API & Style Guidelines

Guidelines for Package & Template Development

v0.1.0 2024-05-01 MIT

tinger <ME@TINGER.DEV>

https://github.com/typst-community/guidelines

This document contains stylistic guidelines and API best practices and patterns to help package and template developers create simple to use packages with extensible and interoperable APIs. The stylistic guidelines apply to documents in general and make it easier for others to read and contribute to code.

Table of contents

Part I.
Manifest

Part II.
Style

II.1.1. Naming

Identifiers use `kebab-case`, i.e. lowercase words separated by hyphens. Identifiers should be concise, but clear. Avoid long name with many hyphens for public APIs. Abbreviations and uncommon names should be avoided or at least clarified when use din public APIs.

Do

```
let name
let short-name
```

Don't

```
let shortname // missing hyphen
let avn // uncommon abbreviation
let unnecessarily-long-name // longer than needed
```

II.1.2. Indentation

Indentation is always 2 spaces per level, tabs can be used to increase accessibility, but spaces and tabs should not be mixed. For 2 spaces this means that for most syntactic elements, the indentation of nested items or continuations lines up with the start of the first line. This keeps nested code narrow, allowing side-by-side editing and preview for most users.

Do

```
// consistent easy to follow indentation
- top level
  - nested
    continuation
```

Don't

```
// inconsistent and too deep indentation
- top level
    - nested
      continued
```

Likewise, in `{code-mode}` the indentation of scopes is done with 2 spaces.

Do

```
// 2 spaces are generally enough to visually tell levels of indentation apart
let id(val) = {
  if false {
    panic()
  }
  val
}
```

Don't

```
// ridiculous indentation levels make this quickly hard to read when previewing
a document
let func(val) = {
        if val {
                // ...
        }
}
```

Avoid deep indentation where possible, if a function has multiple code paths that end in simple return values and another which is heavily nested and large, try to invert conditions to decrease nesting.

Do

```
let nested(a, b, c) = {
  if not a {
    return 0
  }
  if not b {
    return 1
  }
  // lots of code ...
}
```

Don't

```
 let nested(a, b, c) = {
   if a {
     if b {
       // lots of code ...
     } else {
       1
     }
   } else {
     0
   }
 }
```

II.1.3. Delimiters

Opening delimiters to unfinished statements like `(...)`, `[...]`, etc. are placed on the same line as their preceding element, e.g. for loops, declarations, etc. This is actually enforced by the compiler in most cases, as it uses line breaks to terminate most statements.

Do

```
// if it fits on one line avoid linebreaks
#let var = if { ... }
// if there's a little more we break K&R style
#let var = for x in xs [
  ...
]
// likewise, else must be on the same line as the if's closing brace
#if var {
  // ...
} else {
  // ...
}
```

Don't

```
// this doesn't parse
#let var =
{
}
// ask yourself not if you could, but if you should
#(
  if true
  {
  }
  else
  {
  }
)
```

II.1.4. Mode Switching

Prefer staying in the primary mode of your current context, this avoids unnecessarily frequent use of # and unintended leading and trailing whitespace in `[markup-mode]`.

Do

```
// we switched into code mode once and stay in it
#figure(caption: [...],
  stack(dir: ltr,
    table[...],
    table[...],
  )
)
```

Don't

```
// we switch back and forth, making writing and reading harder
#figure(caption: [...])[
  #stack(dir: ltr)[
    #table[...]
  ][
    #table[...]
  ]
]
```

Use the most appropriate mode for your top level scopes, if a function has a lot of statements that don't evaluate to content, `{code-mode}` is likely a better choice than `[markup-mode]`.

Do

```
// we have a high ratio of text to code
let filler = [
  // lots of text
]

// we have a high ratio of code to text
let computed = if val {
  let x
  let y
  let z
  // ...
}
```

Don't

```
// needless content mode
let func(a, b, c) = [
  #let as = calc.pow(a, 2)
  #let bs = calc.pow(b, 2)
  #let cs = calc.pow(c, 2)
  #return calc.sqrt(as + bs + cs)
]
```

II.1.5. Joining

Prefer statement joining over manual joining, this keeps code and markup similar in structure to the end document.

Do

```
// in a document this clearly communicates intent
#for x in ("a", "b", "c") [
  - #x
]
```

Don't

```
// this is harder to read and edit
#let res
#for x in ("a", "b", "c") {
  res += [- #x]
}
#res
```

Even in `{code-mode}` this can be applied to complex control flows which evaluate to content.

Do

```
// we can read the control flow and it's effect on the document top to bottom
let func(a, b, c) = {
  [prefix: ]
  a
  if b {
    c
  }
}
```

Don't

```
// even with a low amount of nesting reading this is harder
let func(a, b, c) = [prefix: ] + a + if b {
  c
}
```

II.1.6. Syntax Sugar

Prefer syntax sugar only for simple markup related tasks.

TODO

II.1.7. Trailing Content Arguments

Prefer trailing content argument calling style `#func(...)[...]` only for short runs of arguments directly. Trailing content arguments are harder to visually separate for complex or large amounts of arguments such as tables with multiple rows and make refactoring more noisy in diffs.

Do

```
#link("https://github.com")[github.com]
#custom[a][b]
```

Don't

```
// most tables
#table[a][b][c]
```

II.1.8. Linebreaks

For documents with a lot of collaboration aided by version control such as git, consider using something similar to Semantic Line Breaks. Avoid hard-wrapping by the editor, as it makes diffs in `[markup-mode]` harder to read. Typst will maintain the correct linebreaks in both cases.

Do

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit,
sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
Nunc non blandit massa enim nec.
Eu volutpat odio facilisis mauris.
Velit euismod in pellentesque massa placerat duis ultricies lacus.
```

Don't

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Nunc non blandit massa enim nec.
Eu volutpat odio facilisis mauris. Velit euismod in pellentesque massa placerat
duis ultricies lacus.
```

Semantic Line Breaking cannot be used for chinese markup at the moment.

II.1.9. Documentation

This section documents the syntax of documentation commens, see Section III.1.3 for the contents and purpose of documentation comments.

Documentation is placed on special comments using three forward slashes `///` and an optional space. These are called doc comments. While the leading space is optional, it is encouraged as it makes the documentaiton easier to read. Doc comments may not be interrupted by empty lines, markup, or regular comments.

Do

```
/// A1
/// A2
/// A3

// discuraged, but valid
///B1
///B2
///B3
```

Don't

```
  /// A1
  //
  /// B1

  /// A1

  /// B1

  /// C1
  Hello World
  /// D1

  /**
   * This is a regular comment, not a doc comment.
   */
  #let item = { ... }

  // This is a regular comment, not a doc comment.
  #let item = { ... }
```

There are two kinds of documentation comments, inner and outer doc comments. Outer doc comments are placed right above the declaration they are attached to.

Do

```
/// Documentation for func
#let func() = { ... }
```

Don't

```
/// Stray doc comment, not the doc for func

#let func() = {
  /// Stray doc comment, not the doc for func
  ...
}
/// Stray doc comment, not the doc for func
```

Inner doc comments are used to document modules and must not have a declaration, instead they refer to the file they are placed in and may only be declared once and as the first non comment item in a file.

Do

```
// optional leading comments

/// Module doc

/// Function or value doc
#let item = { ... }
```

Don't

```
/// Function or valtion doc
#let item = { ... }

/// Stray doc comment, not the module or func doc
```

Outer doc comments may be used on `let` bindings only.

Do

```
/// Function doc
#let func() = { ... }

/// Value doc
#let value = { ... }
```

Don't

```
/// Stray doc comment, markup may not be documented
Hello World

/// Stray doc comment, imports my not be documented
#import "module.typ"

/// Stray doc comment, scopes may not be documented
#[
  ...
]
```

Doc comments contain a description of the documented item itself, as well as an optional semantic trailer. The content of descriptions should generally be simple Typst markup and should not contain any scripting, (i.e. no loops, conditionals or function calls, except for `#link("...")[...]`), this allows the documentation to be turned into Markdown or plaintext or LSP to send to editors.

II.1.9.1. Description

As mentioned before, the description should be simple, containing mostly markup and no scripting.

II.1.9.2. Semantic Trailer

The semantic trailer is fully optional, starts with an empty doc comment line to separate it from the description, and may contain:
- multiple `term` items for parameter type hints and descriptions,
- a return type hint `-> type`,
- and multiple property annotations (`#property("private")` or `#property("deprecated")`).

Types in type lists or return type annotations may be separated by `|` to indicate that more than one type is accepted, the exact types allowed depend on the doc parser, but the built in types are generally supported.

Parameter description and return types (if present) are placed tightly together, property annotations if present are separated using another empty doc comment line.

Parameter documentation is created by writing a term item containing the parameter name, a mandatory type list in parenthesis and optional description. If the parameter is an argument sink it's name must also containe the spread operator `..`. Each parameter can only be documented once, but doesn't have to, undocumented parameters are considered private.

Do

```
/// Function description
///
/// / b (any): b descpription
/// / a (int | float): a description
#let func(a, b, c) = { ... }

/// Function description
///
/// / ..args (int): args description
#let func(..args) = { ... }
```

Don't

```
/// Function description
///
/// / c (any): c doesn't exist
/// / a (int | float): a description
#let func(a, b) = { ... }

/// Missing empty line between function description and trailer
/// / b (any): b descpription
/// / a (int | float): a description
#let func(a, b) = { ... }

/// Function description
///
/// / b (any): b descpription
/// / a: missing types
#let func(a, b) = { ... }

/// Function description
///
/// / args (int): missing spread operator for args
#let func(..args) = { ... }
```

The return type can only be annotated once, on a single line after all parameters if any exist. For non function types the return type annotation can be used as a normal type annotation.

Do

```
/// Function doc
///
/// / arg (types): Description for arg
/// -> types
#let func(arg) = { ... }

/// Value doc
///
/// -> type
#let value = { ... }
```

Don't

```
/// Missing empty line between description trailer
///
/// -> type
/// / arg (types): arg descrption after return type
#let func(arg) = { ... }
```

Property annotations can be used to document package specific or otherwise important information like deprecation status, visibility or contextuality and may only be used after the return type (if one exists) and an empty doc comment line.

Do

```
/// Function doc
///
/// / arg (types): Description for arg
/// -> types
///
/// #property("deprecated")
/// #property("contextual")
#let func(arg) = { ... }

/// Value doc
///
/// #property("contextual")
#let value = { ... }
```

Don't

```
/// Missing empty line between return type and properties
///
/// / arg (types): Description for arg
/// -> types
/// #property("deprecated")
/// #property("contextual")
#let func(arg) = { ... }

/// Return type after properties and missing empty separation lines
///
/// / arg (types): Description for arg
/// #property("deprecated")
/// #property("contextual")
/// -> types
#let func(arg) = { ... }
```

Part III.
API

III.1.1. Flexibility

Good APIs must be flexible enough to allow users to use them in situations a developer might not have anticipated directly.

III.1.1.1. requiring or providing `context`

Starting with Typst 0.11, the context feature allows certain functions to access information about the current location inside the document. The documentation gives the following example:

```
#let value = context text.lang
#value
--
#set text(lang: "de"); #value
--
#set text(lang: "fr"); #value
```

en – de – fr

The same `value` is rendered three times, but with different results. This is of course a powerful tool for library authors! However, there is an important restriction that `context` needs to impose to be able to do that: `context` values are opaque content; the above does not result in a string such as `"en"`, it just renders that way:

```
#let value = context text.lang
Rendered: #value

Type/representation: #type(value)/#raw(repr(value))
```

Rendered: en

Type/representation: `content/context()`

This means that returning a context expression limits what can be done with a function's result. As a rule of thumb, if it's useful for a user to do more with your function than just render its result, you likely want to require the user of your function to use context instead of providing it yourself:

```
Do
```

```
/// Returns the current language. This function requires context.
///
/// -> string
#let fancy-get-language() = { text.lang }
#context fancy-get-language()
// Ok: the length of the language code should be 2
#context fancy-get-language().len()
```

```
Don't
```

```
/// Returns the current language as opaque content.
///
/// -> content
#let fancy-get-language() = context { text.lang }
#fancy-get-language()
// Doesn't work: type content has no method `len`
//   #fancy-get-language().len()
```

The first variant of the `fancy-get-language` function allows the caller to do something with the returned language code (which, with this simplistic function is *necessary* to domething useful); the latter one can only be rendered.

There are of course exceptions to the rule: requiring using `context` is *a bit* more complicated to call for users, so if there is no benefit (e.g. the function returns complex content where inspecting it doesn't make sense anyway) it may be more useful to just return opaque content so that the user does not need to think about context.

III.1.2. Simplicity

Good APIs should be simple, easy to use and aware of their primary use case. Flexibility as discussed in Section III.1.1 should not interfere with the usability of an API.

III.1.2.1. Higher Order Templates

A common pattern for Typst templates is to provide a single top level function, which a user imports and applys with a show-all rule, i.e. `show: func`. This makes for a simple API in the case of no arguments, but requires extra thought when arguments have to be applied. This idiom also comes with additional boilerplate in almost any template, templates are almost never appled without any arguments. By using functions as values, a template author can provide a simpler API for the most common use case of applying this function as a template.

Consider returning a function which is applied in the show-all rule from your template function, instead of taking the content itself.

Do

```
#let doc(..args) = body => {
  // .. use args
  body
}
```

Don't

```
#let doc(..args, body) = {
  // .. use args
  body
}
```

What was previously this:

```
#import "@preview:template:0.1.0": doc
#show: doc.with(..args)
```

Can now be used like this:

```
#import "@preview:template:0.1.0": doc
#show: doc(..args)
```

This also comes with the benefit of being future proof, as current discussions on the Typst Community Discord indicate that templates will likely be sets of rules instead of functions in the future. A template using this idiom can then simply return the styles instead of a function, requiring no other change from a downstream user.

III.1.3. Documentation

Documentaiton comemnts are important for consumers of the API as well as new contributors to your package or project to understand the usage and purpose of an item. An item may be a state variable, a counter, a function, a reusable piece of regular content, or anything that can be bound using a `let` statement.

> For the exact syntax used for documentation comments, refer to Section II.1.9.

For public functions the documentation should always incldue all public parameters, their usage and purpose as well as possible constraints on their values. Internal arguments may be ommited, if they are not meant to be exposed to a user, but should still be documented for contributors in the source or a contribution document. Depending on the doc aprser in use, named arguments may need their defaults explicitly stated and should have if the doc parser can't parse them automatically. If functions require or return contextual values, this should also be communicated clearly. Functions should also document their return type, even if it is general, such as `any`. Should a function return different types depending on it's inputs or the state of the document, it must be documented when this may happen or at least which types to expect and check for at the call site.

Values like states and counters, if exposed, should document their invariants, such as the allowed types for states or the expected depth range of a counter for example. The type of the value itself should likewise be annotatd using the function return type syntax.

Do

```
// - well documented invariants
// - most doc parsers will identify named defaults, we omit them for brevity
//   in this example

/// Does things.
///
/// This function is contextual if `arg` == `other`.
///
/// - arg (int): Fancy arg, must be less than or equal to `other`.
/// - other (int): Other fancy arg, must be larger than `arg`.
/// -> content
#let func(arg, other: 1) = {
  assert(arg <= other)

  if arg == other {
    context { ... }
  } else {
    ...
  }
}
```

Don't

```
// user don't know how these definitions must be shaped, optional fields on
// dictionaries especially can make it hard to get a full graps of those implicit
// invariants if they can only inspect the state var in their doc to reverse
// engineer the definition shape

/// Contains the function definitions.
#let item = state("defs", (:))

// - invariants of `arg` are not documented and may result in poor UX
// - other is not documented
// - does not document contexuality

/// Does things.
///
/// - arg (any): Fancy arg.
#let func(arg, other: 1) = {
  // the lack of typing information would cause a hard to understand error
  // message for a novice user
  assert(arg <= other)

  if arg == other {
    context { ... }
  } else {
    ...
  }
}
```

Part IV.
Index