Word count (excluding code): 1219.

The following design specification outlines "a single, high-level Typst API that will result in usable forms for both PDF and HTML export." It follows the PDF 1.7 specification and the Living Standard for HTML.

A full example of the proposed API in a Typst source file is as follows:

```typ
#set text(font: "Inter")
#set form.text(stroke: 0.5pt + gray, radius: 2pt, inset: 4pt)
#show form.checkbox: set text(size: 0.9em)

#form[
  #form.group("applicant")[
    #form.fieldset(legend: [Applicant])[
      Name: #form.text(name: "name", width: 1fr, multiline: false) \
      #form.text(
        name: "email",
        kind: "email",
        label: [E-mail],
        placeholder: "hi@example.org",
        width: 20em,
      ) \
      #form.date(
        name: "dob",
        label: [Date of birth],
        max: datetime(year: 2008, month: 6, day: 10),
      )
    ]
  ]
]

#form.fieldset(legend: [Preferences])[
  #form.select(
    name: "country",
    label: [Country],
    options: (
      ("DE", "Germany"),
      ("FR", "France"),
      (value: "UK", label: "United Kingdom", disabled: true),
    )
  )
  #form.checkbox(name: "newsletter", value: true, label: [Subscribe])
  #form.radios(name: "contact-pref", value: "email")[
    #form.radio(option: "email") E-mail
    #form.radio(option: "phone") Phone
  ]
]
```

Listing 1: Interactive forms API example

Type granularity for form elements

The first major decision here was whether or how to integrate Typst content blocks (which can be recursively realized) with the various form elements, and which types to support for various attributes. When rendered in HTML, many of the attributes in form elements translate to 'string-ey' HTML attributes. For any content that has a user-facing display (such as `legend` in `#form.fieldset`

and `label` in form elements), I would try to allow a Typst Content element. For attributes that internally structure the form (such as `title`, `kind`, and `value`) that naturally take a string type, we use strings.

You will notice that the example above includes a few other types in certain attributes, such as datetime and boolean. In PDF, the various attributes of form elements often map to particular bit flags or values in field dictionaries, and thus sometimes require a stronger sense of type.[1]

In many cases, however, neither the HTML nor the PDF encoding strictly requires a strong sense of type. With an eye towards future-proofing (if the HTML or PDF spec is extended or changed, or Typst comes to support other targets as its roadmap suggests), and also to make the API feel more native in Typst, I have in these cases tended towards strongly typed attributes where it is appropriate.

The example of `date.max` in the example above is instructive towards this 'strongly typed' design, and also demonstrates a design decision with respect to values that pertain only to one target format. In PDF, there is no explicit 'date' field: dates are just entered as plain-text (and can only be validated through custom JavaScript in the Validate event). In HTML, by contrast, an input with type='date' can enforce both `min` and `max` values through attributes that accept a string in the "YYYY-MM-DD" format. Thus in PDF, the `#form.date` field would serialize to a field dictionary with its `V` (value) set to a rich text string, and we would 'lose' the extra type specificity (along with the `max` attribute) provided in the Typst source.

Note also the boolean `multiline` attribute in the `#form.text` field. In HTML, this attribute distinguishes between `<input type="text">` (when `false`) and `<textarea>` (when `true`) elements, whereas in PDF this sets the appropriate `Multiline` flag in the `Ff` value of the field dictionary.

In general, I would propose an API design that follows this principle, exposing attributes on fields that match the target format with the greater type specificity and/or richer set of attributes. The tradeoff here is that writing a `#form.date` as per the example above may compile to a form field in PDF that is very similar to what the user would get had they written a `#form.text` field.

While this perhaps diverges from some of the existing Typst APIs (which have historically treated PDF as the target format that orients the API surface), I believe that it is appropriate for the interactive form API on at least two counts:

1. The user can always choose to 'think in PDF' using elements such as `#form.text` should they not care about HTML.
2. Interactive forms are more common in HTML than they are in PDF, and so the interactive form API should support HTML as a first-class target.

The main downside to this design decision is that the interactive form API surface is larger, as it would support the range of different elements available in HTML.

Form grouping, naming, and namespacing

PDF partial names cannot contain periods, as per 12.7.3.2 (PDF 1.7), periods are used as a separator when constructing full names (hierarchical names with parents prefixed). Thus the first argument to the `#form.group` function, its `name`, throws an error when it contains periods. (An alternate option here would have been to automatically construct nested hierarchies when periods were included in the `name`. I rejected this on the basis that it could lead to unexpected behavior for users who unknowingly put periods in a group name.)

Note also that the various functions for form elements are all within the `#form` namespace. Though this arguably leads to a bit of repetition, it seemed more flexible than trying to handle all of the

[1] While there is in some sense stronger type validation of the user input in HTML forms (through the various values for the 'type' attribute on an input element, for example)—PDF forms, by contrast, seem to have very little in the way of type validation for user input—this does not matter so much to the *authoring* of the form structure itself, which is what we are concerned with in Typst.

possible permutations of a form in a DSL (as is done with cells in the table API). All form elements should be attached to a form parent, I believe, and this namespacing has the added benefit of reminding the Typst user that these functions are usually used within the context of a `#form[...]` block.

That said, the API design prototyped in Listing 1 could also be extended to allow form elements to occur outside of `#form[...]` blocks if the layout of form elements needs to occur within other Typst structures (so as to avoid the risk of users just wrapping the entire document in a `#form[...]` block). This might work like so:

typ
```
#table(
  columns: 4,
  [], [Agree], [Neutral], [Disagree],
  [The API feels native],
  form.radio(group: "q1", option: "agree"),
  form.radio(group: "q1", option: "neutral"),
  form.radio(group: "q1", option: "disagree"),
)
```

Listing 2: Possible extended form API example

Provided that there exists a `#form` declaration in the document that contains the `group` "q1", Listing 2 could still compile.

Styles, Fieldsets, and Radios

The interactive form API I am proposing would integrate with and allow `show` rules (i.e., line 3 in Listing 1), as both CSS styling in HTML as well as Rich Text Strings and Widget Annotations in the PDF spec allow various visual customizations of forms and form elements. Following APIs in the Typst model such as table, styles could also be applied by passing optional attributes to function invocations such as `#form.select`.

I have decided to include `#form.fieldset` as a way to group form elements partially so that styling can be targeted to particular groups, and also following the design philosophy outlined above of treating HTML as a first-class target.

I considered following the structure of `#table` for the `#form.radios` API, i.e. taking an arbitrary number of tuples that represent `(option, label)` like so:

typ
```
#form.radios(
  name: "contact-pref",
  value: "email",
  ("email", [E-mail]),
  ("phone", [Phone]),
)

```

Listing 3: Alternative syntax for `#form.radios`

This would not obviously enable extensions such as that in Listing 2, however: though cells in tables *always* fall grammatically within the bounds of the table, I am not certain that this is the case with radio options (especially in HTML).

Notes

The research for this proposal was conducted in ~3 hours (as reading the PDF specification is no joke!), and it took another ~2 hours to write this report from my notes. I used Claude Code in the research phase to interactively quiz myself on the divergences between the PDF and HTML specs.

All of the prose and code in both this report and cover-letter was typed by hand, and then spell-checked by Claude Code. Both PDFs were built (with love) using Typst and Rheo.