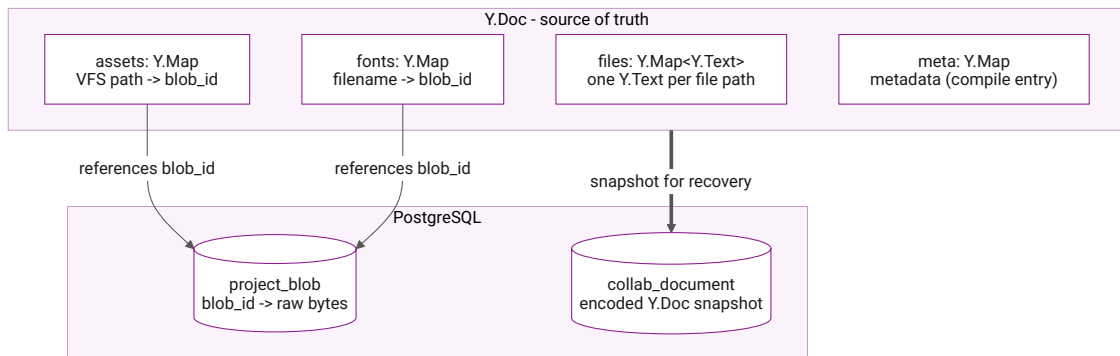


Figure 12: Document data model

Each project is backed by one Y.Doc whose top-level Y.Maps hold text files, asset references, font references, and project metadata such as the compile entry point. Edits flow through a Hocuspocus provider in the browser to the Hocuspocus server bridged onto Elysia's /api/collab WebSocket, which converges the document across clients and persists it.

The shared document is a conflict-free replicated data type (CRDT) [16], a structure in which concurrent edits from different clients converge to the same state without a central coordinator. A client that loses connectivity can keep on editing against its local copy and reconciles automatically on reconnect, with no lost edits and no manual merge. The relational schema only covers the surrounding concerns: authentication, project membership, invite tokens, the raw asset bytes, and a durable encoded snapshot of the Y.Doc for recovery.

4.4.3.1 BINARY ASSETS

Text files live inside the Yjs document, but binary assets such as images and fonts do not belong in a CRDT. Their bytes are instead stored relationally in PostgreSQL [15], and the document holds only references, namely an asset map and a fonts map, each entry pointing to the blob row that carries its bytes (Figure 12). This split raises the question of which store decides when an asset is still in use. The design keeps the Yjs document authoritative even for lifecycle. A garbage-collection step observes those maps and removes any byte row the document no longer references, whether an entry was deleted or overwritten.

4.4.4 LOGIN

Access to the editor begins with a sign-in page (Figure 13). The application keeps no passwords of its own. Instead, it delegates authentication to the OAuth providers GitHub, GitLab, and Google, which cover the accounts students and staff already hold. The page is split between a brand panel and a short list of provider buttons, so signing in is a single click followed by the provider's own consent screen. The button list is dynamic, since the page asks the backend which providers are available and renders only those that have credentials configured, so a deployment can offer a subset without any frontend change.

Every OAuth flow carries a state parameter to guard against cross-site request forgery, and a successful callback establishes the session that the dashboard and the collaborative editor rely on. Sign-in also preserves intent, so a visitor who follows a deep link, such as a project invitation, is returned to that destination afterwards rather than dropped on the dashboard. Only same-origin destinations are honored, so the redirect cannot be turned into an open redirect to an external site. Because identity comes from the provider, each user has a stable account that the editor reuses to attribute edits and to derive the per-user presence color (Figure 17).

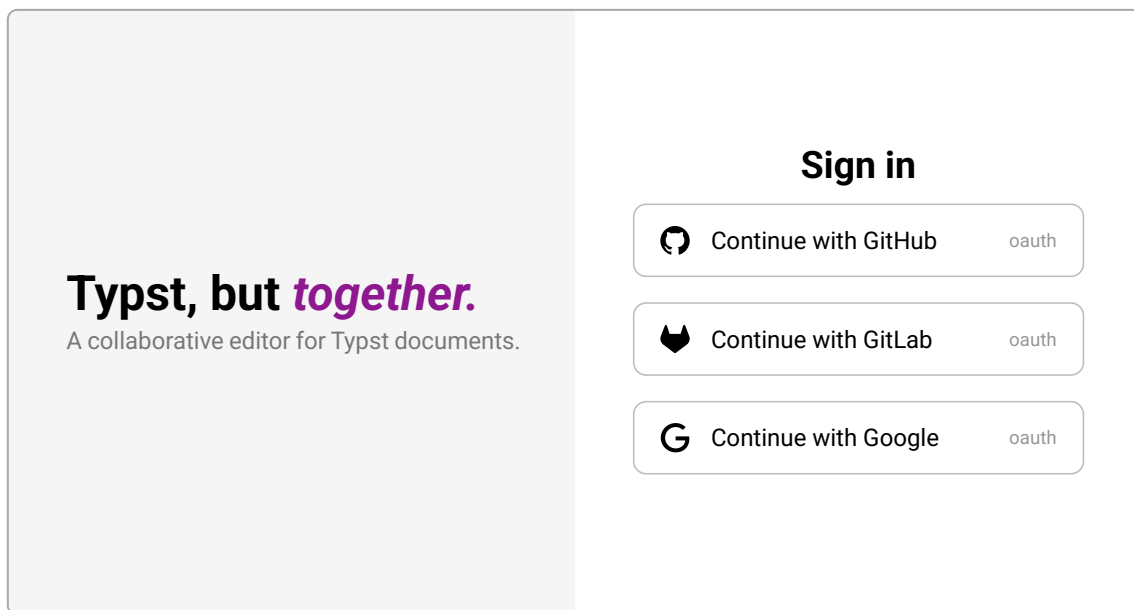


Figure 13: Mockup of the sign-in page with OAuth providers

4.4.5 DASHBOARD

After signing in, the user lands on a dashboard that lists every project they own or have been invited to (Figure 14). Each project is shown as a card carrying a rendered preview of its document's first page, so the user recognizes a project by its appearance rather than by its name alone. Alongside the preview, the card states when the project was created

and when it was last edited, and a per-card menu exposes rename, share, and delete actions.

A top bar spans the dashboard with the application brand, a search field that filters the cards by name, and the user's account menu. The two primary entry points sit above the grid: *New project* opens the creation dialog (Figure 15), while *Join project* accepts an invite link from another user. The first-page preview is generated by the same client-side compiler used in the editor, so the dashboard needs no server-rendered thumbnails.

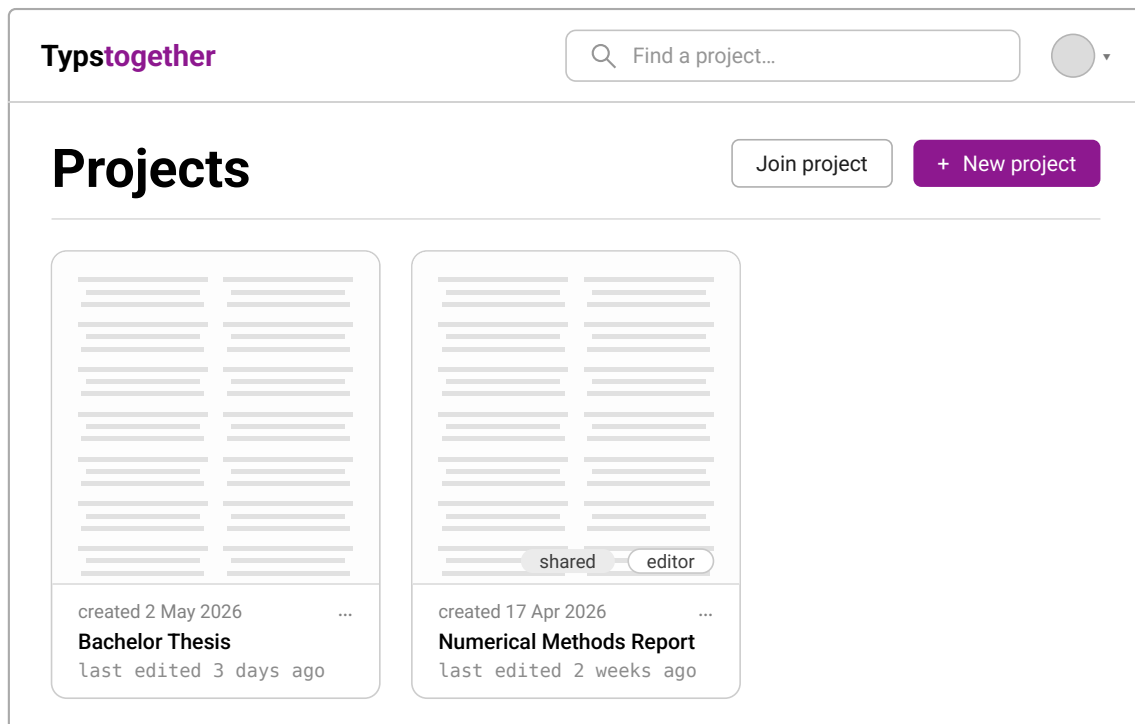


Figure 14: Mockup of the project dashboard

4.4.6 PROJECT CREATION

Creating a project optionally starts from a template (Figure 15). Choosing one from the Typst Universe catalog materializes it directly into the project's initial `Y.Doc`, where each text file becomes a `Y.Text`, each binary becomes a blob referenced from the asset map, and the template's declared entry is written to the metadata map. This seeded document is encoded and persisted up front, so the first collaborator to connect already has a project that compiles.

New project

Name

My document

Template

all letter report resume thesis

Search templates...

blank

Blank

Empty document

formal-letter

Business letter

conference-paper

Conference paper

modern-resume

Curriculum vitae

technical-report

Technical report

clean-thesis

Academic thesis

[Browse Typst Universe](#)

Cancel

Create

Figure 15: Mockup of the new-project dialog

Without a template the server seeds nothing. The project starts empty, and the first client to open it creates a single empty `main.typ` so there is always a file to edit. Either way the result is an ordinary `Y.Doc`, so a template is just a pre-populated document, not a separate scaffolding path.

4.4.7 EDITOR LAYOUT

The editor presents a project as a three-pane workspace (Figure 16): a collapsible sidebar on the left, the text editor in the middle, and a live preview on the right, separated by draggable handles. A slim activity rail switches the sidebar between the file tree, fonts, search, and compiler diagnostics, and a top toolbar carries the project name and role, the avatars of present collaborators, and the connection status.

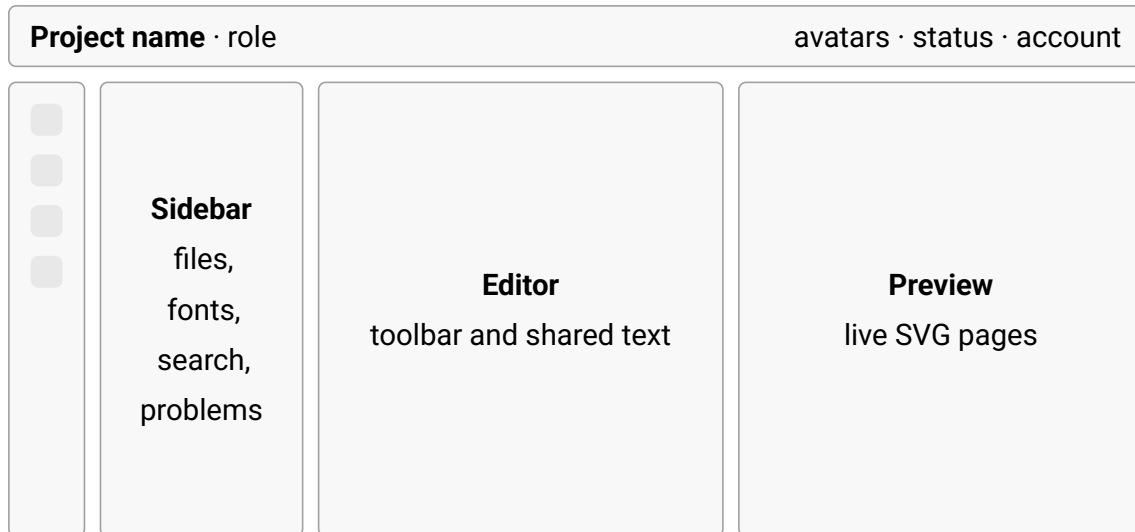


Figure 16: Mockup of the editor layout

Unlike a single tour lesson ([Section 4.3](#)), a project is a tree of files with binary assets, so the workspace foregrounds the file management that the tour deliberately omits.

4.4.8 PRESENCE

A collaborative editor must show who else is present and where their cursors are ([Figure 17](#)). This presence information is fundamentally different from project content.

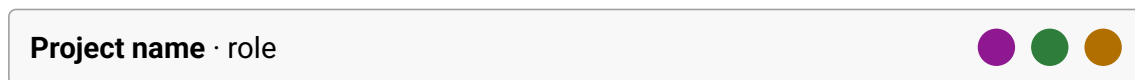


Figure 17: Mockup of the editor's presence indicators

It is per-session, changes constantly, and is worthless the moment a collaborator disconnects. The design therefore keeps presence off the persisted document and carries it on Yjs's separate awareness channel that propagates each client's cursor, selection, and identity to its peers without ever being written to the `Y.Doc` or the database. Each collaborator is given a stable color derived from their user identity, so the same person appears consistently across every client. When a tab closes its awareness state expires on its own, and the matching cursor and avatar disappear without any cleanup.

5 IMPLEMENTATION

5.1 QUALITY ASSURANCE

The two applications and the shared `typst-web` library follow one quality-assurance setup, the same runtime, formatter, linter, and pipeline shape across all three codebases. This section describes that shared baseline once and notes where a codebase diverges, in the test runner ([Section 5.1.5.1](#)) and, most visibly, in how each is deployed ([Section 5.2](#)).

5.1.1 DEVELOPMENT ENVIRONMENT

Every part of the project runs on the Bun runtime and toolchain, which bundles the package manager, script runner, and test runner into a single binary. For most work a contributor therefore needs only Bun installed to install dependencies, start a dev server, run the tests, or lint the code.

The shared `typst-web` library is itself a Bun workspace monorepo, combining the Rust WebAssembly core (`typsten`) with the three published packages that build on it (`typst-web-service`, `typst-web-yjs`, and `codemirror-typst`). `bun install` wires the workspace and `bun dev` runs the packages in watch mode. Its one prerequisite beyond Bun is the Rust and `wasm-pack` toolchain that compiles `typsten` to WebAssembly ([Section 5.3.1.2](#)). The engine is built once with `bun run build:wasm`, and from there the workflow is plain Bun like the rest.

The Tour of Typst is a single Vite project, where `bun install` followed by `bun dev` is enough for a hot-reloading editor served against the local lesson tree, with no backend to run.

The Collaborative Editor is a Bun workspace monorepo of three packages, `frontend`, `backend`, and a shared package for the types exchanged between client and server ([Section 5.5.7](#)). One `bun dev` starts the Elysia backend and the Vite frontend together through `concurrently` [17], while its only stateful dependency, PostgreSQL, is provided by the Docker Compose stack so that no local database install is required. The same stack, extended with the Caddy reverse proxy, is what runs in production ([Section 5.2](#)).

5.1.2 GUIDELINES

All three repositories share one contribution workflow, and it is enforced in continuous integration rather than left to discipline. Changes reach `main` only through pull requests, and a small set of rules keeps the history consistent:

- **Conventional commit titles.** A pull-request title must match `<type>: <Description>`, where `type` is one of `build`, `chore`, `ci`, `docs`, `feat`, `fix`, `perf`, `refactor`, or `test`, and the description begins with a capital letter. A dedicated workflow fails the check for titles that do not.
- **Automatic labelling.** A labeler workflow reads that same enforced prefix and applies a matching label to the pull request. Deriving the label from the title keeps categorization consistent and the history filterable.
- **Squash merges.** Pull requests are squashed, and the merge commit inherits the pull-request title, so the convention also shapes the commit history on `main`.

Style itself is never debated in review: a single formatter and a single linter decide it, and both run in CI ([Section 5.1.3](#)).

5.1.3 STATIC ANALYSIS AND CODE STYLE

The codebase is held to one linter and one formatter. `bun run lint` is the single entry point, as it first fails on any unformatted file and then runs the lint pass, so a green lint job already implies a formatted tree.

Linting uses `oxlint` with type-aware analysis switched on. This lets rules that need type information, such as floating-promise and unsafe-argument checks, run beside the purely syntactic ones, recovering much of what a separate TypeScript compiler [\[18\]](#) pass would catch in a fraction of the time. The shared configuration enables the `import`, `promise`, `react`, `jsx-ally`, `typescript`, `unicorn`, and `oxc` plugin sets, which together form a strict rule set.

The Typst sources, including this thesis and the Tour's lesson templates, are formatted with `typstyle`, which the pipeline checks in CI.

Prose is also subject to review. The spell checker codebook [\[19\]](#) examines code, comments, and Markdown against a dictionary, while project-specific terms, such as tool names and framework keywords, are excluded through a dedicated configuration file.

5.1.4 DEPENDENCY MANAGEMENT

A project's dependency tree is one of its largest attack surfaces. A single compromised transitive package can exfiltrate secrets or corrupt the shipped artifact, and such supply-chain attacks are common in the npm ecosystem. Two measures reduce that exposure, and both rest on the same idea, namely that a brand-new package version should never be installed immediately.

The first measure delays installation at the runtime level. The `bunfig.toml` file sets `minimumReleaseAge` to 604,800 seconds, equivalent to one week, so that Bun refuses to install any package version younger than seven days. Because most malicious releases are detected and withdrawn within hours of publication, this waiting period prevents the typical attack from taking effect before the package is ever resolved, and it requires no action from a maintainer (Figure 18).

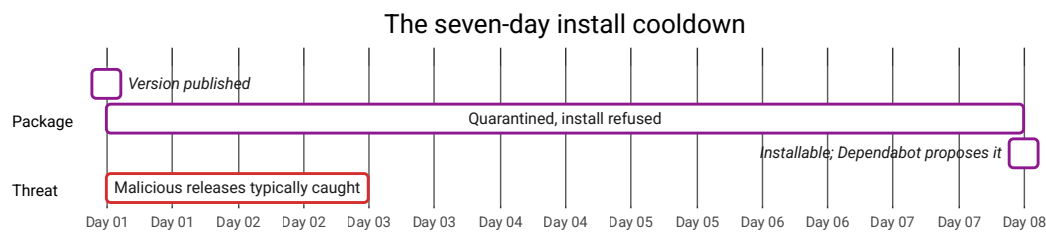


Figure 18: A week-long hold outlasts most malicious-release detection

The second measure automates updates while remaining equally cautious. Dependabot [20] opens one grouped pull request per week for the Bun packages and another for the GitHub Actions, in each case with a matching seven-day waiting period so that it never proposes a freshly published release. Every update arrives as an ordinary pull request, labelled dependencies and titled with the `build` prefix, so that it passes through the same checks as any other change (Section 5.1.2, Section 5.1.3) before it can be merged. Routine upgrades therefore remain current without requiring manual effort, while the waiting period keeps the project from adopting the very latest releases.

5.1.5 TESTING

5.1.5.1 UNIT TESTING

Tests run on Bun’s built-in test runner through `bun test`, with no separate framework to configure. The one exception is the shared `typst-web` library, whose packages were established before this convention and run their suites on Vitest rather than `bun test`. The suites deliberately target pure logic and the service boundary rather than the user interface, where the return per test is highest and the tests stay fast and deterministic.

In the Tour, the tested units are the parts that are easy to get wrong yet invisible at a glance, namely the content loader that indexes lessons by locale and key, and the line diff that drives the solution highlight ([Listing 37](#)).

In the Collaborative Editor, every backend module is covered at two levels. A service test exercises the business logic directly against a database in a rolled-back transaction, and a route test drives the same logic through its HTTP surface, so both the rules and their wiring are checked. Authorization, invite redemption, blob garbage collection, and document persistence each carry their own suite. On the frontend, the pure helpers, path manipulation, relative-time formatting, and the file-tree derivation ([Listing 48](#)), are tested in isolation.

The editor’s formatting actions ([Section 5.5.5.5](#)) form the widest frontend suite, and they test cleanly because both their input and their output are just text with a marked selection. Rather than drive a real editor, each test builds a minimal fake view from a string in which a down-arrowhead marker (v) stands for the selection. A pair at one spot is a collapsed cursor, a pair around text is a range, and several pairs set up the multi-cursor cases. After the action runs, the resulting document is rendered back into the same notation and compared, so an assertion reads as a before-and-after picture of where the selection ends up ([Listing 2](#)). The approach is inspired by the Zed [\[21\]](#) editor’s codebase, which tests its editor actions with the same style of marked-range assertions.

```
1 const v = mockView("hvello");
2 wrapSelection(v, "*");
3 expect(markDoc(v)).toBe("h*vello");
```

Listing 2: Formatting actions tested with a before/after marker notation

For the Collaborative Editor, continuous integration runs the suite with coverage (`bun test --coverage`). A script renders the result as a Markdown table with a coverage badge and

posts it as a sticky pull-request comment, updating the same comment on each push, so a change's effect on coverage is visible inline during review (Figure 19).

5.1.5.2 MANUAL MULTI-USER TESTING

Automated suites cover logic and the service boundary, but the collaborative behaviour, two cursors converging on one document, a deletion redirecting the other reader, is judged by watching several users at once. Doing so through the real OAuth flow is awkward, since every test user would need a genuine GitHub, GitLab, or Google account, and juggling those logins across windows is tedious.

A dev-only route removes that friction. `GET /api/dev/sign-in-as?name=Alice` upserts a user by name, creates a Better Auth [22] session, sets the signed session cookie, and redirects to the dashboard, so any incognito window becomes a distinct signed-in collaborator in one request. The route is mounted only from the development entry point and never from the production application, so it cannot ship to a deployed build. With it, a single machine can drive a whole room of collaborators and the convergence and presence behaviour can be observed directly.

5.1.6 CONTINUOUS INTEGRATION

All three repositories run the same shape of pipeline, namely a set of independent checks on every pull request that must all pass before merge, and a deploy workflow that fires on every push to `main` (Figure 19). There is no staging environment, so `main` is production.

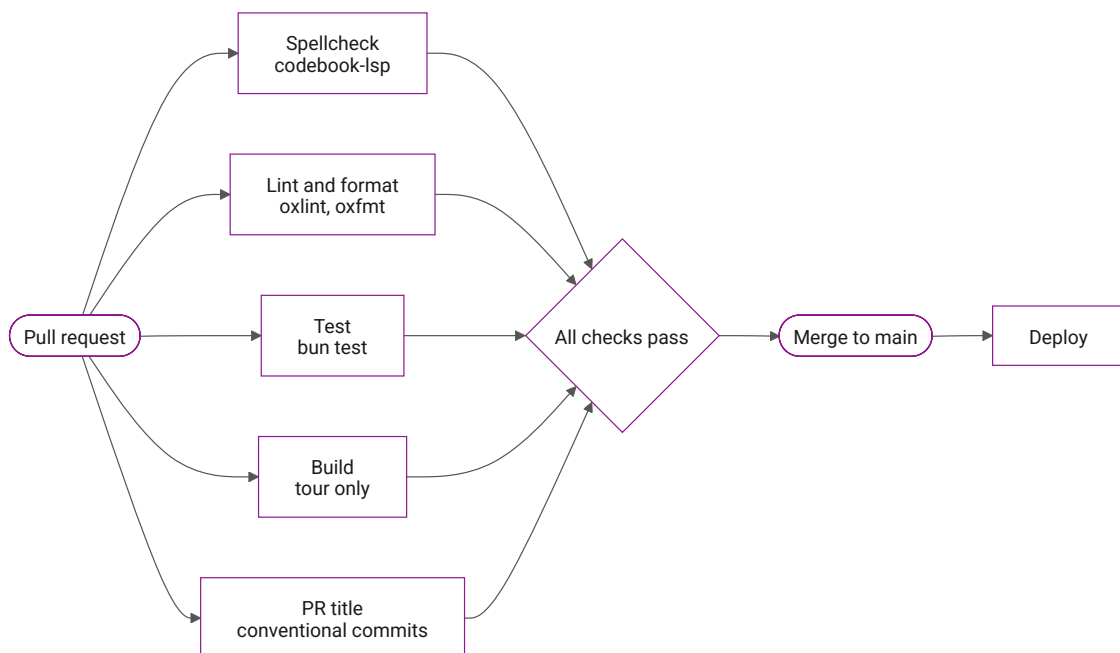


Figure 19: Shared CI/CD: checks gate every PR; merging `main` deploys

The pull-request checks are:

- **Spellcheck** runs codebook spellchecker over the code, comments, and Markdown (Section 5.1.3).
- **Lint** installs dependencies and runs the Oxlint [23] linter with type awareness. The Oxfmt formatter then checks for unformatted files. The Tour additionally checks its Typst sources with typstyle.
- **Test** runs `bun test`. The Collaborative Editor runs it with `--coverage` and posts the report as a sticky pull-request comment (Figure 19).
- **Build** compiles the Tour's static bundle, catching bundler and type errors before they can reach deployment.
- **PR title** validates the Conventional-Commit title (Section 5.1.2).

Deployment differs by target. The Tour builds its static bundle and publishes it to GitHub Pages. The Collaborative Editor deploys to a self-hosted runner, which copies the environment file and rebuilds the Docker Compose stack in place, bringing up PostgreSQL, the backend, and Caddy together (Section 5.2).

5.2 DEPLOYMENT & PUBLISHING

The three products deploy in two shapes. The Tour and the `typst-web` demo are static bundles served from GitHub Pages, while the Collaborative Editor runs as a small server stack. The `typst-web` library itself is published to npm rather than deployed.

5.2.1 TYPST-WEB

The library is published to npm as three packages under the `@vedivad/` scope, namely `typst-web-service`, `typst-web-yjs`, and `codemirror-typst`. The Rust `typsten` engine is not published on its own. It is bundled into `typst-web-service` at build time, so consumers get the WebAssembly module transparently. Each package is ES Modules only and ships only its built `dist/` output, and the libraries it shares with the host application (CodeMirror, Yjs) are declared as peer dependencies so the host provides a single copy.

Versioning runs on Changesets rather than hand-edited numbers. A change is recorded with `bun run changeset`, which captures the bump level and a changelog summary. The three `@vedivad` packages form a fixed group, so they always share one version, published with public access.

Releasing is automated by a workflow on every push to `main` (Listing 3). It builds the WebAssembly engine and the packages and runs the tests, so the artifacts are present

before any publish, then defers to the Changesets action. While changesets are pending, it opens a “Version Packages” pull request that applies the bumps and rewrites the internal workspace: ranges to real versions. Once that pull request merges, the same step runs `changeset publish` to release to npm with provenance.

```

1 # changesets pending: open a "Version Packages" PR; none: publish to npm
2 - uses: changesets/action@v1
3   with:
4     version: bun run version-packages # changeset version
5     publish: bun run release          # changeset publish
6   env:
7     NPM_CONFIG_PROVENANCE: "true"

```

Listing 3: A workflow step opens the version PR or publishes

The interactive demo, by contrast, is deployed rather than published. A push to `main` builds the WebAssembly engine and the packages, builds the demo against them, and deploys the result to GitHub Pages under its own base path to showcase the editor.

5.2.2 INTERACTIVE TOUR

The Tour is a fully static single-page application with no backend (Section 4.3), so deploying it is just serving files. A push to `main` builds and prerenders the bundle and publishes it to GitHub Pages, with the site’s base path injected through `VITE_BASE_URL` so assets resolve under the project subpath.

5.2.3 COLLABORATIVE EDITOR

The Collaborative Editor needs a persistent server, so it ships as a Docker Compose stack that comprises a PostgreSQL database, the Elysia backend on Bun, and a Caddy reverse proxy that serves the built SPA and fronts the backend (Section 4.4). Rather than push artifacts to a managed platform, it deploys through a self-hosted GitHub Actions runner on the production host, so that a push to `main` runs the deploy job in place, where the Docker daemon and the production secrets are already available (Figure 20).

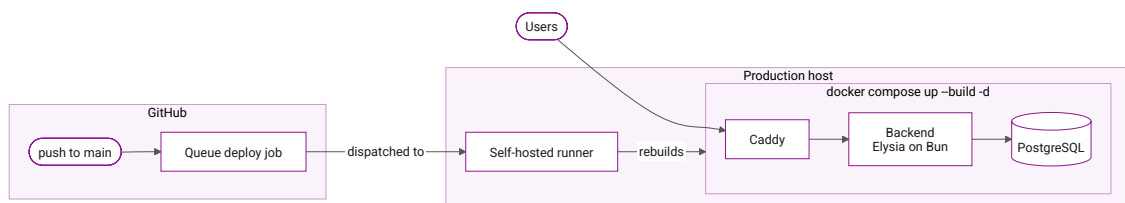


Figure 20: GitHub dispatches the deploy job to a runner on the production host

The job itself is modest. It copies the production environment file into place and runs `docker compose up --build -d`, which rebuilds only the images whose inputs changed and restarts the backend, PostgreSQL, and Caddy in place. Because the runner lives on the host, no SSH keys, image registries, or remote-deploy credentials have to be managed in CI. The trade-off is a single point of failure, which a thesis-scale deployment can reasonably accept.

The production Caddyfile ([Listing 4](#)) sends `/api/*` to the backend and serves the SPA for everything else, with `reverse_proxy` upgrading WebSocket handshakes automatically.

```
1 {$FRONTEND_URL:http://localhost} {  
2     handle /api/* {  
3         reverse_proxy backend:3000  
4     }  
5     handle {  
6         root * /srv  
7         try_files {path} /index.html  
8         file_server  
9     }  
10 }
```

Listing 4: Production Caddyfile routing both REST and WebSocket traffic

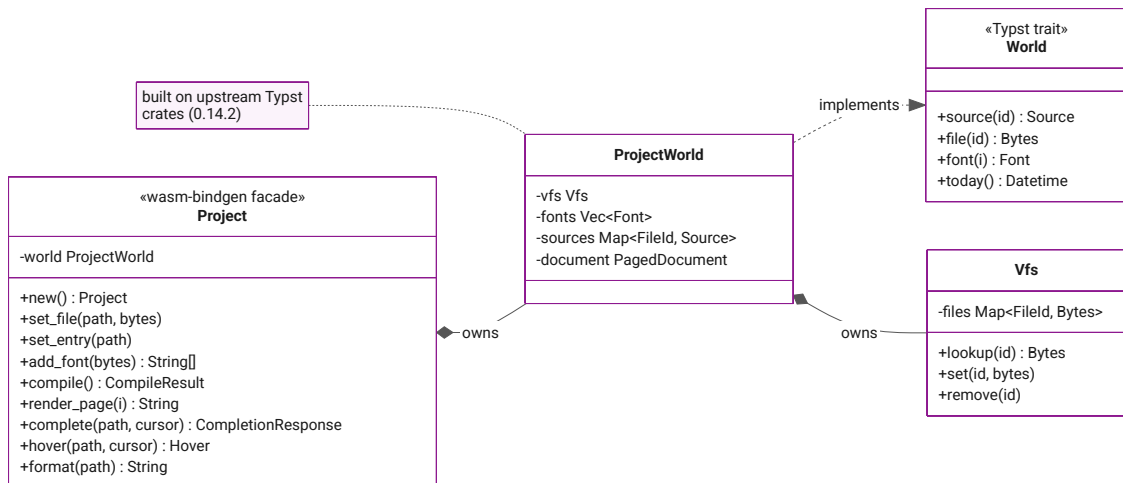
5.3 TYPST-WEB

5.3.1 THE ENGINE IN WEBASSEMBLY

The shared engine is `typsten`, a Rust crate that links the real Typst compiler and its companion crates into a single WebAssembly [\[24\]](#) module. The Design chapter motivated compiling the genuine engine rather than reimplementing it ([Section 4.2](#)). This section covers how that module is built and how it works inside. Three ideas run through it.

- A single crate links the many upstream Typst crates into one module.
- A thin, typed boundary exposes that module to JavaScript.
- Almost all of its logic stays plain Rust that runs under `cargo test` rather than in a browser.

[Figure 21](#) shows the layering.

Figure 21: Core types of `typsten`, built on the upstream Typst crates

5.3.1.1 CRATE LAYOUT AND THE HOST-TESTABLE CORE

`typsten` is a crate pinned to Typst version 0.14.2 across every `typst-*` dependency, so the parser, compiler, language intelligence, and exporters never disagree on a version. Its modules split by concern: `world` and `vfs` hold the file system and the Typst World, `compile` and `render` build the document and draw its pages, `ide`, `format`, and `highlight` answer editor queries, `diagnostics` carries typed errors, and `project` is the boundary.

That last module is special. By convention the only code carrying `#[wasm_bindgen]` [25] lives in `project.rs`, a facade that holds no logic and delegates each call in one line to the core (Listing 5). Everything below it is ordinary Rust, so the entire engine is exercised by `cargo test` on the host, where tests are fast and a debugger works. WebAssembly tests are reserved for checking the boundary itself.

```

1 #[wasm_bindgen]
2 pub struct Project {
3     world: ProjectWorld,
4 }
5
6 #[wasm_bindgen]
7 impl Project {
8     #[wasm_bindgen(constructor)]
9     pub fn new() -> Project { Project { world: ProjectWorld::new() } }
10
11     pub fn set_file(&mut self, path: &str, bytes: Vec<u8>) {
12         self.world.set_file(path, bytes);
13     }
14
15     pub fn compile(&self) -> CompileResult {
16         let result = compile_project(&self.world);
17         comemo::evict(COMEMO_MAX_AGE); // trim memoized entries
18         result
19     }
20     // Every other method is one more such one-line delegation.
21 }

```

Listing 5: The wasm-bindgen facade delegates to a plain-Rust core

Figure 22 maps that delegation out: each facade method forwards to the core module named for its job.

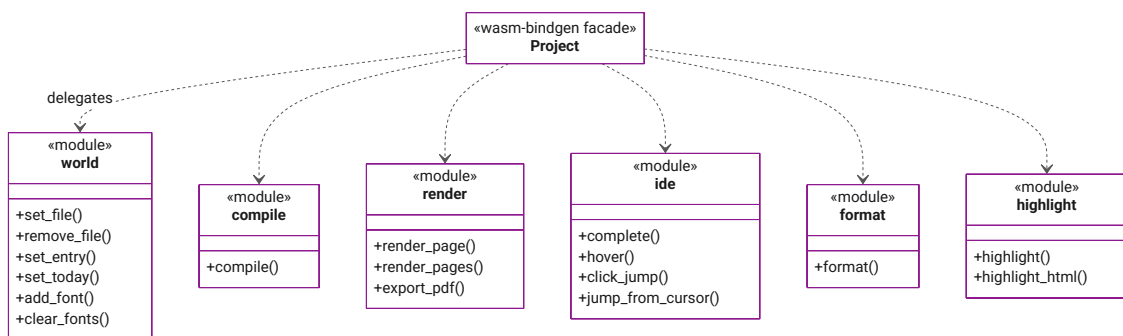


Figure 22: Where each Project method comes from

5.3.1.2 BUILDING THE MODULE

`wasm-pack` [26] compiles the crate for the `wasm32-unknown-unknown` target. The release profile optimizes for size rather than speed (Listing 6). Setting `opt-level = "z"` tells the compiler to minimize code size, link-time optimization and a single codegen unit let it trim and inline across the whole crate, and `panic = "abort"` drops the unwinding machinery that a browser never uses. A final `wasm-opt` pass then shrinks the binary further and enables the bulk-memory and sign-extension proposals the engine relies on. The output

is the `.wasm` binary, a small JavaScript glue file, and a TypeScript declaration file generated from the Rust types.

Linking the whole engine is not free. The binary is tens of megabytes uncompressed, which is the standing cost of full fidelity, paid once per session while the module streams in. The service build copies the `.wasm` next to its own bundle so the engine self-resolves at runtime through `new URL("./typsten_bg.wasm", import.meta.url)`.

```

1 [profile.release]
2 opt-level = "z"
3 lto = true
4 codegen-units = 1
5 panic = "abort"
6
7 [package.metadata.wasm-pack.profile.release]
8 wasm-opt = ["-O", "--enable-bulk-memory", "--enable-sign-ext",
9             "--enable-nontrapping-float-to-int"]

```

Listing 6: The size-oriented release profile for the WebAssembly build

5.3.1.3 THE JAVASCRIPT BOUNDARY

The facade is the only surface JavaScript sees, and only plain data crosses it: strings, numbers, byte arrays, and small structured records. Those records are defined once in Rust and derive `Serialize` [27] and `Tsify` [28], so their TypeScript types are generated rather than hand-written (Listing 7). A `Diagnostic`, for example, becomes an equivalent TypeScript interface with no separate declaration to maintain, so the two sides cannot drift apart as the engine changes.

<pre> 1 #[derive(Serialize, Tsify)] 2 #[tsify(into_wasm_abi)] 3 pub struct Diagnostic { 4 pub severity: Severity, 5 pub message: String, 6 pub hints: Vec<String>, 7 pub location: Option<Location>, 8 } </pre>	<pre> 1 interface Diagnostic { 2 severity: Severity; 3 message: string; 4 hints: string[]; 5 location?: Location; 6 } </pre>
---	--

Listing 7: The Rust struct and the TypeScript interface `tsify` generates from it

5.3.1.4 RUNNING OFF THE MAIN THREAD

A full compile would block the interface if it ran on the main thread, so the engine lives in a Web Worker. A thin wrapper hosts it (Listing 8). `init` loads the `.wasm` and constructs the `Project`, and every method delegates to it. The `wasm Project` never leaves the worker. Only the wrapper is published over Comlink [11], which keeps the boundary to plain data.

The wrapper also fills a gap left by the sandbox. WebAssembly has no clock, so `datetime.today()` would have nothing to return. The wrapper therefore stamps the current date into the world through `set_today` immediately before every compile, so a long-lived worker never serves a stale date.

```

1 class TypstenWorker {
2   #project: Project | undefined;
3
4   async init(wasmUrl: string): Promise<void> {
5     await init({ module_or_path: wasmUrl });
6     this.#project = new Project();
7   }
8
9   compile(): ReturnType<Project["compile"]> {
10    // WASM has no clock: refresh the date right before every compile.
11    const now = new Date();
12    this.#p().set_today(now.getFullYear(), now.getMonth() + 1,
13      now.getDate(), now.getHours(), now.getMinutes(), now.getSeconds());
14    return this.#p().compile();
15  }
16  // ... setFile, renderPages, complete, hover, format delegate likewise
17 }
18
19 Comlink.expose(new TypstenWorker());

```

Listing 8: The worker wrapper crosses Comlink, never the wasm Project

The main thread reaches the worker through a `Comlink.wrap` proxy, but that proxy belongs to the `TypstProject` API of the next section.

5.3.1.5 THE TYPST WORLD OVER A VIRTUAL FILE SYSTEM

Typst compiles against a `World` as the trait through which it reads source files, fonts, the standard library, and the current date. `ProjectWorld` implements it over an in-memory virtual file system, a map from file identifier to bytes ([Listing 9](#)). Nothing in the module touches the network or disk. The host pushes files in beforehand, and the engine only ever reads this in-memory copy. This is why the module can present itself as pure functions, where identical inputs always give identical output.

The same principle covers time. With no clock in the sandbox, `today` simply returns whatever date JavaScript last pushed in. File identifiers are parsed from plain paths, and a leading `@namespace/name:version/...` is recognized as a package path, so an imported package resolves through the same VFS as project files.

```

1 impl World for ProjectWorld {
2     // library(), book(), main(): trivial accessors over owned fields.
3
4     fn file(&self, id: FileId) -> FileResult<Bytes> {
5         self.vfs.lookup(id).cloned() // read straight from the VFS map
6     }
7
8     fn font(&self, index: usize) -> Option<Font> {
9         self.fonts.get(index).cloned()
10    }
11
12    fn today(&self, _offset: Option<i64>) -> Option<Datetime> {
13        self.today // no clock in WASM, JS pushes the date in
14    }
15
16    // source() reads like file() but caches the parsed Source.
17 }

```

Listing 9: The Typst World reads from an in-memory VFS

Fonts follow the same in-memory model. The default families are embedded into the binary from `typst-assets`, loaded directly so the WASM build avoids `typst-kit`'s native font-discovery dependencies, which have no place in a browser. A caller adds families the engine does not bundle, such as CJK or a brand font, through `add_font`, which returns the canonical family name Typst matches by. Typst's `FontBook` has no per-entry removal, so `clear_fonts` resets to the embedded set and the host re-adds whatever it wants to keep.

5.3.1.6 INCREMENTAL COMPILATION

A live preview recompiles on almost every keystroke, so a naive engine that reparsed and re-laid-out the whole document each time would not keep up. `typsten` stays responsive through memoization with `comemo` [29] and a cached parse tree. The `World::source` accessor returns a cached `Source` whose hash is content-based, so when the input has not changed `comemo` finds the previous evaluation and layout and reuses them. An edit refreshes that cached `Source` in place (Listing 10), reparsing only the region that changed. The rest of the parse tree is untouched, so its memoized results stay valid.

```

1 pub fn set_file(&mut self, path: &str, bytes: Vec<u8>) {
2     let id = file_id(path);
3
4     if self.sources.get_mut().contains_key(&id) {
5         if let Ok(text) = std::str::from_utf8(&bytes) {
6             self.sources.get_mut().get_mut(&id).unwrap().replace(text);
7         } else {
8             self.sources.get_mut().remove(&id); // non-UTF-8
9         }
10    }
11
12    self.vfs.set(id, Bytes::new(bytes));
13 }

```

Listing 10: An edit refreshes the cached source in place

A host test guards this path, which is otherwise easy to regress. The first compile records a single full parse, the edit records one incremental update with no second full parse, and a later recompile reflects the change without parsing from scratch. Memory is bounded by removing memoized entries older than ten compile generations after each run, the `comemo::evict` call in the facade.

5.3.1.7 OUTPUT AND DIAGNOSTICS

Compilation runs the engine once and treats its diagnostics as data rather than exceptions, so it never returns a `Result` and never throws (Listing 11). On success, the laid-out document is cached and any warnings are returned. On failure, the errors are returned and the previous document is kept, so a transient mistake never blanks the preview. Pages are rendered to SVG on demand by `render_pages`, so a viewer draws only the pages in view. The same cached document also backs `export_pdf`.

```

1 pub fn compile_project(world: &ProjectWorld) -> CompileResult {
2     let warned = compile::<PagedDocument>(world);
3     let mut diagnostics: Vec<Diagnostic> = warned.warnings.iter()
4         .map(|d| Diagnostic::from_source(world, d)).collect();
5     match warned.output {
6         Ok(document) => world.cache_document(document), // new preview
7         Err(errors) => diagnostics.extend( // keep last-good preview
8             errors.iter().map(|d| Diagnostic::from_source(world, d)),
9         )
10    }
11    CompileResult { pages: page_sizes(world), diagnostics }

```

Listing 11: Compilation returns diagnostics as data, keeps the last document

5.3.1.8 EDITOR QUERIES

The remaining methods answer the editor as the user types, and each one is a thin wrapper over the official `typst-ide` [30] or `typstyle` [31] crate (Listing 12). Completions and hover tooltips come from `typst-ide`, formatting from `typstyle` applied to the cached source, and syntax highlighting from Typst’s own highlighter, which returns tagged byte ranges (`typ-key`, `typ-func`, and the like) that the editor applies as decorations. Reusing the real parser means the highlighting can never drift from the language the compiler accepts.

```

1 pub fn hover(world: &ProjectWorld, path: &str, cursor: usize)
2   -> Option<Hover> {
3     let source = world.source(file_id(path)).ok()?;
4     let tip = tooltip(world, None, &source, cursor, Side::Before)?;
5     Some(Hover::from(tip))
6 }
7
8 pub fn format(world: &ProjectWorld, path: &str) -> Option<String> {
9     let source = world.source(file_id(path)).ok()?;
10    Typstyle::new(Config::new()).format_source(source).render().ok()
11 }

```

Listing 12: Editor queries defer to `typst-ide` or `typstyle`

5.3.1.9 JUMPING BETWEEN EDITOR AND PREVIEW

One pair of queries does what the rendered SVG cannot express on its own. An exported page carries its text without source spans and its internal links without destinations. `click_jump` recovers both from the laid-out frame, resolving a click in the preview to a source location, an internal link target, or an external URL. `jump_from_cursor` runs the same mapping in reverse, scrolling the preview to the position that matches the cursor (Figure 23).

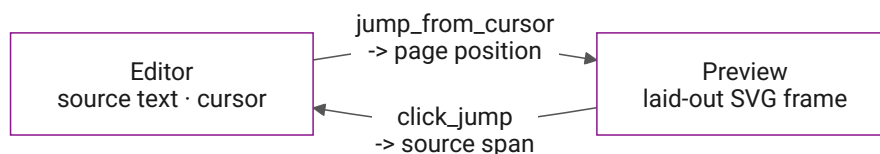


Figure 23: The jump pair bridges editor and preview

Together the two make navigation seamless, both between the editor and the preview and within the preview itself. For an external URL, `click_jump` only reports the target and leaves opening it to the JavaScript side.

5.3.2 THE TYPSTPROJECT SERVICE

The engine ships as @vedivad/typst-web-service, an editor-agnostic npm package whose TypstProject class is the only thing that talks to the worker (Section 4.2). Everything an application needs goes through one object, which owns a mirror of the virtual file system and a dedicated worker, turns a stream of edits into coalesced compiles, and installs Typst packages in the browser. Figure 24 shows how the class is put together.

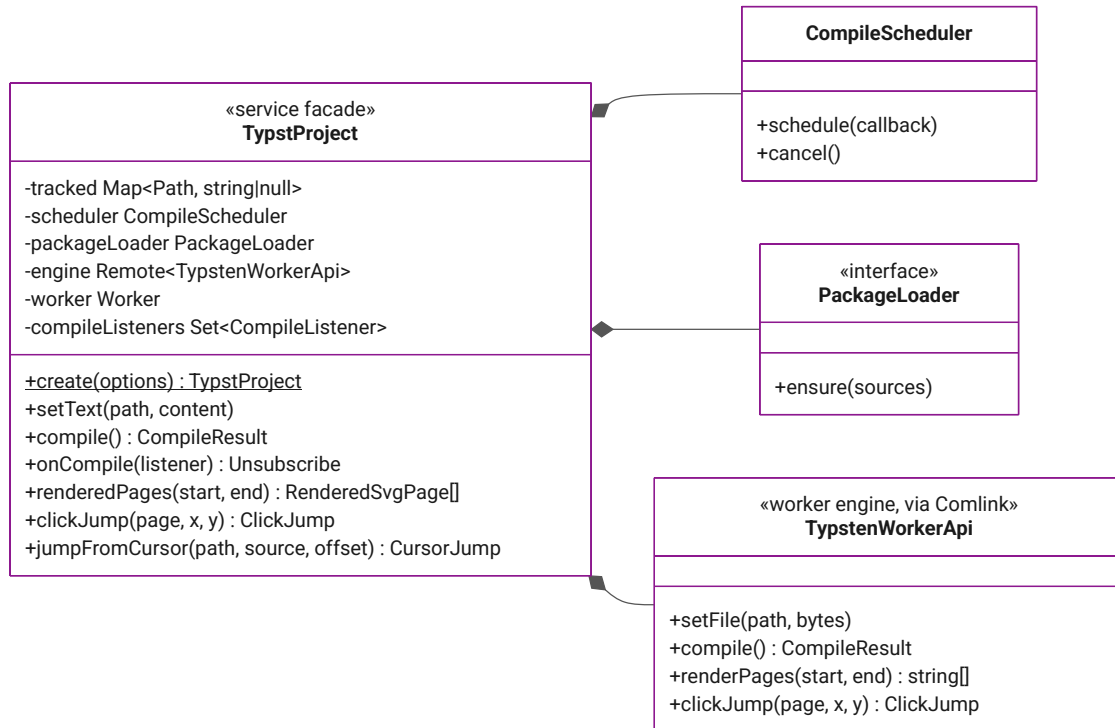


Figure 24: High-level class diagram of TypstProject

5.3.2.1 ONE PROJECT, ONE WORKER

A project is created through a static create factory, not the constructor (Listing 13).

```

1 export class TypstProject {
2   // ...
3   static async create(
4     options: TypstProjectCreateOptions
5   ): Promise<TypstProject> {
6     const worker = new Worker(
7       new URL("./typsten-worker.js", import.meta.url),
8       {
9         type: "module",
10      }
11    );
12
13    const engine = Comlink.wrap<TypstenWorkerApi>(worker);
14    const wasmUrl = new URL("./typsten_bg.wasm", import.meta.url).href;
15    await engine.init(wasmUrl);
16
17    const project = new TypstProject(engine, worker, options);
18    await engine.setEntry(project._entry);
19
20    return project;
21  }
22 }

```

Listing 13: TypstProject.create spawns the worker and awaits the engine

The factory spawns the worker, wraps it with `Comlink.wrap` into a typed proxy, waits for the wasm to initialize, and only then hands back a ready `TypstProject`. That proxy is the main-thread counterpart to the worker's `Comlink.expose` from the previous section, so a call on `engine` forwards its arguments to the worker and returns a promise. One project owns exactly one worker and one wasm instance, so separate projects stay isolated rather than sharing an engine.

5.3.2.2 MIRRORING EDITS INTO THE VFS

`TypstProject` owns the project files on the main thread and mirrors edits into the worker's VFS, the in-memory copy `typsten` compiles against. It records each file in a tracked map from path to content, namely a string for a text file and `null` for a binary one. That record lets a write short-circuit, since `writeText` returns early when the new content matches what is tracked, so re-sending an unchanged buffer never reaches the worker (Listing 14). Text is UTF-8 encoded before it crosses, and paths are normalized to a leading slash. The public `setText` wraps that private `write` and schedules a compile only when the content actually changed, so a redundant edit costs nothing. A batch `setMany` writes many files and schedules a single compile rather than one per file.

```

1 class TypstProject {
2   // ...
3   private writeText(p: Path, content: string): Promise<unknown> | null {
4     if (this.tracked.get(p) === content) return null;
5     this.tracked.set(p, content);
6     return this.engine.setFile(p, encoder.encode(content));
7   }
8
9   async setText(path: Path, content: string): Promise<void> {
10    const write = this.writeText(normalizePath(path), content);
11    if (write) {
12      await write;
13      this.scheduleCompile();
14    }
15  }
16 }

```

Listing 14: Writes dedup against the tracked mirror before touching the worker

5.3.2.3 SCHEDULING COMPILES

Recompiling on every keystroke would be wasteful, so each mutation calls `scheduleCompile` rather than compiling directly. The scheduler combines a debounce with a throttle. The debounce holds the compile until edits pause: every new edit pushes its deadline back by `debounceMs`, so a fast burst collapses into one compile once typing goes idle. That alone could defer the preview indefinitely during continuous typing, so a throttle adds a ceiling, a second deadline `maxWaitMs` after the burst began that is never pushed back. Whichever deadline arrives first triggers the compile and cancels the other, so the preview refreshes when typing settles, or at least every `maxWaitMs` while it continues.

5.3.2.4 COMPILING AND RESULTS

A compile guards against itself, because the worker is asynchronous and a fast typist can start a newer compile before an older one returns. `compile` stamps each run with an incrementing version and, when the worker answers, keeps the result only if its version is still the latest ([Listing 15](#)). A stale answer is dropped rather than allowed to overwrite fresher output or flash outdated diagnostics. Errors take the same path as success, where a failed compile becomes a `CompileResult` carrying diagnostics, not a thrown exception, so the caller always receives a result it can render.

```

1 class TypstProject {
2   // ...
3   async compile(): Promise<CompileResult> {
4     this.scheduler.cancel();
5     const version = ++this.compileVersion;
6     let result: CompileResult;
7     try {
8       await this.packageLoader.ensure(
9         [...this.tracked.values()].filter((v) => v !== null),
10      );
11      result = await this.engine.compile();
12    } catch (err) {
13      result = errorAsCompileResult(err); // errors never throw
14    }
15    if (version === this.compileVersion) { // still the latest?
16      this._lastResult = result;
17      for (const fn of this.compileListeners) fn(result);
18    }
19    return result;
20  }
21 }

```

Listing 15: A version counter drops stale results from slower, superseded compiles

Results reach the application through `onCompile`, which registers a listener and returns a function to unregister it. A late subscriber is not left blank, since `onCompile` replays the last result immediately when a compile has already happened. The result types themselves (`CompileResult`, `Diagnostic`, `PageInfo`, and the rest) are re-exported straight from the `typsten` crate, so the service adds no parallel set of types that could drift from the engine's.

5.3.2.5 INSTALLING PACKAGES IN THE BROWSER

Typst documents pull in community packages with `@preview/name:version`, and the engine reads them from its VFS but never fetches them itself. The service fills that gap on the main thread (Listing 16). Before each compile it scans the tracked sources for `@preview` imports, downloads each package as a compressed tar from the Typst registry, decompresses it with `flate` [32], unpacks it with `nanotar` [33], and writes every member into the VFS. Because a package may import further packages, the loader scans each fetched `.typ` file in turn and follows those imports breadth-first until no new package appears. Each spec is fetched once and memoized, and a failed download is evicted so the next compile can retry it. The result is a package manager that runs entirely in the browser against nothing more than a static file host.

```

1 async function fetchPackage(spec: string): Promise<string[]> {
2   const [name, version] = spec.split(":");
3   const qualified = `@preview/${spec}`;
4   const res = await fetch(`${REGISTRY}/${name}-${version}.tar.gz`);
5   if (!res.ok) throw new Error(`failed to fetch ${qualified}`);
6
7   const tar = gunzipSync(new Uint8Array(await res.arrayBuffer()));
8   const nested = new Set<string>();
9   for (const entry of parseTar(tar)) {
10    if (!entry.data || entry.name.endsWith("/")) {
11      continue;
12    }
13
14    await setFile(`${qualified}/${entry.name}`, entry.data);
15    // a package may import further @preview packages of its own
16    if (entry.name.endsWith(".typ")) {
17      for (const s of specsIn(decoder.decode(entry.data))) {
18        nested.add(s);
19      }
20    }
21  }
22  return [...nested]; // specs to follow transitively
23 }

```

Listing 16: Fetching a package: download, decompress, unpack, and feed the VFS

5.3.2.6 RENDERING AND EXPORTING

Pages are rendered on demand rather than all at once. `renderPage` returns one page as SVG, and `renderedPages` returns a window, zipping the worker's SVG strings with the `PageInfo` dimensions from the last compile into `RenderedSvgPage` records ([Listing 17](#)).

```

1 class TypstProject {
2   // ...
3   async renderedPages(start: number, end: number) {
4     const pages = this._lastResult?.pages ?? [];
5     const svgs = await this.engine.renderPages(start, end);
6     return svgs.map((svg, i) => {
7       const dims = pages[start + i]; // page size from the last compile
8       return {
9         index: start + i,
10        width: dims?.width ?? 0,
11        height: dims?.height ?? 0,
12        svg,
13      };
14    });
15  }
16 }

```

Listing 17: Rendered pages pair each SVG with its dimensions

5.3.2.7 THE FULL EDIT-TO-PREVIEW CYCLE

Taken together, these stages form one asynchronous round trip from an edit to a refreshed preview. [Figure 25](#) traces it end to end, where the edit mirrors into the VFS, the debounce schedules a compile, the compile installs any `@preview` packages and runs the engine, listeners receive the result, and the consumer pulls the rendered pages. The worker boundary turns every step into a promise, which is why an API that reads synchronously runs asynchronously underneath.

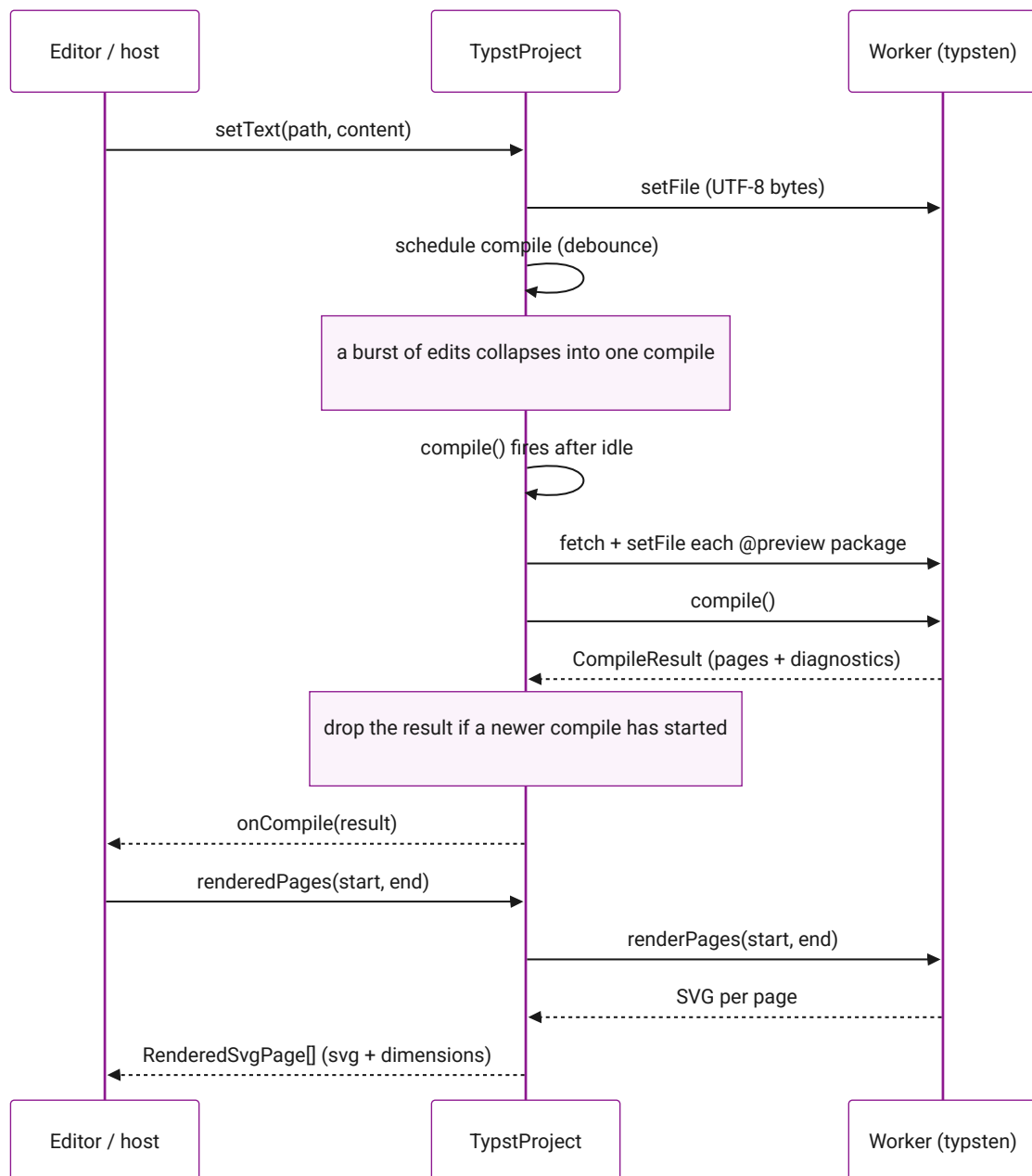


Figure 25: One edit-to-preview cycle across the editor, TypstProject, and the worker

5.3.2.8 BRIDGING EDITOR AND PREVIEW COORDINATES

The engine works in UTF-8 byte offsets while CodeMirror and JavaScript strings count UTF-16 code units, so the two diverge as soon as a document contains a non-ASCII character. Every editor query converts at the boundary with the helpers in `coords.ts` (Listing 18).

```

1 export function cmOffsetToByte(text: string, offset: number): number {
2   return encoder.encode(text.slice(0, offset)).length;
3 }
4
5 export function byteToCmOffset(text: string, byte: number): number {
6   if (byte <= 0) return 0;
7   const bytes = encoder.encode(text);
8   if (byte >= bytes.length) return text.length;
9   return decoder.decode(bytes.subarray(0, byte)).length;
10 }

```

Listing 18: Offset conversion between engine bytes and CodeMirror code units

The other mismatch is screen geometry: clicks arrive in CSS pixels, but the engine resolves positions in Typst points. The `PreviewNavigator` reconciles them without owning the rendered pages: it reads the live page elements on every event, finds the one under the pointer, converts the hit into that page's own point coordinates from its SVG `viewBox`, and passes it to `clickJump` (Listing 19). The reverse direction, `scrollToSource`, runs `jumpFromCursor` and scrolls the matching page into view. This is the main-thread half of the source-and-preview jump from the previous section (Figure 23).

```

1 pointToPage(clientX: number, clientY: number) {
2   const pages = this.opts.pages();
3   for (let page = 0; page < pages.length; page++) {
4     const svg = rootSvg(pages[page]);
5     if (!svg) continue;
6     const rect = svg.getBoundingClientRect();
7     if (clientX < rect.left || clientX > rect.right ||
8         clientY < rect.top || clientY > rect.bottom) {
9       continue;
10    }
11    const { x, y } = clientToPagePoint(
12      rect, svg.viewBox.baseVal.width, clientX, clientY,
13    );
14    return { page, x, y };
15  }
16  return null;
17 }

```

Listing 19: The navigator hit-tests a click against the live page elements

5.3.3 THE CODEMIRROR EXTENSION

@vedivad/codemirror-typst is the editor layer, a CodeMirror 6 [34] extension that brings a TypstProject's language features to the editor as syntax highlighting, diagnostics, completion, hover, and formatting (Section 4.2). Every feature is one small plugin built the same way, where an editor event triggers a single TypstProject query and the answer becomes editor state. The package carries no language logic of its own, so each feature stays in step with the real compiler. Table 6 lists the features and the one query each makes.

FEATURE	EDITOR TRIGGER	PROJECT QUERY	EDITOR RESULT
Compile-sync	edit	setText	schedules a compile
Diagnostics	compile result	onCompile	lint marks
Highlighting	viewport, edit	highlight	decorations
Completion	typing	completion	completion popup
Hover	pointer	hover	tooltip
Formatting	keymap	format	minimal doc edit

Table 6: Editor events to one TypstProject query and back to editor state

5.3.3.1 COMPOSING THE SETUP

A consumer builds one TypstProject, then calls createTypstSetup to get the bundle of extensions to hand to CodeMirror (Listing 20). The sync option decides who owns the text. Under editor-driven the bundle includes the compile-sync plugin, so CodeMirror is the source of truth. Under external it omits that plugin, leaving another source (the Yjs adapter of the next section) to push edits into the project.

```

1 const project = await TypstProject.create();
2 await project.setText("/main.typ", "= Hello, Typst!");
3
4 const setup = createTypstSetup({ project, sync: "editor-driven" });
5
6 new EditorView({
7   state: EditorState.create({
8     doc: project.getText("/main.typ") ?? "",
9     extensions: [basicSetup, ...setup, typstFilePath.of("/main.typ")],
10   }),
11 });

```

Listing 20: Wiring an editor with one project, the setup bundle, and file-path facet

5.3.3.2 MIRRORING EDITS AND SURFACING DIAGNOSTICS

Two plugins form the compile loop inside the editor. The compile-sync plugin is write-only. On mount and on every document or path change it mirrors the buffer into the project with `setText`, and never compiles or subscribes (Listing 21). The project dedups the write and schedules the compile, so editors sharing a project share one schedule.

```

1 update(update: ViewUpdate): void {
2   const nextPath = update.state.facet(typstFilePath);
3   if (update.docChanged || nextPath !== this.currentPath) {
4     this.currentPath = nextPath;
5     this.push(update.view);
6   }
7 }
8
9 private push(view: EditorView): void {
10  // mirror the buffer into the VFS; the project schedules its own compile
11  this.options.project
12    .setText(this.currentPath, view.state.doc.toString())
13    .catch((err) => console.error("[typst]", err));
14 }

```

Listing 21: Compile-sync only mirrors edits into the project, never compiles

The diagnostics plugin is the read side. It subscribes to `project.onCompile` once and, for each result, keeps the diagnostics for this editor's file, converts them, and dispatches them through `@codemirror/lint` (Listing 22). The conversion to `CMDiagnostic` maps each engine byte range to CodeMirror's UTF-16 offsets (the `byteToCmOffset` helper again), widens an empty range so it stays visible, and folds the compiler's hints into the message. Note that `onCompile` replays the last result synchronously on subscribe, so the dispatch defers with `queueMicrotask` to avoid firing into a view that a tab switch is still building.

```

1 private applyDiagnostics(result: CompileResult): void {
2   queueMicrotask(() => {
3     if (this.destroyed) return;
4     const view = this.view;
5     const path = view.state.facet(typstFilePath);
6     const diagnostics = result.diagnostics
7       .filter((d) => d.location && normalizePath(d.location.file) === path)
8       .map((d) => toCMDiagnostic(view.state, d));
9     view.dispatch(setDiagnostics(view.state, diagnostics));
10  });
11 }

```

Listing 22: Diagnostics are filtered to the active file before dispatch

5.3.3.3 SYNTAX HIGHLIGHTING WITHOUT A GRAMMAR

Highlighting ships no Lezer [35] grammar or TextMate [36] definition. Instead the plugin asks `project.highlight(source, from, to)` for the visible range and receives spans tagged with Typst's own `typ-*` classes, the same ones the official `typst.app` web editor uses. Reusing the compiler's parser means the highlighting can never drift from the language it accepts.

The plugin follows the same shape as diagnostics, where a view plugin issues requests and a state field holds the decorations. On a document or viewport change it debounces and requests only `view.viewport`, never the whole document (Listing 23). Each request carries an id and a snapshot of the document, and a result is dropped if a newer request has started, the plugin was destroyed, or the document changed in the meantime, since its offsets would no longer line up. While a request is in flight the state field maps the existing decorations through edits, so highlighting tracks the text until fresh spans arrive.

```

1 class HighlightPlugin implements PluginValue {
2   // ...
3   private async refresh(): Promise<void> {
4     const id = ++this.requestId;
5     const requestDoc = this.view.state.doc;
6     const { from, to } = this.view.viewport; // only the visible range
7
8     let spans: HlSpan[];
9     try {
10      spans = await this.project.highlight(
11        requestDoc.toString(), from, to);
12    } catch {
13      return; // worker still starting, or project torn down
14    }
15
16    if (this.destroyed || id !== this.requestId
17        || this.view.state.doc !== requestDoc) return;
18    this.view.dispatch({
19      effects: setHighlights.of(decorationsFor(spans, requestDoc.length)),
20    });
21  }
22 }

```

Listing 23: Highlighting requests only the viewport and drops stale results

`decorationsFor` turns the spans into CodeMirror mark decorations, one per span, skipping empty ranges and sorting them so that nested marks, such as emphasis within a heading, compose through the CSS cascade (Listing 24). The `typ-*` classes are colored by a token theme, and a small bridge maps them onto `@lezer/highlight` tags so an existing CodeMirror theme can drive the palette.

```

1 export function decorationsFor(
2   spans: HlSpan[],
3   docLength: number,
4 ): DecorationSet {
5   const ranges = [];
6   for (const span of spans) {
7     const from = Math.max(0, Math.min(span.from, docLength));
8     const to = Math.max(0, Math.min(span.to, docLength));
9     if (to > from) // mark decorations must be non-empty
10       ranges.push(Decoration.mark({ class: span.tag }).range(from, to));
11   }
12   return Decoration.set(ranges, true);
13 }

```

Listing 24: Each span becomes a mark decoration tagged with its typ-* class

5.3.3.4 COMPLETION AND HOVER

Completion is a single completion source registered with CodeMirror’s autocompletion. It forwards the cursor to `project.completion` and maps the response into CodeMirror’s shape (Listing 25), where each `CompletionKind` becomes a CodeMirror completion type, the engine’s byte offset becomes a UTF-16 from, and a `typst-ide` snippet with `${...}` placeholders is translated into a CodeMirror snippet template.

```

1 function toCM(
2   source: string,
3   result: CompletionResponse,
4 ): CompletionResult | null {
5   if (!result.completions.length) return null;
6   const from = byteToCmOffset(source, result.from);
7   const options = result.completions.map((c) => ({
8     label: c.label,
9     type: KIND_TO_TYPE[c.kind],
10    detail: c.detail ?? undefined,
11    apply: c.apply?.includes("${")
12      ? snippet(typstSnippetToCMTemplate(c.apply))
13      : (c.apply ?? c.label),
14  }));
15   return { from, options, validFor: /^[#\w.-]*$/ };
16 }

```

Listing 25: Engine completions mapped to CodeMirror options, kinds, and snippets

Hover is a `hoverTooltip` that calls `project.hover` and builds the tooltip from the result (Listing 26). A hover is either a code signature or a sentence of prose. For code, the snippet is run back through `project.highlightHtml`, so the tooltip uses the same token palette as the editor. If a lint diagnostic already covers the position, the hover yields to the diagnostic’s own tooltip.

```

1 return hoverTooltip(async (view, pos) => {
2   const path = view.state.facet(typstFilePath);
3   const result = await project.hover(path, view.state.doc.toString(), pos);
4   if (!result || !result.value.trim()) return null;
5
6   const html = result.kind === "code"
7     ? `<pre>${await project.highlightHtml(result.value)}</pre>`
8     : undefined;
9
10  return {
11    pos,
12    create() {
13      const dom = document.createElement("div");
14      if (html) dom.innerHTML = html;
15      else dom.textContent = result.value;
16      return { dom };
17    },
18  };
19 });

```

Listing 26: Hover renders the engine tooltip, reusing editor highlighting

5.3.3.5 FORMATTING

Formatting is bound to a keymap (Shift-Alt-f, with an optional format-on-save). It runs `project.format` over the whole document, since the engine has no range formatter, and applies the result carefully (Listing 27). Rather than replacing the whole buffer, it computes the minimal change from the common prefix and suffix, so the cursor and the undo history survive. If the format returns nothing (the source did not parse) or the buffer changed while the asynchronous call was in flight, it aborts without modifying the document.

```

1 async function runFormat(view: EditorView, project: TypstProject) {
2   const path = view.state.facet(typstFilePath);
3   const doc = view.state.doc.toString();
4   const formatted = await project.format(path, doc);
5
6   if (formatted === undefined) return; // not formattable (parse error)
7   if (view.state.doc.toString() !== doc) return; // buffer moved meanwhile
8
9   const changes = diffChanges(doc, formatted);
10  if (changes.length > 0) {
11    view.dispatch({ changes });
12  }
13 }

```

Listing 27: Formatting applies a minimal diff, preserving cursor and undo

5.3.4 THE YJS ADAPTER

The collaboration layer is `@vedivad/typst-web-yjs`, a small package kept separate so collaboration stays opt-in, depending only on `yjs` [10] and the service (Section 4.2). It exposes two one-way adapters, `syncYTextToTypstProject` for a single shared `Y.Text` and `syncYMapToTypstProject` for a `Y.Map` keyed by file path. Each observes the shared document and mirrors its latest state into a `TypstProject`, which then compiles (Figure 26). This is the `sync: "external"` counterpart from the previous section, so the editor leaves the project alone and peer edits arriving through Yjs drive it instead.

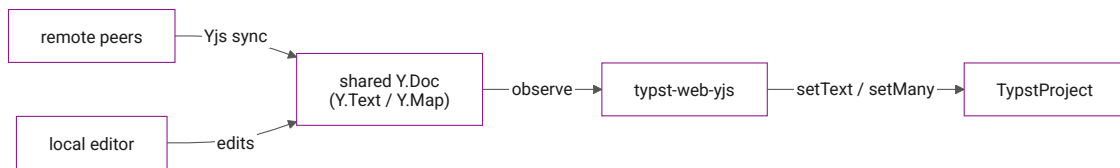


Figure 26: One-way sync from the shared document into a `TypstProject`

5.3.4.1 SERIALIZING WRITES TO THE PROJECT

A `TypstProject` write is asynchronous, yet Yjs emits a change event for every keystroke, local or remote. Firing `setText` per event would let writes overlap and even land an older snapshot last. A small `SerializedSync` prevents that by keeping at most one write in flight and always writing the latest state (Listing 28). Each event marks it dirty and starts a drain loop if none is running, and the loop keeps writing the current document while new events arrive, so a burst collapses into as few writes as the project can absorb.

```

1 requestSync(): Promise<void> {
2   if (this.disposed) return this.running ?? Promise.resolve();
3   this.dirty = true;
4   if (!this.running) this.running = this.drain();
5   return this.running;
6 }
7
8 private async drain(): Promise<void> {
9   try {
10    while (this.dirty && !this.disposed) {
11      this.dirty = false;
12      await this.writeLatest();
13    }
14  } finally {
15    this.running = undefined;
16    if (this.dirty && !this.disposed) this.running = this.drain();
17  }
18 }
  
```

Listing 28: One write in flight at a time, always writing the latest state

5.3.4.2 MIRRORING ONE FILE, AND MANY

For a single file, `syncYTextToTypstProject` is almost all wiring, since it observes the `Y.Text`, requests a sync on every change, and writes `setText` as the latest state (Listing 29).

```

1 export function syncYTextToTypstProject(
2   options: SyncYTextToTypstProjectOptions,
3 ): TypstYjsSync {
4   const { project, ytext, path } = options;
5   const sync = new SerializedSync(() =>
6     project.setText(path, ytext.toString()),
7   );
8
9   const observer = () => void sync.requestSync();
10  ytext.observe(observer);
11
12  return {
13    ready: sync.ready,
14    flush: () => sync.flush(),
15    dispose: () => {
16      ytext.unobserve(observer);
17      sync.dispose();
18    },
19  };
20 }

```

Listing 29: The single-file adapter observes the text and writes the latest on change

For a whole project, `syncYMapToTypstProject` watches a `Y.Map` keyed by path. It observes the map for structural changes (a file added, removed, or replaced) and attaches an observer to each `Y.Text` for in-place edits, collecting the affected paths. The write step then classifies each dirty path and mirrors it, where a lone change goes through `setText` or `setBinary`, several batch into one `setMany`, and deleted keys become `remove` calls (Listing 30).

```

1 const text: Record<string, string> = {};
2 const binary: Record<string, Uint8Array> = {};
3 const removed: string[] = [];
4
5 for (const path of [...dirtyPaths]) {
6   const value = files.get(path);
7   if (value === undefined) removed.push(path);
8   else if (value instanceof Y.Text) text[path] = value.toString();
9   else binary[path] = value;
10 }
11 await project.setMany({ ...text, ...binary }); // setText/setBinary if one
12 for (const path of removed) await project.remove(path);

```

Listing 30: The multi-file write step diffs the dirty paths into project calls

5.4 TOUR OF TYPST

The Tour of Typst is the first product built on `typst-web`. It is a static single-page application with no backend (Section 4.3): the lessons ship inside the bundle and every compile runs client-side through the library of the previous sections. This section covers what the product adds on top of that library, its content pipeline, the lesson workspace, and the handling of solutions and progress, rather than the engine mechanics already documented under Section 4.2.

5.4.0.1 KEY DEPENDENCIES

The tour is a fully static single-page application bundled by Vite [37] and deployed to GitHub Pages without any server-side component. Most of the stack (Table 7) is shared with the Collaborative Editor. The pieces specific to a tutorial are the content tooling, `marked` [38] and `shiki` [39], which render the lesson prose and highlight its examples.

PACKAGE	ROLE
<code>solid-js</code>	UI framework with fine-grained, signal-based reactivity
<code>vite + vite-plugin-wasm</code>	Build tool and dev server with first-class WASM support
<code>@codemirror/*</code>	Extensible code editor engine (v6)
<code>@vedivad/codemirror-typst</code>	Typst language support bundling the compiler, language intelligence, and formatter as a single WASM module
<code>solid-ui</code>	<code>shadcn-style</code> component library for SolidJS, built on Kobalte headless primitives
<code>marked + shiki</code>	Markdown parsing and syntax highlighting for lesson content
<code>tailwindcss</code>	Utility-first CSS framework (v4 with Vite plugin)
<code>bun</code>	Package manager and script runner

Table 7: Key dependencies of the Tour of Typst

5.4.0.2 BUILD

Vite drives the whole build. `vite-plugin-wasm` emits the `typst-web` WebAssembly as a static asset that loads lazily on first use, and a prerender step writes one static HTML page per chapter, which keeps the initial page load fast and the site friendly to static hosting on GitHub Pages. The tutorial content is bundled into the same build, as the next section describes.

5.4.1 CONTENT AS DATA

The content is data, not code. Each lesson is a directory under `content/<locale>/<part>/<chapter>/` holding its prose (`index.md`), the editor starter (`template.typ`), an optional reference solution (`solution.typ`), and any YAML data files it uses ([Listing 1](#)). [Table 8](#) resolves the `hello-world` lesson, whose files live under `content/en/00-basics/01-hello-world/`, to show how each maps onto the page. Because a lesson is just files, authoring a new one means adding a directory and its content, with no component to write or build step to adjust. The same layout also keeps prose, starter, and solution side by side, so a lesson can be read and revised as a unit.

FILE	BECOMES
<code>index.md</code>	Markdown rendered to the prose pane
<code>template.typ</code>	Editor starter, written into the VFS as <code>/main.typ</code> (what the learner edits)
<code>solution.typ</code>	Optional reference solution, swapped into the editor on request

Table 8: The files of the `en:hello-world` lesson, one per page part

A lesson that needs data adds YAML files to the same directory (for instance the bibliography lesson ships a `refs.yaml`). Each is mounted into the VFS as `/<name>` alongside `/main.typ` but never shown, so the lesson stays a single editable file.

All of a lesson's files share one key, here `en:hello-world`, formed from the locale (`en`) and the chapter directory with its ordering prefix stripped (`01-hello-world`). The intermediate part directory (`00-basics`) is discarded, since it organizes the files on disk but plays no part in the key. That same key appears as the `hello-world` chapter under the `Basics` part of `en's tour.ts`, which is what places the lesson in the navigation. The key is the only coupling between the content on disk and the per-locale content tree.

5.4.1.1 BUNDLING THE LESSONS

Nothing is fetched at runtime. At build time Vite's `import.meta.glob` pulls every lesson file into the bundle, one glob per file type, each `eager` so the content lands in the main chunk and `?raw` so it arrives as text rather than a parsed module ([Listing 31](#)).

```

1 const markdown = import.meta.glob("../..content/**/*.md", {
2   eager: true,
3   query: "?raw",
4   import: "default",
5 });

```

Listing 31: Every lesson file is bundled at build time as a raw string

Each glob hands back a flat record keyed by file path. A small helper parses each path into its locale and chapter and re-keys it as `locale:chapter`, so a lesson's prose, template, solution, and data are each one map lookup away (Listing 32). The chapter directory's numeric prefix (`01-hello-world`) orders the lessons but is stripped from the key.

```

1 const composeKey = (locale: string, key: string) => `${locale}:${key}`;
2
3 function buildFileIndex(files: Record<string, string>, filename: string) {
4   const index = new Map<string, string>();
5   for (const [path, contents] of Object.entries(files)) {
6     const entry = parseContentPath(path); // { locale, key, filename }
7     if (!entry || entry.filename !== filename) continue;
8     index.set(composeKey(entry.locale, entry.key), contents);
9   }
10
11   return index;
12 }

```

Listing 32: Paths are re-keyed by locale and chapter for one-lookup access

5.4.1.2 PER-LOCALE CONTENT TREES

Each locale owns a `tour.ts` module that declares its own table of contents, an ordered list of parts, each an ordered list of chapters with a key and a title (Listing 33). The set of locales is discovered from these modules rather than configured centrally, and a language switcher, its choice remembered across visits, picks which tree drives the navigation. Because each locale is an independent tree, a translation can reorder or drop chapters instead of being a strict overlay on one canonical structure.

```

1 export const meta = { locale: "en", label: "English" };
2
3 export const parts: readonly Part[] = [
4   {
5     title: "Basics",
6     chapters: [
7       { key: "hello-world", title: "Hello, World!" },
8       { key: "text-formatting", title: "Text Formatting" },
9       // ...
10    ],
11  },
12
13  // ... more parts
14 ];

```

Listing 33: A locale's `tour.ts` declares its own ordered parts and chapters

5.4.1.3 ROUTING

Routing is client-side, since the tour ships as static files (Section 4.3.1). The active locale's parts are concatenated into one ordered chapter list that drives both the previous/next controls and the URL. A lesson's address is just its chapter key (Listing 33), as `/<chapter>`: the part is not encoded, so reordering chapters in `tour.ts` never breaks an existing link. The bare root redirects to the first chapter, and an unrecognized path renders a not-found screen rather than a silent fallback. The locale stays out of the URL as a persisted setting that selects which tree drives navigation, so a shared link resolves the same chapter regardless of the recipient's language.

5.4.1.4 RENDERING LESSON PROSE

The Markdown prose is rendered to HTML with `marked`, extended with `marked-alert` for note and tip callouts. Fenced code blocks are highlighted by `shiki` using its Typst and YAML grammars, so a lesson's examples are colored at build time and appear styled before the editor loads. A custom code renderer wraps each highlighted block in a `data-code` attribute that preserves the original source, which lets the page attach a copy button to every example (Listing 34).

```

1  const renderer = new Renderer();
2
3  renderer.code = ({ text, lang }) => {
4    const html = highlighter.codeToHtml(text, {
5      lang: highlighter.getLoadedLanguages().includes(lang) ? lang : "text",
6      themes: shikiThemes, // github-light / github-dark-dimmed
7      defaultColor: false,
8    });
9
10   return `

Listing 34: A custom marked renderer highlights blocks and stores their source



## 5.4.2 THE LESSON WORKSPACE



A lesson is presented in the three-panel “read, try, see” layout of the Design chapter (Section 4.3), with the prose on the left and a live editor and an SVG preview on the right. The editor and preview are the typst-web stack of the previous sections, so this section covers only how the product wires them in, not how they compile or highlight.



56


```

5.4.2.1 ONE SHARED PROJECT

The whole workspace runs on a single `TypstProject`. `EditorShell` creates it once on mount and hands it to the editor and preview through SolidJS context ([Listing 35](#)).

The Tour picks the compile schedule here, a 50 ms debounce with a 300 ms ceiling, and otherwise leaves the engine to the library. Initialization is deferred and non-blocking. The shell renders its children only once the project resolves, but the surrounding lesson prose is outside the shell, so a learner can read while the `WebAssembly` module streams in.

```

1 export default function EditorShell(props: { children: JSX.Element }) {
2   const [resources, setResources] = createSignal<TypstResources>();
3
4   onMount(() => {
5     void (async () => {
6       const project = await TypstProject.create({
7         autoCompile: { debounceMs: 50, maxWaitMs: 300 },
8       });
9       setResources({ project });
10    })();
11  });
12
13  return (
14    <Show when={resources()}>
15      {(res) => (
16        <TypstResourcesContext.Provider value={res()}>
17          {props.children}
18        </TypstResourcesContext.Provider>
19      )}
20    </Show>
21  );
22 }

```

Listing 35: `EditorShell` creates one project and shares it through context

5.4.2.2 SEEDING A LESSON

On mount and on every lesson switch, the editor resets the shared project and loads the lesson into it ([Listing 36](#)). It clears the virtual file system, writes the lesson's data files plus the starter `template.typ` as `main.typ` with a single `setMany`, and compiles. A subscription then renders each fresh result to SVG pages for the preview. The learner only ever edits the `main.typ`, while the data files are mounted alongside but never shown, so a lesson stays a single focused file rather than a project to manage.

```

1 await project.clear();
2 await project.setMany({ ...props.auxFiles, [MAIN_PATH]: props.doc });
3 await project.compile();
4
5 unsubscribe = project.onCompile((result) => {
6   if (result.pages.length === 0) return;
7   void (async () => {
8     const pages = await project.renderedPages(0, result.pages.length);
9     props.onCompile?.(pages); // hand the SVG pages to the preview
10  })();
11 });

```

Listing 36: Each lesson reseeds the project, then streams pages to the preview

5.4.2.3 TOUR-SPECIFIC TOUCHES

One small Tour touch sits on top of this wiring, where a `MutationObserver` injects a copy button into each compile-diagnostic tooltip (those tooltips are created by the lint extension), so a learner can copy an error message without selecting it by hand.

5.4.3 SOLUTIONS AND PROGRESS

Edits persist under a single `localStorage` key holding a map from `locale:chapter` to the current buffer, rewritten whole on each change. Loading a lesson restores its saved buffer, or the starter template when none exists, and a reset stores the template back as that buffer.

Asking for the solution swaps the editor to `solution.typ` and dims the lines it shares with the starter, so the material the lesson introduces stands out. A longest-common-subsequence diff marks the lines the solution adds, and a state field dims the rest (Listing 37). The diff rebuilds on edits, so the highlight keeps up if the learner edits the revealed solution.

```

1 export function dimUnchangedLines(changed: Set<number>): Extension {
2   return StateField.define<DecorationSet>({
3     create: (state) => buildDecorations(state.doc, changed),
4     update: (deco, tr) =>
5       tr.docChanged
6         ? buildDecorations(tr.state.doc, changed)
7         : deco,
8     provide: (f) => EditorView.decorations.from(f),
9   });
10 }

```

Listing 37: A line diff drives a decoration that dims the unchanged lines

5.5 COLLABORATIVE EDITOR

5.5.1 MONOREPO LAYOUT

The codebase is a Bun-managed workspace of three packages.

```
typstogether/
├── frontend/
├── backend/
└── shared/
```

Listing 38: The typstogether monorepo packages

- `frontend/` is the SolidJS single-page application, bundled by Vite (detailed below).
- `backend/` is the Elysia server on Bun (detailed below).
- `shared/` is a small package of Yjs key constants (`FILES_KEY`, `ASSETS_KEY`, `FONTS_KEY`, `META_KEY`, and `ENTRY_KEY`) that the wire protocol of the collaboration channel depends on both sides agreeing about.

The frontend’s dependency on the backend is type-only, resolved directly against the source file without a build step. The implications for the cross-package contract with Elysia and Eden are covered in [Section 5.5.7](#).

5.5.2 BACKEND

The backend stack consists of the following dependencies listed in [Table 9](#).

PACKAGE	ROLE
<code>elysia</code>	Type-safe HTTP/WebSocket framework with Eden Treaty client
<code>bun</code>	Runtime, bundler, test runner, and package manager
<code>drizzle-orm</code>	Type-safe SQL ORM bound to PostgreSQL
<code>@electric-sql/pglite</code>	In-memory PostgreSQL used in development and tests
<code>better-auth</code>	Authentication library with pg adapter and OAuth providers
<code>@hocuspocus/server</code>	Yjs persistence server bridged onto Elysia’s WebSocket route

Table 9: Backend dependencies of the Collaborative Editor

A single Elysia instance exposes all routes under `/api`. The collaboration channel at `/api/collab` is a WebSocket route that bridges into Hocuspocus, which persists each project’s encoded `Y.Doc` to a `collab_document` row. Because both protocols share the `/api/*` namespace, a reverse proxy only needs one rule to forward both.

5.5.2.1 MODULE LAYOUT

Each route group under `backend/src/modules/<name>/` follows the same four-file convention.

```
backend/src/modules/<name>/  
├ index.ts  
├ model.ts  
├ service.ts  
└ macro.ts
```

Listing 39: The four-file module convention

- `index.ts` declares the Elysia route group, where it applies any macros with `.use()`, registers TypeBox models via `.model()`, and chains one HTTP method call per route. The path prefix is set in the `Elysia({ ... })` constructor.
- `model.ts` holds the TypeBox schemas for request bodies, URL parameters, and responses, combining `drizzle-typebox`'s `createSelectSchema` with hand-written `t.Object` and `t.Union` for derived shapes.
- `service.ts` exports a singleton class instance with the module's business logic. Errors are typed `HttpError` subclasses that a central `onError` handler maps to HTTP status codes.
- `macro.ts`, where applicable, defines Elysia macros that other modules attach for cross-cutting concerns like authentication and authorization.

5.5.2.2 AUTH AND ACCESS MACROS

Authentication and project authorization are factored into two layers of Elysia macros, so route handlers stay free of access-control logic.

The base macro is `auth: true` (from `modules/auth/macro.ts`). It resolves the Better Auth session from the request headers, throws `Unauthorized` when there is no signed-in user, and otherwise injects `user` and `session` into the handler context.

On top of that, the project-access macros enforce per-project roles. `projectMember`, `projectEditor`, and `projectOwner` are built from a single `requireRole` factory (Listing 41), differing only in their allowed-role lists. All three resolve `:id`, load the caller's membership, validate the role, and inject `{ project, role }` into the handler context, so a protected route never carries its own check.

```

1 const requireRole = (allowed: readonly ProjectRole[]) => ({
2   auth: true as const,
3   params: byIdProjectModel,
4   resolve: async ({ params, user }) => {
5     const { project, role } = await projectService.getMembership(
6       user.id, params.id);
7     if (!allowed.includes(role)) {
8       throw new ForbiddenError(`${allowed.join(" or ")} role required`);
9     }
10    return { project, role };
11  },
12 });

```

Listing 40: Factory for project-access macros, parameterized by allowed roles

```

1 export const projectAccessMacro = new Elysia({ name: "project-access-
  macro" })
2   .use(authMacro)
3   .macro("projectMember", requireRole(["owner", "editor", "viewer"]))
4   .macro("projectEditor", requireRole(["owner", "editor"]))
5   .macro("projectOwner", requireRole(["owner"]));

```

Listing 41: Project-access macros

Composition flows top-down, since the `.use(authMacro)` in Listing 41 means any route opting into a project-access flag also runs the base auth resolution first, so a missing session produces `Unauthorized` before any membership lookup.

This is what lets the routes stay concise and focused, with auth and access checks living in a single place rather than being duplicated across multiple handlers.

5.5.2.3 RATE LIMITING

A third macro reuses the same pattern for rate limiting. The `cooldown` macro builds on `authMacro` and, like the access macros, attaches per route through a declarative flag, so a handler never inspects timestamps itself. It is applied to project creation, where an accidental double-submit would otherwise create two projects (Listing 42).

```

1 export const cooldownMacro = new Elysia({ name: "cooldown-macro" })
2   .use(authMacro)
3   .macro("cooldown", (options: CooldownOptions) => ({
4     auth: true,
5     beforeHandle: ({ user }) =>
6       // key is namespaced by the caller's id, so the limit is per user
7       enforceCooldown(`${options.key}:${user.id}`,
8         options.ms, options.message),
9   }));

```

Listing 42: A per-user cooldown macro, namespaced by the caller's id

The cooldown is enforced in `beforeHandle`, ahead of the guarded work, so two near-simultaneous requests collapse to one. A still-active window throws `TooManyRequestsError`, which the central handler maps to 429. State is a single in-process `Map` of expiry timestamps, swept lazily so it stays bounded by recently active keys. This suffices for one backend instance. Horizontal scaling would move the state to a shared store such as Redis [40].

5.5.2.4 THE COLLABORATION SERVER

The `/api/collab` WebSocket does not implement the protocol itself. An Elysia `ws` route forwards each raw frame into a single `Hocuspocus` instance through `handleConnection`, keeping a `Map` from socket to connection. `Hocuspocus` handles the Yjs sync protocol, and three hooks supply everything specific to this application (Figure 27).

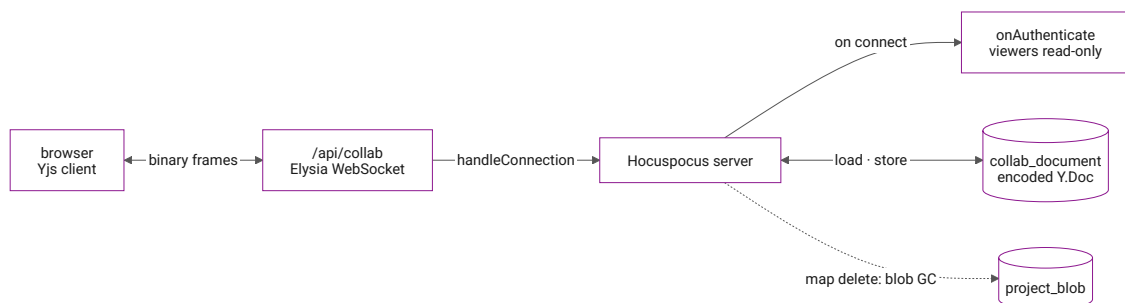


Figure 27: Collaboration server that persists the `Y.Doc` and removes orphaned blobs

The first hook is authentication, and it is also where read-only access is enforced. Recall that a viewer must not be able to mutate the shared document even over an open socket (Section 4.4). `onAuthenticate` resolves the Better Auth session from the connection headers and, for a viewer, marks the connection read-only, so Hocuspocus rejects that client's updates at the source rather than trusting the interface to disable editing (Listing 43). The project id arrives as the Hocuspocus document name, giving one `Y.Doc` per project.

```

1 export async function onAuthenticate(data: onAuthenticatePayload) {
2   const session = await auth.api.getSession({ headers:
    data.requestHeaders });
3   if (!session) throw new UnauthorizedError();
4
5   const { readOnly } = await authorizeCollab(
6     session.user.id, data.documentName);
7
8   if (readOnly) {
9     data.connectionConfig.readOnly = true;
10  }
11 }

```

Listing 43: The socket authenticates the session and marks viewers read-only

The second hook is persistence, a Hocuspocus database extension that loads a project's encoded `Y.Doc` from the `collab_document` row on first open and writes it back on change with an upsert. The whole document is stored as one binary snapshot, which is the durable copy the relational schema keeps for recovery ([Section 4.4](#)).

The third hook removes orphaned binary assets. Because the document is authoritative over which blobs are in use, the server observes the `assets` and `fonts` maps once the document loads, and when an entry is deleted or replaced it removes the blob row the old value referenced ([Listing 44](#)). Garbage collection is therefore event-driven off the shared document rather than a scheduled sweep, and the observer is detached when the document unloads.

```

1 afterLoadDocument({ documentName, document }) {
2   const observer = (event: Y.YMapEvent<string>) => {
3     for (const change of event.changes.keys.values()) {
4       if (change.action !== "delete" && change.action !== "update") {
5         continue;
6       }
7
8       void blobService.deleteBlob(documentName, change.oldValue as string);
9     }
10  };
11
12  for (const key of [ASSETS_KEY, FONTS_KEY]) {
13    document.getMap(key).observe(observer);
14  }
15 }

```

Listing 44: Deleting a reference in the shared document removes its blob

5.5.3 INVITATIONS

A project is shared by link rather than by naming accounts up front. The owner creates a link scoped to a role. Read-only for a viewer or edit for an editor. Anyone who opens it while signed in, joins with that role. Because the link is a bearer secret, the service stores only a hash of it and checks expiry and revocation when the link is redeemed ([Listing 45](#)).

```

1 function generateToken(): string {
2   const bytes = crypto.getRandomValues(new Uint8Array(32)); // 256 bits
3   return Buffer.from(bytes).toString("base64url");
4 }
5
6 const hashToken = (t: string) =>
7   new Bun.CryptoHasher("sha256").update(t).digest("hex");
8
9 // redeem: the presented token is hashed and matched against the stored
10 // hash, so the raw secret never reaches the database
11 const [invite] = await db.select().from(projectInvite)
12   .where(eq(projectInvite.tokenHash, hashToken(token)));

```

Listing 45: An invite token is persisted only as a SHA-256 hash

The token is 256 bits of cryptographically secure randomness. Because lookup matches on the hash rather than the raw value, a leaked database cannot yield a usable link, and the comparison runs over fixed-length hashes rather than the secret itself. The link expires after one week, fixed when it is created, and the owner can revoke it at any time by stamping `revokedAt`. Redemption rejects an expired or revoked invite, and is idempotent for an existing member, who keeps their current role rather than being downgraded.

5.5.4 BINARY ASSET UPLOADS

Asset bytes are uploaded through the blobs module and stored relationally, while the document keeps only references to them ([Section 4.4.3.1](#)). Two constraints guard the upload endpoint, namely a maximum file size and an allow-list of MIME types. Elysia enforces the size cap declaratively on the multipart field, 10 MB for general assets and a higher 30 MB for fonts, which accommodates large CJK (Chinese, Japanese, Korean) font files, whose tens of thousands of glyphs make them far larger than Latin typefaces.

Each upload path carries its own allow-list, where assets are checked against `ALLOWED_MIME_TYPES` and fonts against `FONT_MIMES`. The asset content type is validated in the service rather than declaratively, because Elysia's byte-sniffing rejects text-based formats such as SVG ([Listing 46](#)).

```

1 export const ALLOWED_MIME_TYPES = [
2   "image/png", "image/jpeg", "image/gif",
3   "image/svg+xml", "image/webp", "application/pdf",
4 ] as const;
5
6 export const FONT_MIMES = new Set([
7   "font/ttf", "font/otf", "font/collection",
8 ]);
9
10 export const MAX_BLOB_SIZE = "10m"; // 10 MB
11 export const MAX_FONT_SIZE = "30m"; // 30 MB
12
13 export const blobUploadModel = t.Object({
14   file: t.File({ maxSize: MAX_BLOB_SIZE }),
15 });
16
17 export const fontUploadModel = t.Object({
18   file: t.File({ maxSize: MAX_FONT_SIZE }),
19 });

```

Listing 46: Assets cap at 10 MB, fonts at 30 MB, each with a MIME allow-list

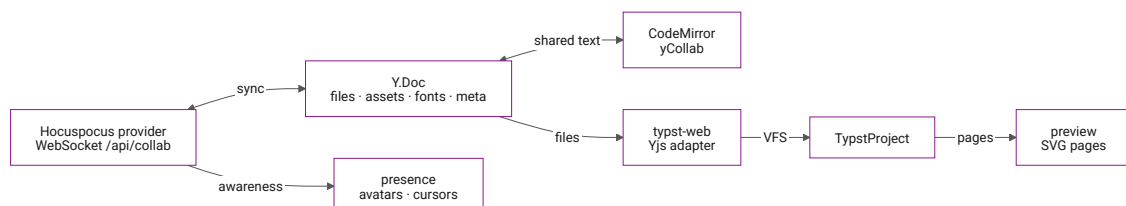
5.5.5 FRONTEND

The frontend shares its base stack with the Tour of Typst ([Table 7](#)), namely SolidJS with the `solid-ui` and Tailwind layer, CodeMirror with `@vedivad/codemirror-typst`, and Vite. Collaboration adds the CRDT layer on top, `yjs` for the shared document and `@hocuspocus/provider` for the WebSocket sync client.

5.5.5.1 COLLABORATION WIRING

A shared Yjs document is the source of truth, and the frontend wires it to both the editor and the local compiler ([Figure 28](#)). One `HocuspocusProvider` opens the WebSocket to `/api/collab` and binds a per-project `Y.Doc` whose four top-level maps hold the project, namely `files` (a `Y.Map` of `Y.Text`), `assets`, `fonts`, and `meta` ([Section 4.4.3](#)).

The maps are held back until the provider reports the document synced, so no consumer ever sees a transient empty project, and the compile entry stored in `meta` is mirrored into a signal so the rest of the app reacts without observing the map itself.

Figure 28: Frontend wiring where the shared `Y.Doc` drives editor and compiler

The editor binds to the shared text directly. Each file's `Y.Text` is attached to `CodeMirror` through `yCollab`, so a local keystroke becomes a `Y.Text` mutation and a peer's mutation arrives as an editor transaction, with a per-file `Y.UndoManager` keeping undo history scoped to its file.

Compilation stays client-side, exactly as in the Tour. A `TypstProject` runs in its worker, and `syncYMapToTypstProject` from the `typst-web Yjs` adapter mirrors the `files` map into its virtual file system, so a peer's edit drives the same local compile a local edit would (Listing 47). The compile entry is read with `untrack`, so changing it re-routes the existing worker rather than rebuilding it.

```
1 const project = await TypstProject.create({
2   entry: untrack(entry),
3   autoCompile: { debounceMs: 200, maxWaitMs: 1000 },
4 });
5
6 const sync = syncYMapToTypstProject({ project, files });
7 await sync.ready;
8 await project.compile();
```

Listing 47: The shared `files` map mirrors into a local `TypstProject`

5.5.5.2 VIRTUALIZED PREVIEW

A compile returns the dimensions of every page, but rasterizing them all on each edit would be wasteful for a long document, so the preview renders lazily. The pane lays out one placeholder box per page, each set to the page's aspect ratio so the scroll height is right before anything is drawn. An `IntersectionObserver` watches those boxes against the scroll container with a generous overscan (1200 px above and below), and only a page that enters that window is rasterized through `project.renderPage` and cached. A page's rendered SVG is dropped again as soon as it leaves the window, and the whole cache is invalidated on the next compile. Scrolling a long project therefore only ever renders the handful of pages on screen.

5.5.5.3 PRESENCE

Presence travels over `Yjs`'s awareness channel rather than the document. On connecting, each client publishes a `user` field carrying its identity (`id`, `name`, `avatar`, and a color derived deterministically from the user `id`), and the `yCollab` binding adds the live cursor as a relative position alongside it (Section 4.4.8). A small hook surfaces the other clients' states, the toolbar renders them as avatars, and clicking one resolves that peer's relative cursor position back to an absolute offset and scrolls the matching file into view.

5.5.5.4 MULTI-FILE EDITING

The `files` map is flat, keyed by full path, yet the sidebar shows a folder tree. Folders are never stored but derived from the path prefixes, so a folder exists precisely when some file lives under it (Listing 48). The derived folders and the files are then grouped by parent and emitted as a depth-first list, folders before files, that skips the subtree of any collapsed folder. The same derivation drives rename and move as prefix rewrites within one Yjs transaction. Because deletion is a shared operation, the editor guards against a collaborator removing the open file, where an observer on the `files` and `assets` maps falls back to the entry or the first remaining file and notes when the removal came from a peer.

```
1 const folders = new Set<string>();
2 for (const path of paths) {
3   for (let dir = dirOf(path); dir; dir = dirOf(dir)) {
4     folders.add(dir);
5   }
6 }
```

Listing 48: The file tree is derived from path prefixes, with no stored folders

5.5.5.5 AUTHORIZING TOOLBAR

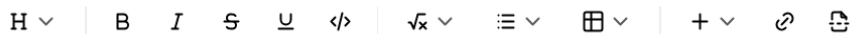


Figure 29: Toolbar with quick actions

Above the editor, there is a toolbar which contains a list of quick actions such as toggling a text bold, italic or underline, inserting a link or entering math or code mode. A keymap mirrors the toolbar, so the same actions are also reachable through the usual shortcuts. Both feed into one set of small action functions that take the editor view and dispatch a single edit, and each is written so that every cursor or selection is handled on its own when more than one is active.

Most actions toggle rather than blindly insert. Triggering bold on text that is already bold removes the markers again, whether the selection covers the bold text itself or the cursor simply sits somewhere inside it. Detecting that surrounding pair depends on the marker, where symmetric ones such as `*` and `$` are matched in order of appearance, so `*a* b *c*` is not mistaken for one long span, while bracketed ones such as `#strike[...]` look for the nearest opener before and closer after the cursor. The line-based actions for headings and lists behave the same way across every line a selection touches, and switch a line from one kind to another in place rather than stacking prefixes. Code and math go one

step further and cycle through their variants, from plain text to inline to a block (display math, for math) and back.

A drop handler rounds the toolbar out. A file dragged in from the sidebar is inserted by its extension, as an `#image`, an `#include`, or a plain path string.

5.5.5.6 FIND AND REPLACE

Next to the file tree, the same sidebar holds a search panel that finds and replaces across every file at once (Section 4.4.7). Since the whole project lives in the shared `files` map, the search runs entirely in the browser. It walks the map, splits each file into lines, and scans every line with `indexOf`, respecting the match-case and whole-word toggles (Listing 49). Matching is plain substring search and matches are grouped per file and shown as highlighted line snippets.

```

1 for (const [path, yText] of files.entries()) {
2   for (const [line, lineText] of yText.toJSON().split("\n").entries()) {
3     const haystack = caseSensitive ? lineText : lineText.toLowerCase();
4
5     for (
6       let col = haystack.indexOf(needle);
7       col !== -1;
8       col = haystack.indexOf(needle, col + needle.length)
9     ) {
10      const end = col + needle.length;
11      if (needWholeWord && !(isBoundary(haystack, col - 1) &&
12        isBoundary(haystack, end))) {
13        continue;
14      }
15      matches.push({ path, line, matchStart: col, matchEnd: end });
16    }
17  }
18 }

```

Listing 49: Search scans each shared `Y.Text` line by line for `indexOf` hits

Clicking a result opens the file and selects the match in the editor. The panel can also replace text, either one match at a time or every match across the project at once. A single replacement edits the matched `Y.Text` in place, while Replace All sweeps each file from back to front so that rewriting a later match leaves the earlier offsets valid. Because every replacement edits the `Y.Text` directly, the change syncs to the other collaborators like a normal edit.

5.5.6 ERROR HANDLING AND LOGGING

On the backend, failures are modelled as data rather than assembled individually at each call site. An abstract `HttpError` carries a status code, and a small hierarchy of subclasses names each failure mode, such as `UnauthorizedError`, `ForbiddenError`, and `NotFoundError`. Services throw these, and a single Elysia `onError` hook maps any `HttpError` to its status and message while letting framework validation and unexpected errors fall through to Elysia's own handling ([Listing 50](#)).

```

1 export abstract class HttpError extends Error {
2   abstract readonly status: number;
3 }
4 export class NotFoundError extends HttpError {
5   readonly status = 404;
6 }
7
8 // one hook turns every typed error into its HTTP response
9 .onError(({ error, status }) => {
10   if (error instanceof HttpError) {
11     return status(error.status, error.message);
12   }
13   return; // fall through to Elysia's default handling
14 })

```

Listing 50: Typed errors map to status codes through a single handler

This keeps route handlers free of status-code bookkeeping, since a service that cannot find a project simply throws `NotFoundError`, and the client receives a 404 without the handler ever mentioning it. Because the subclasses are ordinary types, the same error can be asserted on directly in the service tests.

The same `onError` hook is also where failures are logged, through `pino` [\[41\]](#) as structured JSON. Each module logs via a child logger (`log.child({ module })`) so every line is tagged with its origin, and Better Auth's own output is routed through the same instance to keep one format across the backend. Crucially, the hook grades severity rather than logging everything alike, where a mapped `HttpError` is an ordinary response and is not logged at all, client-input faults (validation, unknown route, malformed body) are logged at `warn` as expected noise, and only a genuinely unhandled error reaches `error` level, where it would otherwise vanish silently.

On the client, a failed request surfaces as a toast with a human-readable message rather than a silent failure, and the collaboration layer treats a dropped connection as a recoverable state that reconciles on reconnect ([Section 4.4.3](#)) rather than an error to report.

5.5.7 END-TO-END TYPE SAFETY

The frontend and backend share a type-safe API contract without a code-generation step. The frontend imports the backend's `App` type as a type-only dependency, and Eden Treaty derives an `api` object from it whose shape mirrors every route. Each route declares its response in `TypeBox` through Elysia's validation layer ([Listing 51](#)). For `/projects/:id` the response is `projectMembershipModel`, a `TypeBox` object built from the Drizzle `project` table plus a literal-union `role`. Eden Treaty infers the frontend's `data` from that exact schema ([Listing 52](#)), so any backend change to the model lights up the frontend's type check on the next save.

```
1 .get("/:id", ({ project, role }) => ({ project, role }), {  
2   projectMember: true,  
3   response: projectMembershipModel,  
4 })
```

Listing 51: Backend route declaring its response shape via a `TypeBox` model

```
1 const { data, error } = await api.projects({ id }).get();  
2 // inferred shape of `data` (when `error` is null):  
3 // { project: { id: string; name: string; ..};  
4 //   role: "owner" | "editor" | "viewer" }
```

Listing 52: Frontend call through Eden Treaty with the fully inferred result

6 EVALUATION

6.1 POST-BUILD USABILITY SURVEY

6.1.1 OVERVIEW

The second survey accompanied the first hands-on test of the interactive tour. In contrast to the first survey, which probed habits and expectations, this round asked a small group of six testers to actually work through the tour and report back. It functioned as a lightweight usability test, albeit without a fixed task scenario or a formal hypothesis. The feedback covers six dimensions: explanation clarity, pacing, the confidence gained, navigation between the instructions and the editor, the helpfulness of error messages, and the likelihood of future adoption. Despite the modest sample size, the responses were consistently positive.

6.1.2 QUANTITATIVE RESULTS

Across all six closed questions the tour scored in the favorable range. Clarity was rated on a scale where three marks the ideal balance between “too hard” and “too easy”: 85.7% of testers landed exactly on that sweet spot, and the remaining 14.3% leaned only slightly towards “too hard”. That small lean is plausibly down to the testers’ mixed backgrounds. Not all of them program, so the more code-oriented passages can feel harder than they do to participants with a development background. Nobody found the tour too easy, and on pacing the verdict was unanimous: 100% described the chapter-to-chapter progression as balanced. For a tour that carries a reader from a blank editor to a working document, an undivided result on pacing is a strong signal that the sequencing works.

The confidence question is the most telling, since it measures what testers actually took away. Every respondent reported some progress and, crucially, nobody selected “I already knew Typst, my skills haven’t improved”, confirming that the tour reached an audience that genuinely benefited from it. A combined 83.3% came out either confident (33.3%) or equipped with the basics and ready to practice further (50%), and only one respondent (16.7%) still felt lost. Given that the first survey showed most participants had at best seen examples of Typst, moving the bulk of them this far in a single sitting is a meaningful outcome for an introductory format.

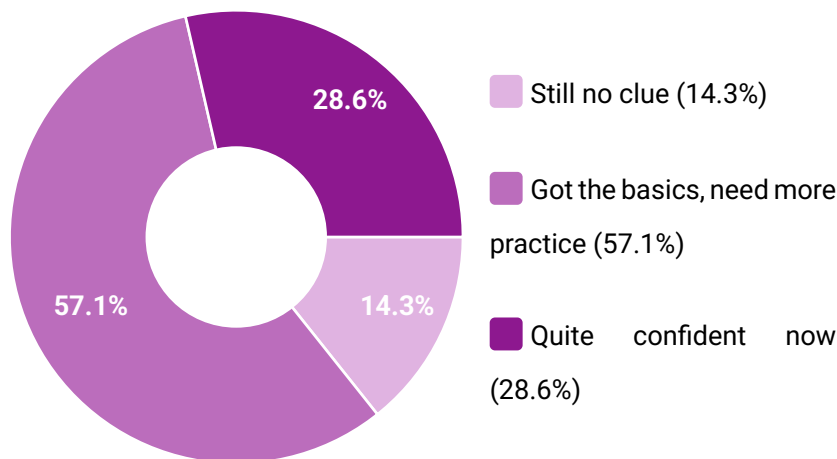


Figure 30: Self-reported confidence in writing Typst after completing the tour.

The two questions on the interactive shell itself were equally positive. Navigation between the instructions and the embedded editor scored 83.3% at the top rating of five and 16.7% at four, with nothing lower, confirming that the seamless switching between reading and editing, a core design goal, works as intended. Error feedback fared even better: 100% agreed that the messages shown on failing code helped them understand what went wrong, and since good diagnostics often decide whether a learner keeps going or gives up, a unanimous “yes” is reassuring.

Finally, testers rated how likely they were to reach for Typst on their next academic project, from one (“sticking with LaTeX, Word, etc.”) to five (“definitely”). Half (50%) leaned towards adoption at four, a third (33.3%) stayed neutral, and 16.7% remained cautious at two. Nobody ruled Typst out and nobody committed to an unconditional five, a realistic spread for first-time users. The clear tilt towards the “likely” end after a single tour is an encouraging adoption signal.

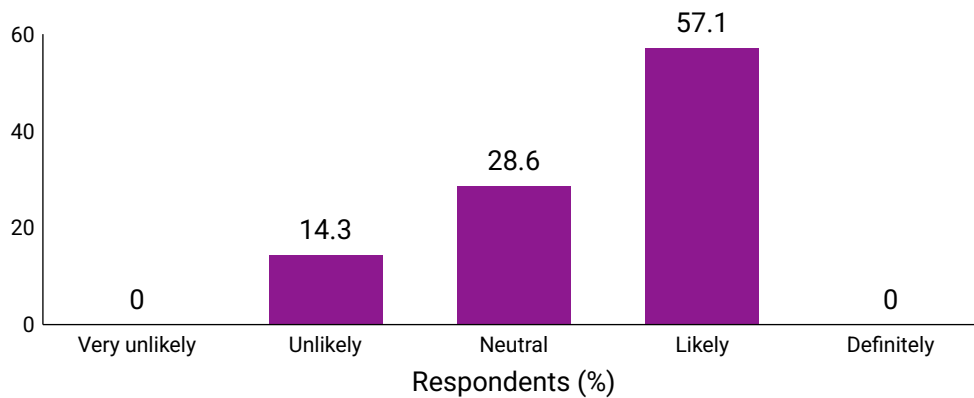


Figure 31: Likelihood of choosing Typst for the next academic project after the tour.

6.1.3 QUALITATIVE FEEDBACK

The free-text responses were constructive and praised the experience as pleasant and fun. Three themes emerged, each with a corresponding design decision:

- **Editor controls:** One tester was unsure what the format button does and wanted guidance on inserting line breaks so that commands and text do not run on indefinitely.
- **Orientation in long documents:** Because each chapter builds on the last, the worked example keeps growing and it can become unclear which part needs editing. Darkening everything outside the current chapter was considered but rejected as unmaintainable across all translations. Instead, the relevant diff is highlighted in the solution, a good trade-off between readability and maintainability.
- **Scope wishes:** A German translation and a dedicated presentation chapter were requested. The German translation has since been added. It was deliberately held back until the content had stabilized, so it only had to be written and reviewed once rather than kept in sync through ongoing changes. A presentation chapter is of limited value on its own: the tour already teaches the Typst foundation, and presentations are mostly layout configuration handled by existing templates whose syntax differs per package. A short chapter on starting from a template may nonetheless be worthwhile, since packages are already mentioned.

6.1.4 CONCLUSION

The results, although based on a smaller group of testers than initially hoped, were satisfying and showed that the application fully meets its requirements: clarity, pacing, navigation, and error feedback all scored in the favorable range, and the tour measurably raised participants' confidence in writing Typst. The reported issues were addressed where reasonably achievable within the project's scope.

6.2 REQUIREMENTS VERIFICATION

Every requirement from [Section 3.1](#) and [Section 3.2](#) is checked against one of four verification methods: *measurement* for quantitative thresholds, *test* for behaviour exercised by the automated suite, *demonstration* for properties observed by running the deployed application, and *inspection* for guarantees established by reviewing the code or configuration. Each ID links back to its definition, and the evidence column points to the figure or section that realizes it.

6.2.1 TOUR OF TYPST

6.2.1.1 FUNCTIONAL REQUIREMENTS

ID	METHOD	EVIDENCE
T-FR-1.1	Demonstration	Each lesson's prose is authored as Markdown and rendered in the reading pane (Section 4.3.2).
T-FR-1.2	Demonstration	The split-pane workspace with draggable panes (Figure 10).
T-FR-2.1	Demonstration	Lessons are ordered into parts and chapters (Section 4.3.2).
T-FR-2.2	Demonstration	Each lesson has a stable, linkable URL and navigation moves between them (Section 4.3.3).
T-FR-3.1	Demonstration	The lesson view embeds the <code>typst-web</code> editing core (Section 4.2) as its editor and preview panes (Section 4.3.4).
T-FR-4.1	Demonstration	The "Show Solution" action reveals the solution with the differing lines highlighted by a line diff (Section 4.3.5 , Listing 37).
T-FR-4.2	Demonstration	The "Format" action runs <code>typstyle</code> through the WASM module (Section 5.3.3.5).

ID	METHOD	EVIDENCE
T-FR-4.3	Demonstration	A lesson, or every lesson, resets to its starting template (Section 4.3.5).
T-FR-5.1	Test	Per-lesson edits and the selected language persist to LocalStorage (Section 4.3.5).
T-FR-5.2	Test	Saved edits are restored when a lesson reopens (Section 4.3.5).
T-FR-6.1	Demonstration	Each locale is an independent content tree (Section 4.3.2).
T-FR-6.2	Demonstration	A switcher changes language and remembers the choice across visits (Section 4.3.2).

Table 10: Verification of the Interactive Tour functional requirements.

6.2.1.2 NON-FUNCTIONAL REQUIREMENTS

ID	METHOD	VERIFICATION
T-NFR-1.1	Measurement	The debounced compile schedule keeps the preview update within the 100 ms window on standard desktop hardware (Section 5.3.2.3).
T-NFR-1.2	Inspection	The WASM modules load lazily, so the lesson text is interactive before the compiler finishes loading (Section 4.3.1).
T-NFR-2.1	Demonstration	The build emits a fully static bundle published to GitHub Pages (Section 5.4.0.2).
T-NFR-2.2	Inspection	Compilation, formatting, and linting run entirely in the browser, with no network calls leaving the client (Section 4.3.1).
T-NFR-3.1	Demonstration	The tour loads and runs without any login or backend (Section 4.3.1).
T-NFR-4.1	Demonstration	Static hosting on GitHub Pages keeps the public URL reachable without credentials (Section 5.4.0.2).

Table 11: Verification of the Interactive Tour non-functional requirements.

6.2.2 COLLABORATIVE EDITOR

6.2.2.1 FUNCTIONAL REQUIREMENTS

ID	METHOD	EVIDENCE
C-FR-1.1	Demonstration	OAuth sign-in via GitHub, GitLab, and Google (Section 4.4.4).
C-FR-2.1	Demonstration	Concurrent edits merge through the shared Y.Doc (Figure 26).
C-FR-2.2	Demonstration	Collaborator cursors and names are carried on the awareness channel (Section 4.4.8 , Section 5.5.5.3).
C-FR-2.3	Demonstration	Deleting an open file redirects and notifies the affected user (Section 5.5.5.4).
C-FR-2.4	Demonstration	The same redirect applies to a deleted folder (Section 5.5.5.4).
C-FR-3.1	Demonstration	Shareable invite links with read-only or edit access (Section 5.5.3).
C-FR-3.2	Demonstration	Invite links expire and can be revoked (Section 5.5.3).
C-FR-3.3	Demonstration	The owner edits or revokes member permissions, enforced by the access macro (Section 4.4.2 , Listing 41).
C-FR-4.1	Demonstration	The dashboard lists projects with first-page previews (Section 4.4.5).
C-FR-5.1	Demonstration	The three-pane workspace embeds the editing core over multi-file projects (Section 4.4.7 , Listing 48).
C-FR-6.1	Demonstration	Templates are chosen from the Typst Universe catalog (Section 4.4.6).
C-FR-6.2	Demonstration	A template materializes files, assets, and the compile entry into the Y.Doc (Section 4.4.6).
C-FR-7.1	Demonstration	Binary assets can be uploaded and become referenceable by path (Section 4.4.3.1).
C-FR-7.2	Test	Asset references are part of the synchronized project state (Section 4.4.3.1).
C-FR-7.3	Inspection	Assets are capped at 10 MB and fonts at 30 MB, each restricted to its own allow-list of MIME types (Listing 46).

Table 12: Verification of the Collaborative Editor functional requirements.

6.2.2.2 NON-FUNCTIONAL REQUIREMENTS

ID	METHOD	VERIFICATION
C-NFR-1.1	Measurement	The browser's network throttling was used to cap the connection at 1 Mbps with a sub-100 ms round-trip, and remote edits still propagated to all clients within the 200 ms budget.
C-NFR-2.1	Test	Forced disconnections reconcile automatically on re-connect via the Yjs sync layer, with no lost updates (Figure 26).
C-NFR-3.1	Inspection	The OAuth flow is delegated to the better-auth library, which attaches and validates the state parameter on every request, mitigating CSRF without custom code.
C-NFR-3.2	Inspection	Invite tokens carry 256 bits of cryptographically secure randomness and are stored only as a hash (Listing 45).
C-NFR-3.3	Inspection	The collaboration server validates a user's session before they join a document room (Figure 27, Listing 43).
C-NFR-4.1	Demonstration	Reaching an editable document takes three clicks: <i>New project</i> , then <i>Create</i> in the dialog after typing a name, then opening the new project from its card.
C-NFR-5.1	Inspection	The backend is built on Elysia/Bun with Hocuspocus and Drizzle ORM against PostgreSQL (Figure 27).
C-NFR-5.2	Inspection	The frontend uses SolidJS with y-codemirror.next for collaborative editing (Figure 28).
C-NFR-5.3	Inspection	Client-server communication is typed end-to-end through Elysia Eden (Section 5.5.7).

Table 13: Verification of the Collaborative Editor non-functional requirements.

6.3 RESULTS

All three deliverables were completed and put to use. The shared `typst-web` library is published to npm and drives both applications, the Interactive Tour is deployed as a static site on GitHub Pages, and the Collaborative Editor runs as a live multi-user instance. The requirements verification above pairs each functional and non-functional requirement

with the evidence that it is met, and the post-build survey (Section 6.1), though based on a small sample, indicated that the Tour reaches its learning goal. This section presents the delivered applications as a reader would encounter them.

6.3.1 INTERACTIVE TOUR

The Tour opens each lesson in a split-pane workspace, with the explanation on one side and a live editor with an instant preview on the other, so a reader moves from reading to writing Typst without leaving the page (Figure 32).

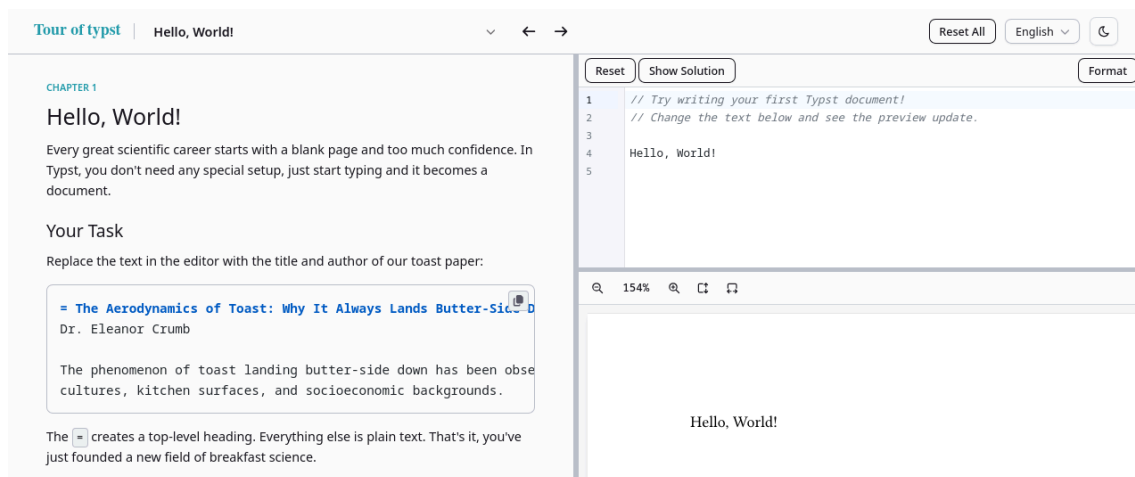


Figure 32: A tour lesson, pairing the explanation with a live editor and preview

When a reader is stuck, the *Show Solution* action reveals the reference code with the lines that differ from the current attempt highlighted, so the change to make is immediate rather than buried in a wall of text (Figure 33).

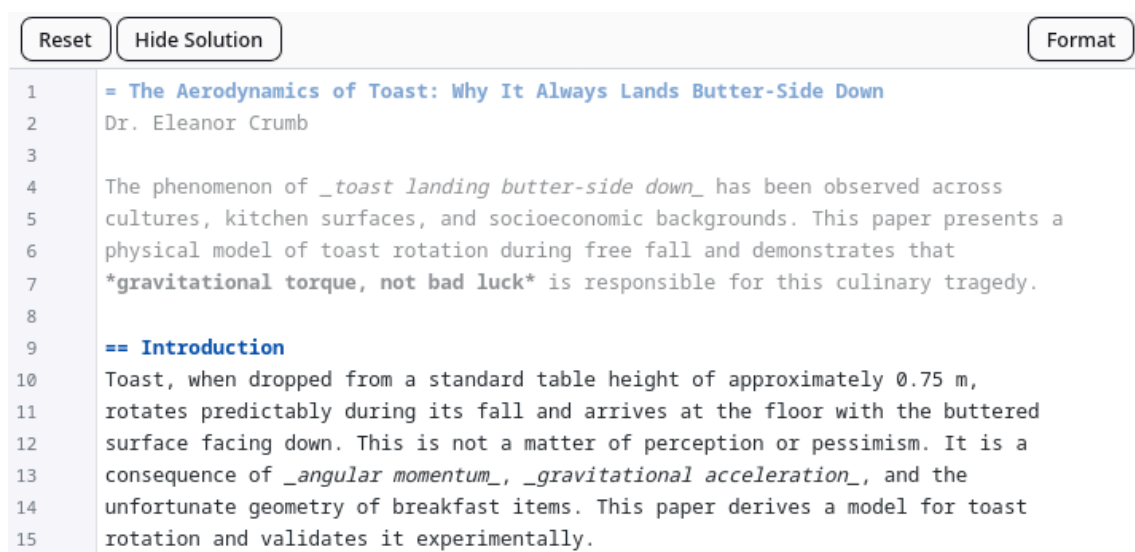


Figure 33: The solution view, highlighting the lines that differ from the current code

6.3.2 COLLABORATIVE EDITOR

Access begins at a sign-in page that delegates to OAuth providers, GitHub, GitLab, and Google, so a user signs in with an account they already hold (Figure 34).

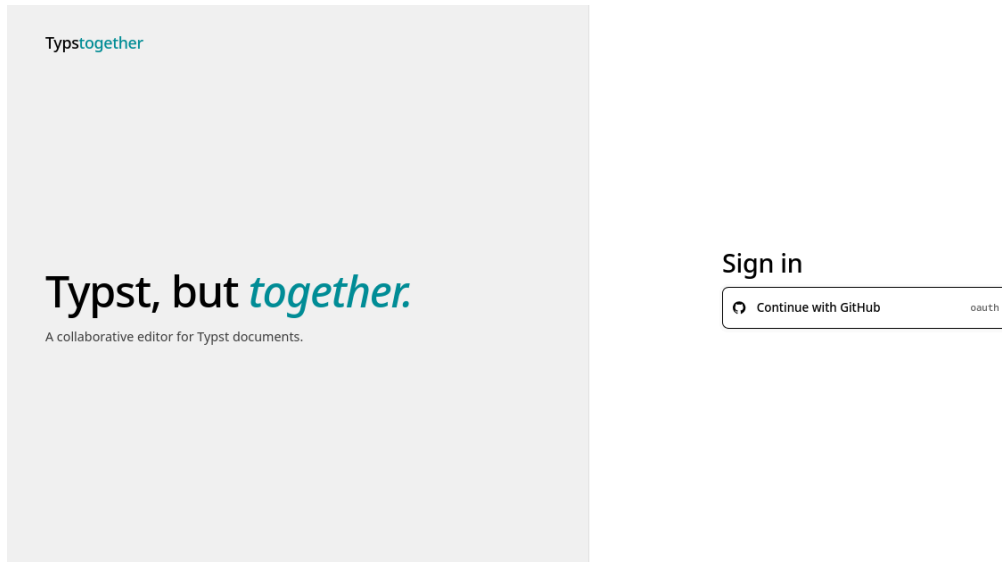


Figure 34: The sign-in page, offering the configured OAuth providers

After signing in, the editor opens on a dashboard that lists the user's projects, each with a first-page preview rendered from its current content (Figure 35).

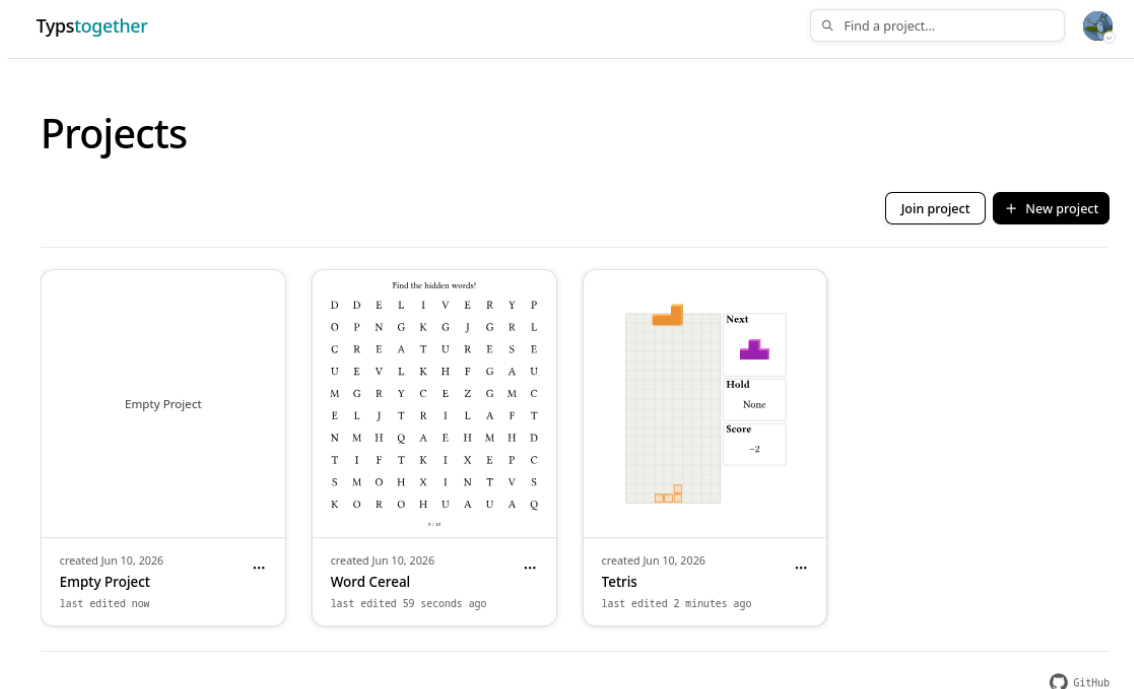


Figure 35: The dashboard, listing projects with live first-page previews

A new project starts either blank or from a Typst Universe template, picked from a catalog of previews so a user can begin from a styled document rather than an empty file. An assortment of tags is available to help the user find templates (Figure 36).

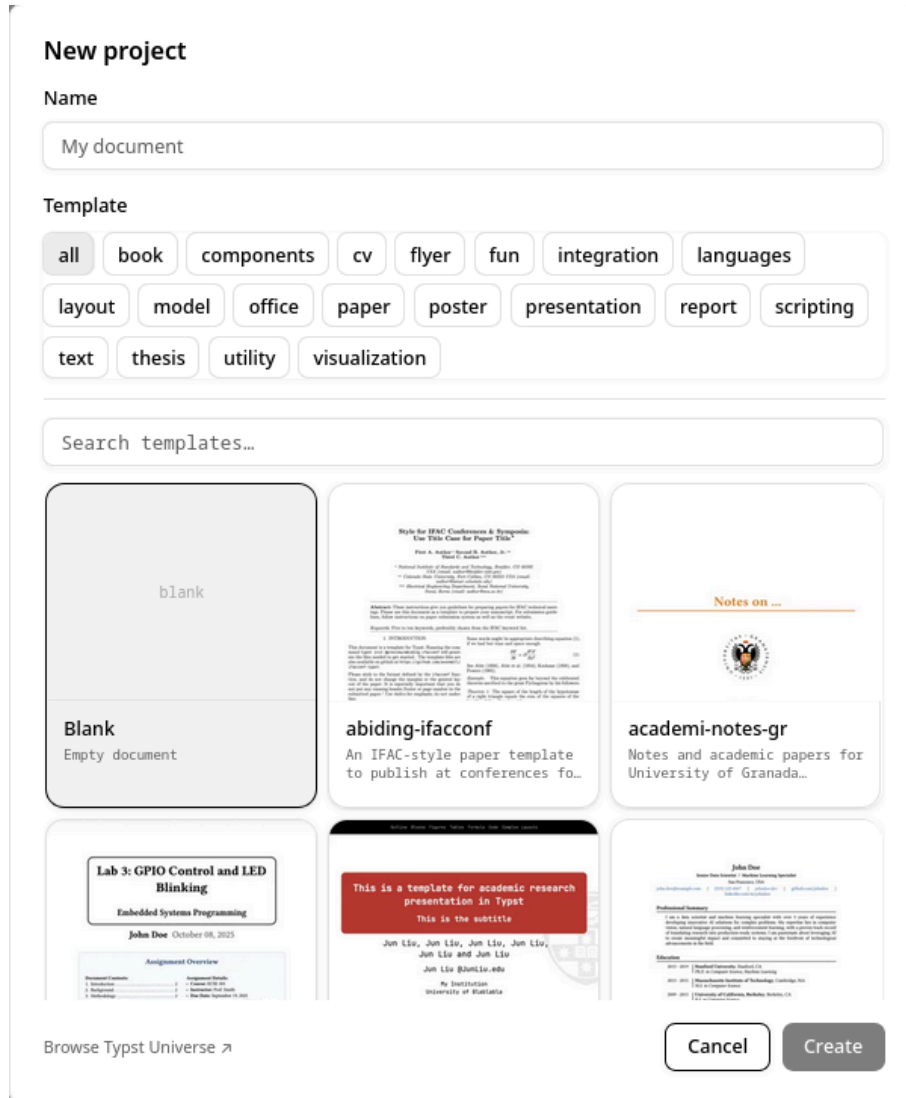


Figure 36: The new-project view, starting from a Typst Universe template

Inside a project, the three-pane workspace pairs a file tree, the shared editor, and the preview. Several people edit one document at once, where changes merge through the underlying CRDT, and each collaborator's cursor and name are shown inline, so it is clear who is working where (Figure 37).

Evaluation

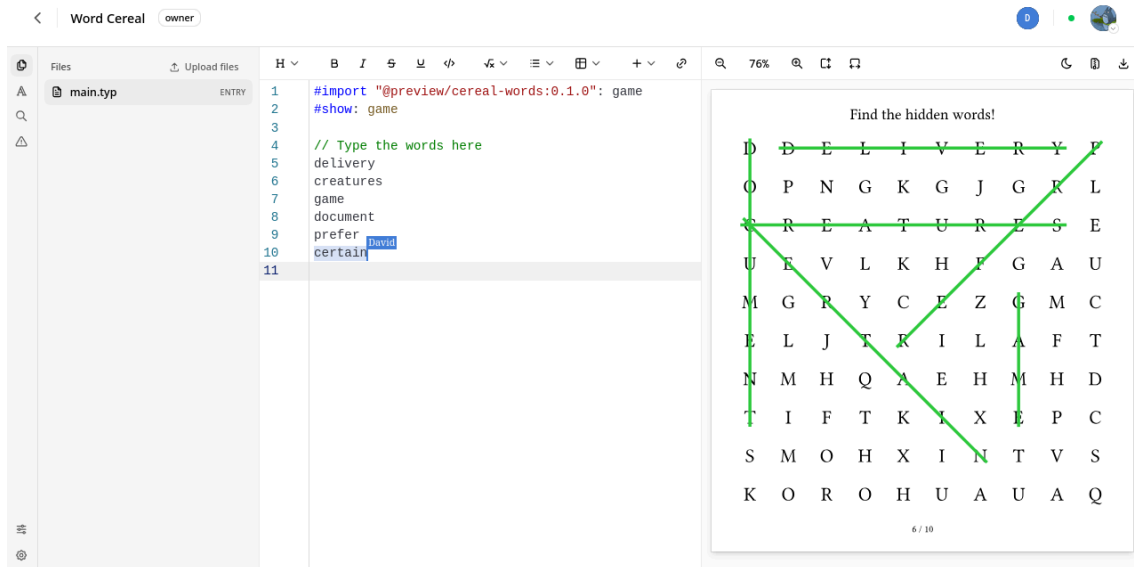


Figure 37: The shared editor, showing two collaborators

The file pane is a full project file system rather than a flat list. A right-click on any file or folder opens a context menu to create, rename, duplicate, or delete entries, and files or whole folders are added by dragging them onto the tree or through the *Upload files* button. It accepts Typst sources alongside the assets a document pulls in, images, and data files such as CSV, JSON, or TOML, so a project keeps everything it needs in one place (Figure 38).

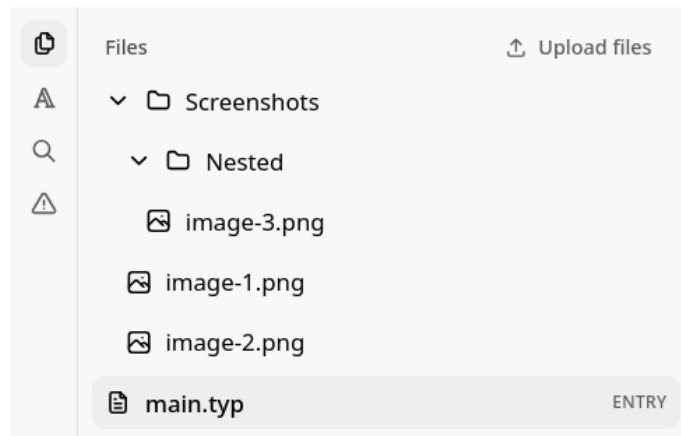


Figure 38: The file explorer, showing the project file tree and upload action

A separate fonts panel lets a project carry its own typefaces rather than relying on a fixed set. A user uploads `.ttf`, `.otf`, or `.ttc` files by dragging them in or through the *Upload fonts* button, and the panel groups them by family. Copying a family name yields exactly the string a `#set text(font: ...)` rule expects, and an uploaded font is shared with every collaborator and made available to the compiler (Figure 39).

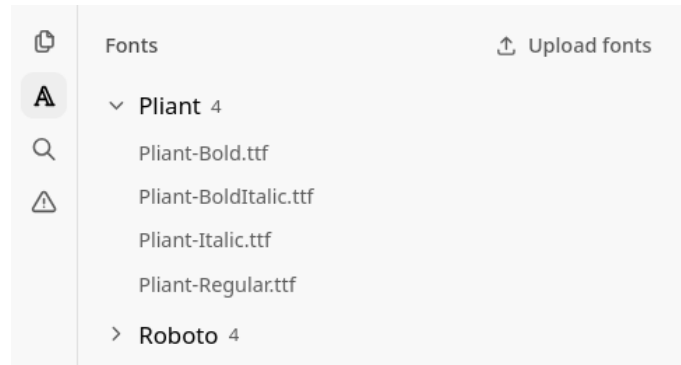


Figure 39: The font explorer, listing uploaded typefaces grouped by family

Search spans the whole project rather than only the open file. Matches are listed per file with their line numbers and surrounding context, and selecting one opens that file and scrolls to the match. Case-sensitive and whole-word toggles narrow the query, and an optional replace field applies a substitution to a single match or to every match across the project (Figure 40).

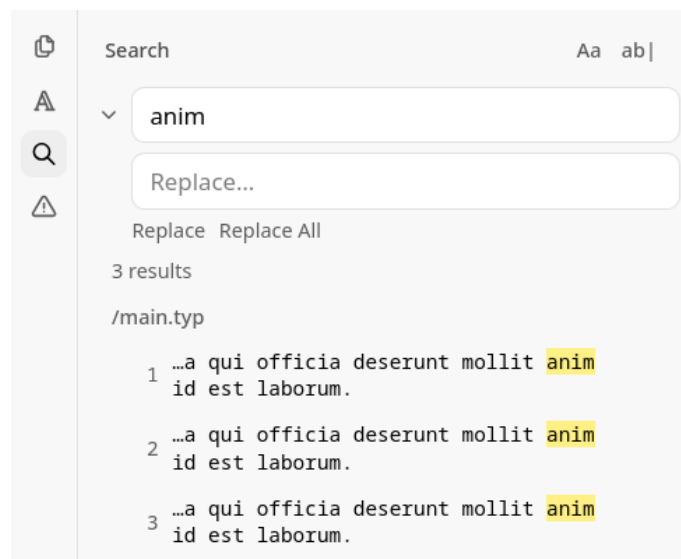


Figure 40: The search panel, listing matches grouped by file with context

A problems panel surfaces the compiler's own diagnostics rather than leaving the reader to spot a failed render. Errors and warnings are listed and grouped by file, each with its message and the line and column it points to, and the same positions are marked inline in the editor. Selecting an entry jumps the cursor straight to the reported location, so a mistake is found and fixed without hunting through the source (Figure 41).



Figure 41: The problems panel, listing compiler errors and warnings

A project is shared through an invite link scoped to read-only or edit access, which the owner can revoke at any time. Once generated, the link can be shared with others or be revoked at any time (Figure 42).

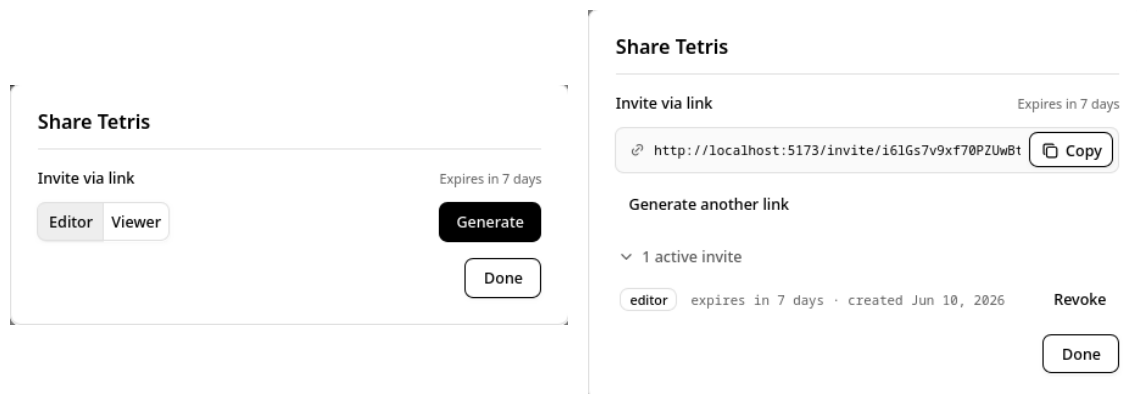


Figure 42: The sharing dialog, before generating an invite link (left) and after (right)

7

CONCLUSION & FUTURE WORK

7.1 CONCLUSION

This thesis set out to close two gaps in the Typst ecosystem: the absence of a guided way to learn the language, and the lack of an open way to collaborate on a document without self-hosting tooling or the use of a closed-source service. It addressed them with two complementary web applications, the interactive *Tour of Typst* and a real-time Collaborative Editor, both built on a shared in-browser Typst engine, the `typst-web` library.

All three deliverables were completed. The `typst-web` library wraps the Typst compiler in a single WebAssembly module and is published on npm, where it powers both applications and is available to the wider community. The Tour teaches Typst from a blank editor to a working document and is deployed as a fully static site, and the post-build survey ([Section 6.1](#)), though based on a small sample, points in favor of it meeting its learning goal. The Collaborative Editor runs as a live multi-user instance with authentication, project management, and shared assets, giving Typst an open-source alternative to the closed-source `typst.app`. The requirements verification pairs each functional and non-functional requirement with the evidence that it is met.

Beyond the individual tools, the work contributes a reusable foundation to the Typst ecosystem. By keeping the engine free of I/O and deterministic over its inputs and publishing it as a library, the same core can underpin tools the authors did not build, and the two applications are themselves the first demonstration that it generalizes across very different products. The result is a coherent set of tooling that lowers the barrier both to learning Typst and to writing it together. What remains, set out below, is depth rather than essentials: the applications are usable as built, and the clearest next steps are to flesh out the Collaborative Editor and to maintain and expand the `typst-web` foundation that both applications share.

7.2 FUTURE WORK

The shared library and the two applications reached different points of maturity, and their natural next steps differ accordingly.

7.2.1 SHARED LIBRARY

The `typst-web` library is published and in use, so its future work is mostly about keeping pace and widening reach. It pins a specific Typst version, so the most ongoing task is tracking new compiler releases as they land. Beyond that, broadening the public API and documenting it would make the library easier to adopt for tools beyond the two built here, which is the clearest measure of the foundation paying off.

Package fetching is one area where the design may eventually be revisited. The library resolves and downloads Typst Universe packages eagerly on the JavaScript side, because the WebAssembly core cannot perform these asynchronous fetches itself without pulling in an expensive shim such as Asyncify. JavaScript Promise Integration (JSPI) is expected to make asynchronous calls into WebAssembly possible directly once it ships and gains `wasm-bindgen` support, which would let the core fetch packages on demand rather than relying on the host to prefetch them. That would move an I/O concern back into the engine, though, in tension with the deliberate boundary that keeps it free of I/O and side effects ([Section 4.2](#)), so it is a trade-off to weigh rather than an unambiguous win.

The module is also monolithic in what it bundles. The Rust core is already organized by capability, with separate modules for compilation, rendering, formatting, syntax highlighting, and the IDE features, but all of them are compiled into a single WebAssembly binary that every consumer ships in full. Adding Cargo feature flags that gate these capabilities would let a consumer build only what it needs, for example a render-only binary for a tool that just displays documents, trading the convenience of one build for a smaller download. The existing module boundaries make this a natural extension rather than a rewrite.

7.2.2 INTERACTIVE TOUR

The Tour is essentially feature-complete, and what remains is incremental and already tracked. A favicon is still to be added, and the content can grow in two directions, more chapters as Typst itself gains features worth teaching, and more languages beyond the current English and German. Because each locale is an independent content tree ([Section 4.3.2](#)), adding a language is largely a content contribution rather than an engineering effort, so the scope of the Tour can expand without touching its architecture.

7.2.3 COLLABORATIVE EDITOR

The Collaborative Editor covers the core collaborative authoring workflow, but it leaves the most room to grow, and a backlog of enhancements is tracked as open issues on its repository. They fall into a few themes:

- **Authoring aids** that bring the editor closer to a full writing environment: an outline panel, a word and character count, a math and special-character browser, a package browser, in-editor code formatting, and a spellchecker with dictionaries and suggestions.
- **Reference management**, both inserting references through the language server and integrating an external manager such as Zotero.
- **Collaboration and workflow** features that go beyond live editing. A review mode with comments and suggestions, and version-control integration with GitHub and GitLab.
- **Performance**, by caching the result of expensive image renders. At present such images are re-rendered on every compile, so rendering each image once and reusing the stored result would avoid this repeated work.
- **Scaling to large projects**, which hold many assets such as images and fonts. These projects currently make the editor freeze during some interactions, for example clicking links in the preview or uploading further files, and a collaborator joining such a project can see compilation fail because the images do not load reliably under heavy load.
- **Larger explorations** such as an AI writing assistant, alongside smaller refinements like keeping the previous preview visible on reload until the new one is ready for a smoother page refresh.

7.2.4 USABILITY EVALUATION

Beyond features, a valuable next step is a structured usability study of the Collaborative Editor. Unlike the Tour, the editor has not yet been put in front of users in any systematic way, so its design rests on our own judgment rather than evidence. A proper evaluation would stand up a dedicated test environment, define explicit evaluation goals and a set of representative task scenarios, and observe a larger and more diverse group working through them. The interface is intentionally simple and reflects our best judgment of what works, but that judgment is exactly what controlled testing can confirm or challenge: real sessions tend to surface friction that designers, too close to the product, no longer notice, and they would replace intuition about the editor's usability with evidence.

A FIGURES

Figure 1	Prior tool experience versus desired tool for thesis writing	4
Figure 2	Current familiarity levels with Typst	5
Figure 3	Support requirements for adopting Typst in a thesis project.	5
Figure 4	System context of the Interactive Tour	7
Figure 5	System context of the Collaborative Editor	10
Figure 6	Layered architecture: typsten core, typst-web, and the two apps	13
Figure 7	High-level architecture of typst-web	14
Figure 8	High-level architecture of the Tour of Typst	15
Figure 9	Mockup of the chapter navigation	16
Figure 10	Mockup of the three-panel “read, try, see” lesson layout	17
Figure 11	High-level architecture of the Collaborative Editor	18
Figure 12	Document data model	19
Figure 13	Mockup of the sign-in page with OAuth providers	20
Figure 14	Mockup of the project dashboard	21
Figure 15	Mockup of the new-project dialog	22
Figure 16	Mockup of the editor layout	23
Figure 17	Mockup of the editor’s presence indicators	23
Figure 18	A week-long hold outlasts most malicious-release detection	26
Figure 19	Shared CI/CD: checks gate every PR; merging <code>main</code> deploys	28
Figure 20	GitHub dispatches the deploy job to a runner on the production host	30
Figure 21	Core types of <code>typsten</code> , built on the upstream Typst crates	32
Figure 22	Where each <code>Project</code> method comes from	33
Figure 23	The jump pair bridges editor and preview	38
Figure 24	High-level class diagram of <code>TypstProject</code>	39
Figure 25	One edit-to-preview cycle across the editor, <code>TypstProject</code> , and the worker .	44
Figure 26	One-way sync from the shared document into a <code>TypstProject</code>	51
Figure 27	Collaboration server that persists the <code>Y.Doc</code> and removes orphaned blobs .	62
Figure 28	Frontend wiring where the shared <code>Y.Doc</code> drives editor and compiler	65
Figure 29	Toolbar with quick actions	67
Figure 30	Self-reported confidence in writing Typst after completing the tour.	72
Figure 31	Likelihood of choosing Typst for the next academic project after the tour. .	73

Figures

Figure 32	A tour lesson, pairing the explanation with a live editor and preview	78
Figure 33	The solution view, highlighting the lines that differ from the current code .	78
Figure 34	The sign-in page, offering the configured OAuth providers	79
Figure 35	The dashboard, listing projects with live first-page previews	79
Figure 36	The new-project view, starting from a Typst Universe template	80
Figure 37	The shared editor, showing two collaborators	81
Figure 38	The file explorer, showing the project file tree and upload action	81
Figure 39	The font explorer, listing uploaded typefaces grouped by family	82
Figure 40	The search panel, listing matches grouped by file with context	82
Figure 41	The problems panel, listing compiler errors and warnings	83
Figure 42	The sharing dialog, before generating an invite link (left) and after (right) .	83

B TABLES

Table 1	Primary objectives of the thesis	1
Table 2	Functional requirements of the Interactive Tour	7
Table 3	Non-functional requirements of the Interactive Tour	9
Table 4	Functional requirements of the Collaborative Editor	10
Table 5	Non-functional requirements of the Collaborative Editor	11
Table 6	Editor events to one TypstProject query and back to editor state	46
Table 7	Key dependencies of the Tour of Typst	53
Table 8	The files of the en:hello-world lesson, one per page part	54
Table 9	Backend dependencies of the Collaborative Editor	59
Table 10	Verification of the Interactive Tour functional requirements.	74
Table 11	Verification of the Interactive Tour non-functional requirements.	75
Table 12	Verification of the Collaborative Editor functional requirements.	76
Table 13	Verification of the Collaborative Editor non-functional requirements.	77

C LISTINGS

Listing 1	Content directory layout of a single lesson	15
Listing 2	Formatting actions tested with a before/after marker notation	27
Listing 3	A workflow step opens the version PR or publishes	30
Listing 4	Production Caddyfile routing both REST and WebSocket traffic	31
Listing 5	The wasm-bindgen facade delegates to a plain-Rust core	33
Listing 6	The size-oriented release profile for the WebAssembly build	34
Listing 7	The Rust struct and the TypeScript interface tsify generates from it	34
Listing 8	The worker wrapper crosses Comlink, never the wasm Project	35
Listing 9	The Typst world reads from an in-memory VFS	36
Listing 10	An edit refreshes the cached source in place	37
Listing 11	Compilation returns diagnostics as data, keeps the last document	37
Listing 12	Editor queries defer to typst-ide or typstyle	38
Listing 13	TypstProject.create spawns the worker and awaits the engine	40
Listing 14	Writes dedup against the tracked mirror before touching the worker	41
Listing 15	A version counter drops stale results from slower, superseded compiles .	42
Listing 16	Fetching a package: download, decompress, unpack, and feed the VFS .	43
Listing 17	Rendered pages pair each SVG with its dimensions	43
Listing 18	Offset conversion between engine bytes and CodeMirror code units	45
Listing 19	The navigator hit-tests a click against the live page elements	45
Listing 20	Wiring an editor with one project, the setup bundle, and file-path facet	46
Listing 21	Compile-sync only mirrors edits into the project, never compiles	47
Listing 22	Diagnostics are filtered to the active file before dispatch	47
Listing 23	Highlighting requests only the viewport and drops stale results	48
Listing 24	Each span becomes a mark decoration tagged with its typ-* class	49
Listing 25	Engine completions mapped to CodeMirror options, kinds, and snippets .	49
Listing 26	Hover renders the engine tooltip, reusing editor highlighting	50
Listing 27	Formatting applies a minimal diff, preserving cursor and undo	50
Listing 28	One write in flight at a time, always writing the latest state	51
Listing 29	The single-file adapter observes the text and writes the latest on change .	52
Listing 30	The multi-file write step diffs the dirty paths into project calls	52
Listing 31	Every lesson file is bundled at build time as a raw string	54

Listings

Listing 32	Paths are re-keyed by locale and chapter for one-lookup access	55
Listing 33	A locale's <code>tour.ts</code> declares its own ordered parts and chapters	55
Listing 34	A custom <code>marked</code> renderer highlights blocks and stores their source	56
Listing 35	<code>EditorShell</code> creates one project and shares it through context	57
Listing 36	Each lesson reseeds the project, then streams pages to the preview	58
Listing 37	A line diff drives a decoration that dims the unchanged lines	58
Listing 38	The <code>typst</code> together monorepo packages	59
Listing 39	The four-file module convention	60
Listing 40	Factory for project-access macros, parameterized by allowed roles	61
Listing 41	Project-access macros	61
Listing 42	A per-user cooldown macro, namespaced by the caller's id	62
Listing 43	The socket authenticates the session and marks viewers read-only	63
Listing 44	Deleting a reference in the shared document removes its blob	63
Listing 45	An invite token is persisted only as a SHA-256 hash	64
Listing 46	Assets cap at 10 MB, fonts at 30 MB, each with a MIME allow-list	65
Listing 47	The shared <code>files</code> map mirrors into a local <code>TypstProject</code>	66
Listing 48	The file tree is derived from path prefixes, with no stored folders	67
Listing 49	Search scans each shared <code>Y.Text</code> line by line for <code>indexOf</code> hits	68
Listing 50	Typed errors map to status codes through a single handler	69
Listing 51	Backend route declaring its response shape via a <code>TypeBox</code> model	70
Listing 52	Frontend call through Eden Treaty with the fully inferred result	70

D BIBLIOGRAPHY

- [1] "Typst App." [Online]. Available: <https://typst.app/>
- [2] Simon Brown, "The C4 model for visualization software architecture." [Online]. Available: <https://c4model.com/>
- [3] Michelene T. H. Chi and Ruth Wylie, "The ICAP Framework: Linking Cognitive Engagement to Active Learning Outcomes," Oct. 28, 2014.
- [4] "Tour of Go." [Online]. Available: <https://go.dev/tour/>
- [5] "Tour of Rust." [Online]. Available: <https://tourofrust.com/>
- [6] "Overleaf." [Online]. Available: <https://www.overleaf.com/>
- [7] "TeXlyre." [Online]. Available: <https://github.com/TeXlyre/texlyre>
- [8] "typstyle." [Online]. Available: <https://typstyle-rs.github.io/typstyle/>
- [9] "typst-web." [Online]. Available: <https://github.com/vedivad/typst-web>
- [10] "Yjs." [Online]. Available: <https://yjs.dev/>
- [11] "Comlink." [Online]. Available: <https://github.com/GoogleChromeLabs/comlink>
- [12] "SolidJS." [Online]. Available: <https://www.solidjs.com/>
- [13] "Elysia." [Online]. Available: <https://elysiajs.com/>
- [14] "Bun." [Online]. Available: <https://bun.sh/>
- [15] "PostgreSQL." [Online]. Available: <https://www.postgresql.org/>
- [16] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-Free Replicated Data Types," in *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, 2011, pp. 386–400.
- [17] "concurrently." [Online]. Available: <https://www.npmjs.com/package/concurrently>

Bibliography

- [18] "TypeScript." [Online]. Available: <https://www.typescriptlang.org/>
- [19] "codebook." [Online]. Available: <https://github.com/blopper/codebook>
- [20] "Dependabot." [Online]. Available: <https://docs.github.com/en/code-security/tutorials/secure-your-dependencies>
- [21] "Zed." [Online]. Available: <https://github.com/zed-industries/zed>
- [22] "Better Auth." [Online]. Available: <https://better-auth.com/>
- [23] "Oxlint." [Online]. Available: <https://oxc.rs/>
- [24] "WebAssembly." [Online]. Available: <https://webassembly.org/>
- [25] "wasm-bindgen." [Online]. Available: <https://github.com/rustwasm/wasm-bindgen>
- [26] "wasm-pack." [Online]. Available: <https://github.com/rustwasm/wasm-pack>
- [27] "Serde." [Online]. Available: <https://serde.rs/>
- [28] "Tsify." [Online]. Available: <https://github.com/madonoharu/tsify>
- [29] "comemo." [Online]. Available: <https://github.com/typst/comemo>
- [30] "typst-ide." [Online]. Available: <https://crates.io/crates/typst-ide>
- [31] "typstyle." [Online]. Available: <https://crates.io/crates/typstyle-core>
- [32] "fflate." [Online]. Available: <https://github.com/101arrowz/fflate>
- [33] "nanotar." [Online]. Available: <https://github.com/unjs/nanotar>
- [34] "CodeMirror." [Online]. Available: <https://codemirror.net/>
- [35] "Lezer." [Online]. Available: <https://lezer.codemirror.net/>
- [36] "TextMate." [Online]. Available: https://macromates.com/manual/en/language_grammars
- [37] "Vite." [Online]. Available: <https://vite.dev/>
- [38] "Marked." [Online]. Available: <https://marked.js.org/>
- [39] "Shiki." [Online]. Available: <https://shiki.style/>

Bibliography

[40] "Redis." [Online]. Available: <https://redis.io/>

[41] "Pino." [Online]. Available: <https://getpino.io/#/>