Bullseye

Hit the target (HTML or paged/PDF) when styling your Typst document

v0.1.0 July 27, 2025

https://github.com/SillyFreak/typst-bullseye

Clemens Koza

Contents

I Introduction

Bullseye supports you in writing packages and documents that target multiple outputs, i.e. currently (Typst 0.13) `"paged"` (PDF, image) and `"html"`.

This package consists of two parts:

- At the foundation, it contains a wrapper around the currently unstable Typst features for target detection and HTML output.
- Built on top of that are a few helpful functions that allow package and document authors to easily write content and show rules that behave differently based on the target.

This document will start with an example blog post as a use case and demonstrate the problems of multi-target documents, then look at the helper functions in Section II as it is more immediately important to users, and then use that to motivate the foundation described in Section III.

I.a A hypothetical blog post

Let's say you're writing a fairly simple blog. You draft a Typst document with its content, which looks roughly like this:

```typ
// there are no page breaks on the web
#set page(height: auto)
// make code pretty in the preview; the blog platform has JS-based code styling
#import "@preview/codly:1.3.0"
#show: codly.codly-init

= My blog post

#lorem(5)

#figure(
  image("image.svg", alt: "a rectangle"),
  caption: [A rectangle]
)

#lorem(5)

```typ
#let x = 0
```

// in HTML, this should be a link with `href="#"`, which navigates back up
(back to top)
```

this is still incomplete, but looks good in the preview. You then try to compile it to HTML:

```
$ typst compile --features html --format html blog-post.typ
...
error: page configuration is not allowed inside of containers
   ┌─ blog-post.typ:2:1
   │
 2 │ #set page(height: auto)
   │  ^^^^^^^^^^^^^^^^^^^^^
```

That didn't work, but since `set page` is just for the preview, you can just remove this line and try again:

```
$ typst compile --features html --format html blog-post.typ
...
warning: block was ignored during HTML export
   ┌─ blog-post.typ:12:2
   │
12 │   image("image.svg", alt: "a rectangle"),
   │   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
warning: block was ignored during HTML export
      ┌─ @preview/codly:1.3.0/src/lib.typ:1744:8
      │
1744 │ ┌         grid(
1745 │ │           columns: if has-annotations {
     · │
1780 │ │           ..footer,
1781 │ │         )
     │ └─────────^
```

It compiles with some important warnings: the resulting HTML file has no image and no code!

```html
<h2>My blog post</h2>
<p>Lorem ipsum dolor sit amet.</p>
<figure>
  <figcaption>Figure 1: A rectangle</figcaption>  <!-- oops! -->
</figure>
<p>Lorem ipsum dolor sit amet.</p>
<div></div>  <!-- oops! -->
<p>(back to top)</p>  <!-- not a lik yet -->
```

I.b Target-conditional code

Our issue boils down to us wanting to conditionally apply certain styling and content. Typst's `target()` function can be used to (contextually) determine what kind of output some content is rendered in. Using that, you could rewrite your code like this (this is the code Bullseye will subsequently simplify, so feel free to only skim it):

```typ
#show: doc => context if target() == "html" {
  // apply this styling for html output:
  // replace rendered images with <img> tags
  show image: img => html.elem("img", attrs: ("src": img.source, "alt": img.alt))
  // instead of using codly, wrap code in a <pre class="language-..."> element
  show raw.where(block: true): code => {
    html.elem("pre", attrs: ("class": "language-" + code.lang), code)
  }
  doc
} else {
  // apply this styling for "regular" output:
  set page(height: auto)
  import "@preview/codly:1.3.0"
  show: codly.codly-init
  doc
}
```

```
17
18 // a link that can be used at the bottom of the post
19 #let back-to-top = html.elem("a", attrs: (href: "#"))[(back to top)]
20
21 = My blog post
…                                                    …
36 #context if target() == "html" { back-to-top }
```

That's quite a bit of code, but …

```
1 $ typst compile --features html --format html blog-post.typ
2 warning: html export is under active development and incomplete
```

… there's only the generic "unstable feature" warning, and the HTML looks reasonable too:

```html
1 <h2>My blog post</h2>
2 <p>Lorem ipsum dolor sit amet.</p>
3 <figure>
4   <img src="image.svg" alt="a rectangle">
5   <figcaption>Figure 1: A rectangle</figcaption>
6 </figure>
7 <p>Lorem ipsum dolor sit amet.</p>
8 <pre class="language-typ"><pre>#let x = 0</pre></pre>
9 <p><a href="#">(back to top)</a></p>
```

But this broke PDF export and the preview:

```
1 $ typst compile blog-post.typ
2 error: unknown variable: html
3    ┌─ blog-post.typ:19:19
4    │
5 19 │ #let back-to-top = html.elem("a", attrs: (href: "#"))[(back to top)]
6    │                    ^^^^
```

And even if we fixed *that*:

```
1 error: unknown variable: target
2   ┌─ blog-post.typ:1:25
3   │
4 1 │ #show: doc => context if target() != "html" {
5   │                          ^^^^^^
```

What now?

… enter Bullseye

II Simplifying target-specific Typst

When targeting both PDF and HTML, it's unavoidable to have some content that must be treated differently depending on the output format. This resulted in two fundamental pain points:

- Since Typst's HTML support and target switching in general are still unstable, the necessary functions to write this code are not always available. This problem will eventually go away.

- The necessary conditionals can lead to unwieldy code that is hard to read and write.

Bullseye tackles both of these problems. Let's look at some specific issues from the blog post example:

II.a Applying target-specific show rules

The original blog post contained the following rules that shouldn't be active for the HTML target:

```typ
#set page(height: auto)

#import "@preview/codly:1.3.0"
#show: codly.codly-init
```

Bullseye provides the `show-target()` function for this purpose, which you can use similar to a regular template function. Here is how the styles above would be extracted to a regular template:

```typ
#show: doc => {
  set page(height: auto)
  import "@preview/codly:1.3.0"
  show: codly.codly-init
  doc
}
```

And here is using `show-target()` for this:

```typ
#import "@preview/bullseye:0.1.0": *

#show: show-target(paged: doc => {
  set page(height: auto)

  import "@preview/codly:1.3.0"
  show: codly.codly-init
  doc
})
```

The template function is wrapped and passed as a named argument `paged`, meaning the show rule is only applied if the output format is PDF or one of the image formats. Multiple arguments can be specified, to apply different show rules for different formats.

This function can also be used to style specific elements, not just for document-wide settings. The following show rules from Section I.b were put into a block to only be applied for HTML output:

```typ
#show: doc => context if target() == "html" {
  show image: img => html.elem("img", attrs: ("src": img.source, "alt": img.alt))
  show raw.where(block: true): code => {
    html.elem("pre", attrs: ("class": "language-" + code.lang), code)
  }
  doc
}
```

Instead of moving both show rules into one shared conditional, `show-target()` makes it painless to individually apply them where you want to have them in your template:

```typ
#show image: show-target(html: img => {
  html.elem("img", attrs: ("src": img.source, "alt": img.alt))
})
```

```typ
#show raw.where(block: true): show-target(html: code => {
  html.elem("pre", attrs: ("class": "language-" + code.lang), code)
})
```

II.b Producing target-specific content

One of the "features" of the blog post was a "back to top" link produced in the HTML output. To conditionally produce this link, the document contained the following code:

```typ
#context if target() == "html" { back-to-top }
```

This isn't too complex, but Bullseye also has a utility function `on-target()` for producing a value only for some output formats, and `none` for others:

```typ
#context on-target(html: back-to-top)
```

This mirrors the structure for show rules. Like `show-target()`, `on-target()` can also accept multiple named arguments.

Both these functions are built on top of `match-target()`, which you can use if you have target-specific functionality that doesn't fit the show rule or extra content cases.

III Experimental feature placeholders

Typst's `html` module and the `target()` function for determining the kind of output are currently unstable, meaning they can't be used without a feature flag. In Section I.b, we saw how this leads to problems even when compiling to PDF, simply because the document is *prepared* for HTML output:

```
error: unknown variable: html
   ┌─ blog-post.typ:19:19
   │
19 │ #let back-to-top = html.elem("a", attrs: (href: "#"))[(back to top)]
   │                    ^^^^
```

```
error: unknown variable: target
  ┌─ blog-post.typ:1:25
  │
1 │ #show: doc => context if target() != "html" {
  │                          ^^^^^^
```

Some workarounds for this problem include

- Restructuring the code to only call experimental functions when HTML export is requested. This doesn't always make the code more manageable, and note that the `target()` function itself is among the unstable functions.
- Requiring the user to always enable HTML support:

  ```
  $ typst compile --features html blog-post.typ
  ```

  This is especially annoying when writing packages for people who may or may not be interested in HTML export. Also, this requires different approaches for plain CLI compilation (demonstrated above), Tinymist users, or web app users (not supported),

For this reason, Bullseye polyfills[1] the `target()` function when HTML support is not enabled, and contains a stub[2] `html` module that allows compiling code *creating* but not *rendering* HTML elements. These features were used in the previous examples, as they were included in this wildcard import:

```typ
#import "@preview/bullseye:0.1.0": *
```

Whether the placeholders or the real Typst code is executed depends on whether the HTML feature is enabled:

- HTML support is not enabled: Bullseye's placeholders are used. The `target()` function always returns `"paged"` (which is correct when HTML export isn't supported), and the `html.elem()` and `html.frame()` functions don't do anything useful. If you tried to unconditionally put an HTML element such as `back-to-top` into your document, it would panic.
- HTML support is enabled: Bullseye's exports simply forward to the standard ones. The `target()` function returns the same result as `std.target()`, and `html` is an exact alias to `std.html`. This is is independent from the export *target*, but it usually won't make a difference if not exporting to HTML.

There is one small difference between the stubbed and original `html` module: when not exporting to HTML, if a `std.html.elem()` appears in the document, it will result in a warning; Bullseye's `html.elem()` will panic instead!

[1] a polyfill is "code that implements a new standard feature of a deployment environment within an old version of that environment"

[2] a stub, in this case of a module instead of a method, is "a short and simple placeholder [that] contains just enough code to allow it to be used"

IV Module reference

IV.a bullseye


```
target() -> str
```

Returns "paged" or "html" depending on the current output target.

When HTML is supported, this is equivalent to Typst's built-in std.target(); otherwise, it always return "paged".

This is a polyfill for an unstable Typst function. It may not properly emulate the built-in function if it is changed before stabilization.

This function is contextual.

```
match-target(..targets: arguments) -> any
```

checks the target() (currently, "paged" and "html" are supported) and returns the associated value in the passed named arguments. If there is none and there is a default argument, that value is returned; otherwise there's a panic.

This function is contextual.

Examples:

```typc
match-target(html: "a", paged: "b")  // returns either "a" or "b"
match-target(html: "a", default: "b")  // returns either "a" or "b"
match-target(html: "a")  // returns either "a" or panics
```

Parameters:

..targets (arguments) – The possible options. Only named arguments with the keys paged, html and default are allowed.

```
show-target(..targets: arguments) -> function
```

Wrapper around match-target() for target-specific show rules. All values should be functions that you'd ordinarily use in a show rule, i.e. a single-parameter function that transforms some content. If no default is specified, it is set to it => it, i.e. non-covered targets remain unchanged.

This function is *not* contextual; it returns a function that provides its own context so that it can be used in show-everything rules (see examples below) that don't provide their own context.

Example:

```typ
#show: show-target(paged: strong, html: doc => ...)
```

```
// is equivalent to
#show: show-target(paged: strong)
#show: show-target(html: doc => ...)
// is equivalent to (pseudocode)
#show: strong if target() == "paged"
#show: doc => ... if target() == "html"
```

Parameters:

`..targets` (`arguments`) – The possible options. Only named arguments with the keys `paged`, `html` and `default` are allowed. The `default` key defaults to `it => it`.

`on-target(..targets: arguments) -> any`

Wrapper around `match-target()` for target-specific values that should default to `none`; particularly content, for which `none` simply has no effect. If no default is specified, it is set to `none`.

This function is contextual.

Example:

```typ
#on-target(paged: [foo])  // returns either [foo] or none

#(1, ..on-target(paged: (2, 3)), 4)  // returns either (1, 2, 3, 4) or (1, 4)
```

Parameters:

`..targets` (`arguments`) – The possible options. Only named arguments with the keys `paged`, `html` and `default` are allowed. The `default` key defaults to `none`.

`html: module`

The `html` module.

When HTML is supported, this is equivalent to Typst's built-in `std.html`; otherwise, it's the Bullseye `html` module documented below. That module doesn't *support* HTML, it just makes sure that calls to the html module that don't end up in a document don't prevent compilation.

This is a stub for an unstable Typst module. It may not properly emulate the built-in module (i.e. miss functions; no functionality beyond that is intended) if it is changed before stabilization.

IV.b `bullseye.html`

- `elem()`
- `frame()`

`elem(..args)`

A stub function for `std.html.elem()`. This function simply contextually panics, i.e. it can be called but its result must not appear in a document:

```typ
// assume `--features html` is not active
#let x = std.html.elem("div")  // panics, because `std.html` does not exist
#let y = bullseye.html.elem("div")  // ok
#y  // panics, because bullseye doesn't actually implement HTML elements
```

This is useful for writing code where alternative content for HTML is specified, but only rendered when HTML support is activated.

`frame(..args)`

A stub function for `std.html.frame()`. This function simply contextually panics, i.e. it can be called but its result must not appear in a document:

```typ
// assume `--features html` is not active
#let x = std.html.frame[body]  // panics, because `std.html` does not exist
#let y = bullseye.html.frame[body]  // ok
#y  // panics, because bullseye doesn't actually implement HTML elements
```

This is useful for writing code where alternative content for HTML is specified, but only rendered when HTML support is activated.