Prequery

v0.1.0 March 19, 2024

Clemens Koza

Abstract

Extracting metadata for preprocessing from a typst document, for example image URLs for download from the web.

Contents

I INTRODUCTION

Typst compilations are sandboxed: it is not possible for Typst packages, or even just a Typst document itself, to access the "ouside world". The only inputs that a Typst document can read are files within the compilation root, and strings given on the command line via `--input`. For example, if you want to embed an image from the internet in your document, you need to download the image using its URL, save the image in your Typst project, and then show that file using the `image()` function. Within your document, the image is not linked to its URL; that step was something *you* had to do, and have to do for every image you want to use from the internet.

This sandboxing of Typst has good reasons. Yet, it is often convenient to trade a bit of security for convenience by weakening it. Prequery helps with that by providing some simple scaffolding for supporting preprocessing of documents. The process is roughly like that:

1. You start authoring a document without all the external data ready, but specify in the document which data you will need. (With an image for example, you'd use Prequery's `image()` instead of the built-in one to specify not only the file path but also the URL.)
2. Using `typst query`, you extract a list of everything that's necessary from the document. (For images, the command is given in `image()`'s documentation.)
3. You run an external tool (a preprocessor) that is not subject to Typst's sandboxing to gather all the data into the expected places. (There is a *not very well implemented* Python script for image download in the gallery. For now, treat it as an example and not part of this package's feature set!)
4. Now that the external data is available, you compile the document.

I.a Fundamental issues and limitations

I.a.a Breaking the sandbox

As I said, there's a reason for Typst's sandboxing. Among those reasons are

- **Repeatability:** the content hidden behind URLs on the internet can change, so not having access to them ensures that compiling a document now will have the same result as compiling it later. The same goes for any other nondeterministic thing a preprocessor might do.
- **Security and trust:** when compiling a document, you know what data it can access, so you can fearlessly compile documents you did not write yourself. This is especially important as documents can import third-party packages. You don't need to trust all those packages to be able to trust a document itself.

The sandboxing is something that Typst ensures, but the preprocessors mentioned in step 3 above will necessarily *not* do the same. So using prequery (in the intended way, i.e. utilizing external preprocessing tools)

- **you need to trust the preprocessors that you run, because they are not (necessarily) sandboxed,** and
- **you need to trust the documents that you compile, including the packages they use, because the documents provide data to the preprocessors, possibly instructing them to do something that you don't want.**

This doesn't mean that using Prequery is necessarily dangerous; it just has more risks than Typst alone.

I.a.b Compatibility

The preprocessors you use will not necessarily work on all machines where Typst runs, including the wep app. This package assumes that you are using Typst via the command line.

II Usage

With that out of the way, here's an example of how to use Prequery:

```typ
1  #import "@preview/prequery:0.1.0"
2
3  // toggle this comment or pass `--input prequery-fallback=true` to enable fallback
4  // #prequery.fallback.update(true)
5
6  #prequery.image(
7      "https://upload.wikimedia.org/wikipedia/commons/a/af/Cc-public_domain_mark.svg",
8      "assets/public_domain.svg")
```

Instead of the built-in image(), we're using this package's image(). That function does the following things:

- it emits metadata to the document that can be queried for the use of preprocessors;
- it "normally" displays the image (note that this fails if the image has not been downloaded yet);
- in "fallback mode" (i.e. "not normally"), it doesn't try to load the image so that compilation succeeds;
- it is implemented on top of the prequery() function to achieve these easily.

We call a function of this sort "a prequery", and the image prequery is just a very common example. Other prequeries could, for example, instruct a preprocessor to capture the result of software that can't be run as a plugin.

As mentioned, this file will fail to compile unless activating fallback mode as described in the commented out part of the example. The next step is thus to actually get the referenced files, using query:

```sh
1  typst query main.typ '<web-resource>' --field value \
2      --input prequery-fallback=true
```

This will output the following piece of JSON:

```json
1  [{"url": "https://upload.wikimedia.org/wikipedia/commons/a/af/Cc-public_
   domain_mark.svg", "path": "assets/public_domain.svg"}]
```

… which can then be fed into a preprocessor. As mentioned, the gallery contains a Python script for processing this query output:

```py
1   import json, os.path, sys, urllib.request
2
3   data = json.load(sys.stdin)
4
5   if len(sys.argv) == 1: root = "."
6   elif len(sys.argv) == 2: root = sys.argv[1]
7   else: raise ValueError("at most one argument expected")
8
9   for item in data:
10      url = item['url']
11      path = os.path.join(root, item['path'])
12      if not os.path.isfile(path):
13          print(f"{path}: downloading {url}")
```

```
14          os.makedirs(os.path.dirname(path), exist_ok=True)
15          urllib.request.urlretrieve(url, path)
16      else:
17          print(f"{path}: skipping")
```

I repeat: I **don't** consider this script production ready! I have made the minimal effort of not downloading existing files multiple times, but files are only downloaded sequentially, and can be saved *anywhere* on the file system, not just where your Typst project can read them. This script is mainly for demonstration purposes. Handle with care!

Assuming Linux and a working Python installation, the query output can be directly fed into this script:

```sh
1  typst query main.typ '<web-resource>' --field value \
2      --input prequery-fallback=true | python3 download-web-resources.py
```

The first time this runs, the image will be downloaded with the following output:

```
1  assets/public_domain.svg: downloading https://upload.wikimedia.org/wikipedia/
   commons/a/af/Cc-public_domain_mark.svg
```

Success! After running this, compiling the document will succeed.

III AUTHORING A PREQUERY

This package is not just meant for people who want to download images; its real purpose is to make it easy to create *any* kind of preprocessing for Typst documents, without having to leave the document for configuring that preprocessing. While the package does not actually contain a lot of code, describing how the `image()` prequery is implemented might help – especially because it relies on a peculiar behavior regarding file path resolution. Here is the actual code:

```
71   #let _builtin_image = image
72
87   #let image(
90     url,
93     ..args,
94   ) = prequery(
95     (url: url, path: args.pos().at(0)),
96     <web-resource>,
101    _builtin_image.with(..args),
102    fallback: [\u{1F5BC}],
103  )
```

This function shadows a built-in one, which is of course not technically necessary. It does require us to keep an alias to the original function, though. The Parameters to the used `prequery()` function are as follows: the first two parameters specify the metadata made available for querying. The last one is also simple, it just specifies what to display if prequery is in fallback mode: the Unicode character "Frame with Picture" 🖼 .

The third parameter, written as `_builtin_image.with(..args)` is the most involved: first of all, this expression is a function that is only called if not in fallback mode. More importantly, `args` is an `arguments` value, and such a value apparently remembers where it was constructed. Compare these two functions (here, `image()` is just the regular, built-in function):

```typ
1  // xy/lib.typ                                                           typ
2  #let my-image(path, ..args) = image(path, ..args)
3  #let my-image2(..args) = image(..args)
```

While they seem to be equivalent (the `path` parameter of `image()` is mandatory anyway), they behave differently:

```typ
1  // main.typ                                                             typ
2  #import "xy/lib.typ": *
3  #my-image("assets/foo.png")   // tries to show "xy/assets/foo.png"
4  #my-image2("assets/foo.png")  // tries to show "assets/foo.png"
```

With `my-image`, passing `path` to `image()` resolves the path relative to the file `xy/lib.typ`, resulting in `"xy/assets/foo.png"`. With `my-image2` on the other hand, the path seems to be relative to where the `arguments` containing it were constructed, and that happens in `main.typ`, at the call site. The path is thus resolved as `"assets/foo.png"`.

This is of course very useful for prequeries, which are all about specifying the files into which external data should be saved, and then successfully reading from these files! As long as the file name remains in an `arguments` value, it can be passed on and still treated as relative to the caller of the package.

IV Module reference

IV.a `prequery`

- [prequery()](#)
- [image()](#)
- [fallback](#)

```
prequery(
  meta: any ,
  lbl: label ,
  body: content ,
  fallback: content ,
) -> content
```

This is the fundamental function for building prequeries. It adds metadata for preprocessing to the document and conditionally shows some fallback content when `fallback` mode is enabled.

The body may be given as a function, so that errors from the body don't let compilation fail when in fallback mode.

Parameters:

`meta` (`any`) – the metadata value to provide for preprocessing

`lbl` (`label`) – the label to give the created metadata

`body` (`content`) – function): the body to display; if a function is given, that function will not be called in fallback mode

`fallback` (`content` = `none`) – the fallback content to display when in fallback mode

```
image(url: string , ..args: arguments ) -> content
```

A prequery for images. Apart from the `url` parameter, the image file name is also mandatory; it is part of `args` for technical reasons. Outside fallback mode, rendering this fails when the referenced image is missing; images need to be downloaded in a preprocessing step.

This function provides a dictionary with `url` and `path` as metadata under the label `<web-resource>`. This metadata can be queried like this:

```sh
1 typst query --input prequery-fallback=true --field value ... '<web-resource>'
```

Fallback: renders the Unicode character "Frame with Picture" (U+1F5BC).

Parameters:

`url` (`string`) – the URL of the image to be shown

`..args` (`arguments`) – arguments to be forwarded to built-in `image`

```
fallback: state
```

A boolean state indicating whether the document should display fallback content for elements requiring results from preprocessing. For example, when using prequery images, compiling the document before download the files during preprocessing will fail. If all prequeries used in the document work even before preprocessing, it is not necessary to turn fallback mode on.

When editing the document and adding images (or other preprocessed content), it is convenient to temporarily add a line at the top to switch fallback mode on:

```typ
1  #fallback.update(true)
```

When querying the document for preprocessing, this can be activated using `--input`:

```sh
1  typst query --input prequery-fallback=true ...
```