Making blog posts available as PDFs with minitype

Katsutoshi Seki | Published: September 14, 2026 | Source: https://sekika.github.io/2026/09/14/minitype-pdf/

Blog posts on this site can now also be read as PDFs. Selecting "View PDF" in an article header opens an A4, two-column version of the post in a new tab.

PDFs are useful for printing, saving an article for later, and reading it in a stable layout that does not depend on screen width or browser zoom. Text is emitted with embedded fonts, so it can be selected, searched, and copied in a PDF viewer. Links in the PDF are preserved as blue hyperlinks, so readers can still open the source article and its references.

Not every post is converted. Pages whose main purpose is browser interaction, such as the 15 puzzle, Pyodide, and Canvas-based pages, would lose their essential behavior in a PDF and therefore do not show a PDF link. Ordinary explanatory posts are converted in both Japanese and English.

1.Using minitype

PDFs are produced with minitype, a headless typesetting engine available as a TypeScript library. It provides layout features including Japanese line-breaking rules, vertical writing, ruby text, and multi-column layouts.

Since the posts in this blog are written in Markdown, the minitype Markdown plugin reads each post file and converts headings, paragraphs, lists, code blocks, tables, images, and links into minitype layout elements. Node.js runs the PDF-generation step.

Some posts also contain raw HTML. This blog converts an img tag that refers to a file in the repository into a Markdown image. PNG, JPEG, and similar images are placed directly, while SVG images are first rasterized to PNG so that they can be embedded in the PDF. It also converts [link text](...) into a Markdown link, which is displayed in blue and remains clickable in the PDF. HTML in code blocks and Jekyll highlight blocks is left unchanged because it is source code shown for explanation.

Existing posts that use layout: math or layout: katex are also handled as mathematical documents. The generator recognizes display equations written with $$ ... $$ or [[...]], and inline equations written with $ ... $, then typesets them as minitype math elements. Repeated backslashes in existing mathematical source are normalized as TeX commands. The same notation in code blocks, inline code, and Jekyll highlight blocks remains unchanged as explanatory source code.

This is not a screenshot-like conversion of the published HTML page. Instead, the Markdown source is typeset again for a PDF page. Browser buttons and JavaScript results are therefore not copied to the PDF; the aim is a readable version of the article itself.

2.Typesetting for this blog

The generation program is generate-pdfs.mjs. It uses A4 portrait pages in horizontal writing, sets page margins, and places the article body in two columns. Titles are centered; the author, publication date, and source URL are right-aligned; and page numbers sit near the foot of each page without reducing the body's bottom margin.

Markdown links become PDF link annotations and are colored blue. Some PDF viewers display minitype's square annotations as red boxes or selected objects, so the generation program removes those square annotations after output while preserving normal hyperlink annotations. Text is not converted to outlines: the PDF embeds a subset of the required fonts, allowing text to be selected, searched, and copied.

Article PDFs are normally written to a path corresponding to the article URL:

```
/pdf/YYYY/MM/DD/slug.pdf
```

For example, the PDF for /2024/05/12/Bootstrap5/ is stored at /pdf/2024/05/12/Bootstrap5.pdf. The Japanese and English article headers construct this URL automatically, so each post does not need to specify a link destination.

3.Deciding which posts are not converted

Ordinary posts are PDF targets by default. A post can be excluded explicitly with this front matter:

```
pdf: false
```

As a safeguard, the generator also examines article content. It excludes posts that contain executable or interactive HTML outside code examples, including script, canvas, form, input, textarea, select, button, and iframe. It ignores code fences, inline code, and Jekyll highlight blocks during this check, so an article that merely explains JavaScript remains eligible for PDF output.

This distinguishes an interactive article such as the 15 puzzle or a browser-based Pyodide environment from an ordinary explanatory article about JavaScript or Pyodide.

4.Keeping PDFs current with pre-commit

Creating PDFs manually would make it easy for a PDF to become outdated after its article changes. To prevent that, the Git pre-commit hook generates PDFs when posts are committed.

Install the Node.js packages and register the hook once:

```
cd tools/minitype-pdf
npm install
cd ../..
make -C setup install-hook
```

When a post in _posts/ is added or changed, the hook first updates js/index.js for site search. It then creates every miss-

ing PDF and regenerates the PDF for the post in the commit. Generated files are staged automatically. The draft-publication pub script performs the same PDF-generation step for a newly published draft.

To generate all currently missing PDFs manually, run this command from the repository root:

```
node tools/minitype-pdf/generate-pdfs.mjs --missing
```

Further setup details and the exclusion rules are documented in setup/Readme.md.

5.The role of GitHub Pages

The standard Jekyll build on GitHub Pages publishes static files already present in the repository, including HTML, CSS, JavaScript, images, and PDFs. PDF generation with minitype, on the other hand, requires Node.js and is not part of that Jekyll build.

This site generates PDFs locally through the pre-commit hook and commits them with their corresponding articles. GitHub Pages then serves the PDFs like any other static file. This separation keeps the GitHub Pages configuration simple while maintaining each article and its PDF in the same Git history.

The next step is to review PDFs for posts containing substantial mathematics, complex tables, or many images, and refine the typesetting where needed.