Search (S)

Overview

Tutorial

- Writing in Typst
- Formatting
- Advanced Styling
- Making a Template

Reference

Guides

Changelog

Roadmap

Community

Tutorial > Making a Template

Making a Template

In the previous three chapters of this tutorial, you have learned how to write a document in Typst, apply basic styles, and customize its appearance in-depth to comply with a publisher's style guide. Because the paper you wrote in the previous chapter was a tremendous success, you have been asked to write a follow-up article for the same conference. This time, you want to take the style you created in the previous chapter and turn it into a reusable template. In this chapter you will learn how to create a template that you and your team can use with just one show rule. Let's get started!

A toy template

In Typst, templates are functions in which you can wrap your whole document. To learn how to do that, let's first review how to write your very own functions. They can do anything you want them to, so why not go a bit crazy?

```
#let amazed(term) = box[✨ #term ✨]

You are #amazed[beautiful]!
```

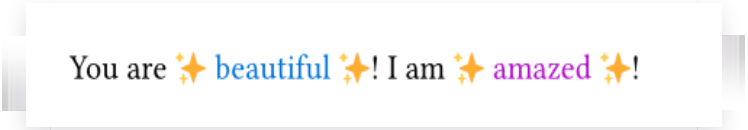
You are ✨ beautiful ✨!

This function takes a single argument, `term`, and

returns a content block with the `term` surrounded by sparkles. We also put the whole thing in a box so that the term we are amazed by cannot be separated from its sparkles by a line break.

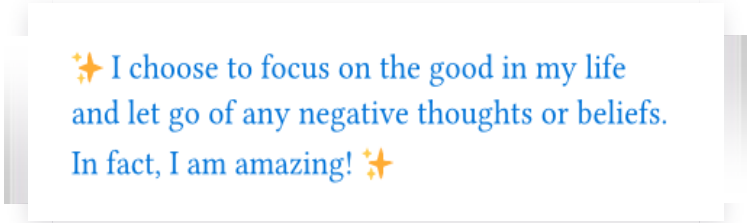
Many functions that come with Typst have optional named parameters. Our functions can also have them. Let's add a parameter to our function that lets us choose the color of the text. We need to provide a default color in case the parameter isn't given.

```
#let amazed(term, color: blue) = {
  text(color, box[✨ #term ✨])
}

You are #amazed[beautiful]!
I am #amazed(color: purple)[amazed]!
```



Templates now work by wrapping our whole document in a custom function like `amazed`. But wrapping a whole document in a giant function call would be cumbersome! Instead, we can use an "everything" show rule to achieve the same with cleaner code. To write such a show rule, put a colon directly behind the show keyword and then provide a function. This function is given the rest of the document as a parameter. The function can then do anything with this content. Since the `amazed` function can be called with a single content argument, we can just pass it by name to the show rule. Let's try it:

```
#show: amazed
I choose to focus on the good
in my life and let go of any
negative thoughts or beliefs.
In fact, I am amazing!
```

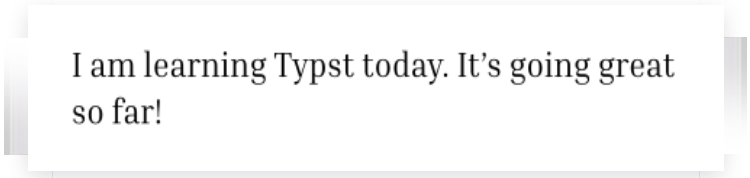


Our whole document will now be passed to the `amazed` function, as if we wrapped it around it. Of course, this is not especially useful with this particular function, but when combined with set rules and named arguments, it can be very powerful.

Embedding set and show rules

To apply some set and show rules to our template, we can use `set` and `show` within a content block in our function and then insert the document into that content block.

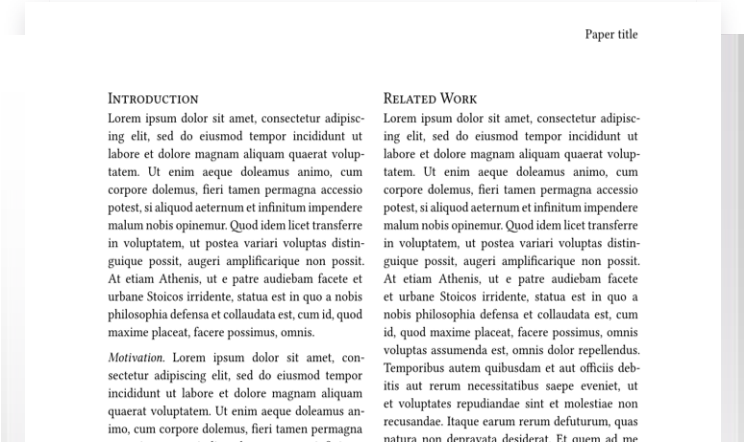
```
#let template(doc) = [
  #set text(font: "Inria Serif")
  #show "something cool": [Typst]
  #doc
]

#show: template
I am learning something cool today.
It's going great so far!
```



Just like we already discovered in the previous chapter, set rules will apply to everything within their content block. Since the everything show rule passes

our whole document to the `template` function, the text set rule and string show rule in our template will apply to the whole document. Let's use this knowledge to create a template that reproduces the body style of the paper we wrote in the previous chapter.

```
#let conf(title, doc) = {
  set page(
    paper: "us-letter",
    header: align(
      right + horizon,
      title
    ),
    columns: 2,
    ...
  )
  set par(justify: true)
  set text(
    font: "Libertinus Serif",
    size: 11pt,
  )

  // Heading show rules.
  ...

  doc
}

#show: doc => conf(
  [Paper title],
  doc,
)

= Introduction
#lorem(90)

...
```



We copy-pasted most of that code from the previous chapter. The two differences are this:

1. We wrapped everything in the function `conf` using an everything show rule. The function applies a few set and show rules and echoes the content it has been passed at the end.
2. Moreover, we used a curly-braced code block instead of a content block. This way, we don't need to prefix all set rules and function calls with a #. In exchange, we cannot write markup directly in the code block anymore.

Also note where the title comes from: We previously had it inside of a variable. Now, we are receiving it as the first parameter of the template function. To do so, we passed a closure (that's a function without a name that is used right away) to the everything show rule. We did that because the `conf` function expects two positional arguments, the title and the body, but the show rule will only pass the body. Therefore, we add a new function definition that allows us to set a paper title and use the single parameter from the show rule.

Templates with named arguments

Our paper in the previous chapter had a title and an author list. Let's add these things to our template. In addition to the title, we want our template to accept a list of authors with their affiliations and the paper's

abstract. To keep things readable, we'll add those as named arguments. In the end, we want it to work like this:

```
#show: doc => conf(
  title: [Towards Improved Modelling],
  authors: (
    (
      name: "Theresa Tungsten",
      affiliation: "Artos Institute",
      email: "tung@artos.edu",
    ),
    (
      name: "Eugene Deklan",
      affiliation: "Honduras State",
      email: "e.deklan@hstate.hn",
    ),
  ),
  abstract: lorem(80),
  doc,
)

...
```

Let's build this new template function. First, we add a default value to the `title` argument. This way, we can call the template without specifying a title. We also add the named `authors` and `abstract` parameters with empty defaults. Next, we copy the code that generates title, abstract and authors from the previous chapter into the template, replacing the fixed details with the parameters.

The new `authors` parameter expects an array of dictionaries with the keys `name`, `affiliation` and `email`. Because we can have an arbitrary number of authors, we dynamically determine if we need one, two or three columns for the author list. First, we determine the number of authors using the `.len()` method on the `authors` array. Then, we set the number of columns as the minimum of this count and three, so that we never create more than three columns. If there are more than three authors, a new row will be inserted instead. For this purpose, we have also added a `row-gutter` parameter to the `grid` function. Otherwise, the rows would be too close together. To extract the details about the authors from the dictionary, we use the field access

syntax.

We still have to provide an argument to the grid for each author: Here is where the array's map method comes in handy. It takes a function as an argument that gets called with each item of the array. We pass it a function that formats the details for each author and returns a new array containing content values. We've now got one array of values that we'd like to use as multiple arguments for the grid. We can do that by using the spread operator. It takes an array and applies each of its items as a separate argument to the function.

The resulting template function looks like this:

```
#let conf(
  title: none,
  authors: (),
  abstract: [],
  doc,
) = {
  // Set and show rules from before.
  ...

  set align(center)
  text(17pt, title)

  let count = authors.len()
  let ncols = calc.min(count, 3)
  grid(
    columns: (1fr,) * ncols,
    row-gutter: 24pt,
    ..authors.map(author => [
      #author.name \
      #author.affiliation \
      #link("mailto:" + author.email)
    ]),
  )

  par(justify: false)[
    *Abstract* \
    #abstract
  ]

  set align(left)
  doc
}
```

A separate file

Most of the time, a template is specified in a different file and then imported into the document. This way, the main file you write in is kept clutter free and your template is easily reused. Create a new text file in the file panel by clicking the plus button and name it `conf.typ`. Move the `conf` function definition inside of that new file. Now you can access it from your main file by adding an import before the show rule. Specify the path of the file between the `import` keyword and a colon, then name the function that you want to import.

Another thing that you can do to make applying templates just a bit more elegant is to use the `.with` method on functions to pre-populate all the named arguments. This way, you can avoid spelling out a closure and appending the content argument at the bottom of your template list. Templates on Typst Universe are designed to work with this style of function call.

```
#import "conf.typ": conf
#show: conf.with(
  title: [
    Towards Improved Modelling
  ],
  authors: (
    (
      name: "Theresa Tungsten",
      affiliation: "Artos Institute",
      email: "tung@artos.edu",
    ),
    (
      name: "Eugene Deklan",
      affiliation: "Honduras State",
      email: "e.deklan@hstate.hn",
    ),
  ),
  abstract: lorem(80),
)

= Introduction
#lorem(90)

== Motivation
#lorem(140)

== Problem Statement
#lorem(50)
```

```
= Related Work
#lorem(200)
```

We have now converted the conference paper into a reusable template for that conference! Why not share it in the Forum or on Typst's Discord server so that others can use it too?

Review

Congratulations, you have completed Typst's Tutorial! In this section, you have learned how to define your own functions and how to create and apply templates that define reusable document styles. You've made it far and learned a lot. You can now use Typst to write your own documents and share them with others.

We are still a super young project and are looking for feedback. If you have any questions, suggestions or you found a bug, please let us know in the Forum, on

our Discord server, on GitHub, or via the web app's feedback form (always available in the Help menu).

So what are you waiting for? Sign up and write something!

Advanced Styling
Previous page

Reference
Next page