Stencil Documentation

Release 0.7.0

Kyle Fuller

March 02, 2017

Contents

Stencil is a simple and powerful template language for Swift. It provides a syntax similar to Django and Mustache. If you're familiar with these, you will feel right at home with Stencil.

```
There are {{ articles.count }} articles.

<ul>
  {% for article in articles %}
    <li>{{ article.title }} by {{ article.author }}</li>
  {% endfor %}
</ul>
```

```
import Stencil

struct Article {
  let title: String
  let author: String
}

let context = [
  "articles": [
    Article(title: "Migrating from OCUnit to XCTest", author: "Kyle Fuller"),
    Article(title: "Memory Management with ARC", author: "Kyle Fuller"),
  ]
]

let environment = Environment(loader: FileSystemLoader(paths: ["templates/"])
let rendered = try environment.renderTemplate(name: context)

print(rendered)
```

CHAPTER 1

The User Guide

For Template Writers

Resources for Stencil template authors to write Stencil templates.

Language overview

- `{{ ... }}` for variables to print to the template output
- `{% ... %}` for tags
- `{# ... #}` for comments not included in the template output

Variables

A variable can be defined in your template using the following:

```
{{ variable }}
```

Stencil will look up the variable inside the current variable context and evaluate it. When a variable contains a dot, it will try doing the following lookup:

- Context lookup
- Dictionary lookup
- Array lookup (first, last, count, index)
- Key value coding lookup
- Type introspection

For example, if *people* was an array:

```
There are {{ people.count }} people. {{ people.first }} is the first
person, followed by {{ people.1 }}.
```

Filters

Filters allow you to transform the values of variables. For example, they look like:

```
{{ variable|uppercase }}
```

See *all builtin filters*.

Tags

Tags are a mechanism to execute a piece of code, allowing you to have control flow within your template.

```
{% if variable %}
  {{ variable }} was found.
{% endif %}
```

A tag can also affect the context and define variables as follows:

```
{% for item in items %}
  {{ item }}
{% endfor %}
```

Stencil includes of built-in tags which are listed below. You can also extend Stencil by providing your own tags.

See *all builtin tags*.

Comments

To comment out part of your template, you can use the following syntax:

```
{# My comment is completely hidden #}
```

Template inheritance

Template inheritance allows the common components surrounding individual pages to be shared across other templates. You can define blocks which can be overidden in any child template.

Let's take a look at an example. Here is our base template (`base.html`):

```
<html>
  <head>
    <title>{% block title %}Example{% endblock %}</title>
  </head>

  <body>
    <aside>
      {% block sidebar %}
        <ul>
          <li><a href="/">Home</a></li>
          <li><a href="/notes/">Notes</a></li>
        </ul>
      {% endblock %}
    </aside>

    <section>
      {% block content %}{% endblock %}
    </section>
  </body>
</html>
```

This example declares three blocks, `title`, `sidebar` and `content`. We can use the `{% extends %}` template tag to inherit from out base template and then use `{% block %}` to override any blocks from our base template.

A child template might look like the following:

```
{% extends "base.html" %}

{% block title %}Notes{% endblock %}

{% block content %}
  {% for note in notes %}
    <h2>{{ note }}</h2>
  {% endfor %}
{% endblock %}
```

Note: You can use "{{ block.super }}' inside a block to render the contents of the parent block inline.

Since our child template doesn't declare a sidebar block. The original sidebar from our base template will be used. Depending on the content of `notes` our template might be rendered like the following:

```
<html>
  <head>
    <title>Notes</title>
  </head>

  <body>
    <aside>
      <ul>
        <li><a href="/">Home</a></li>
        <li><a href="/notes/">Notes</a></li>
      </ul>
    </aside>

    <section>
      <h2>Pick up food</h2>
      <h2>Do laundry</h2>
    </section>
  </body>
</html>
```

You can use as many levels of inheritance as needed. One common way of using inheritance is the following three-level approach:

- Create a `base.html` template that holds the main look-and-feel of your site.
- Create a `base_SECTIONNAME.html` template for each "section" of your site. For example, `base_news.html`, `base_news.html`. These templates all extend `base.html` and include section-specific styles/design.
- Create individual templates for each type of page, such as a news article or blog entry. These templates extend the appropriate section template.

Built-in template tags and filters

Built-in Tags

for

A for loop allows you to iterate over an array found by variable lookup.

```
<ul>
  {% for user in users %}
    <li>{{ user }}</li>
  {% endfor %}
</ul>
```

The `for` tag can take an optional `{% empty %}` block that will be displayed if the given list is empty or could not be found.

```
<ul>
  {% for user in users %}
    <li>{{ user }}</li>
  {% empty %}
    <li>There are no users.</li>
  {% endfor %}
</ul>
```

The for block sets a few variables available within the loop:

- `first` - True if this is the first time through the loop
- `last` - True if this is the last time through the loop
- `counter` - The current iteration of the loop

if

The `{% if %}` tag evaluates a variable, and if that variable evaluates to true the contents of the block are processed. Being true is defined as:

- Present in the context
- Being non-empty (dictionaries or arrays)
- Not being a false boolean value
- Not being a numerical value of 0 or below
- Not being an empty string

```
{% if variable %}
  The variable was found in the current context.
{% else %}
  The variable was not found.
{% endif %}
```

Operators `if` tags may combine `and`, `or` and `not` to test multiple variables or to negate a variable.

```
{% if one and two %}
    Both one and two evaluate to true.
{% endif %}

{% if not one %}
    One evaluates to false
{% endif %}
```

```
{% if one or two %}
    Either one or two evaluates to true.
{% endif %}

{% if not one or two %}
    One does not evaluate to false or two evaluates to true.
{% endif %}
```

You may use `and`, `or` and `not` multiple times together. `not` has higest prescidence followed by `and`. For example:

```
{% if one or two and three %}
```

Will be treated as:

```
one or (two and three)
```

== operator

```
{% if value == other_value %}
  value is equal to other_value
{% endif %}
```

Note: The equality operator only supports numerical, string and boolean types.

!= operator

```
{% if value != other_value %}
  value is not equal to other_value
{% endif %}
```

Note: The inequality operator only supports numerical, string and boolean types.

< operator

```
{% if value < other_value %}
  value is less than other_value
{% endif %}
```

Note: The less than operator only supports numerical types.

<= operator

```
{% if value <= other_value %}
  value is less than or equal to other_value
{% endif %}
```

Note: The less than equal operator only supports numerical types.

> operator

```
{% if value > other_value %}
  value is more than other_value
{% endif %}
```

Note: The more than operator only supports numerical types.

>= operator

```
{% if value >= other_value %}
  value is more than or equal to other_value
{% endif %}
```

Note: The more than equal operator only supports numerical types.

ifnot

Note: `{% ifnot %}` is deprecated. You should use `{% if not %}`.

```
{% ifnot variable %}
  The variable was NOT found in the current context.
{% else %}
  The variable was found.
{% endif %}
```

now

filter

Filters the contents of the block.

```
{% filter lowercase %}
  This Text Will Be Lowercased.
{% endfilter %}
```

You can chain multiple filters with a pipe (|).

```
{% filter lowercase|capitalize %}
  This Text Will First Be Lowercased, Then The First Character Will BE
  Capitalised.
{% endfilter %}
```

include

You can include another template using the *include* tag.

```
{% include "comment.html" %}
```

The *include* tag requires you to provide a loader which will be used to lookup the template.

```
let environment = Environment(bundle: [Bundle.main])
let template = environment.loadTemplate(name: "index.html")
```

extends

Extends the template from a parent template.

```
{% extends "base.html" %}
```

See *Template inheritance* for more information.

block

Defines a block that can be overridden by child templates. See *Template inheritance* for more information.

Built-in Filters

capitalize

The capitalize filter allows you to capitalize a string. For example, *stencil* to *Stencil*.

```
{{ "stencil"|capitalize }}
```

uppercase

The uppercase filter allows you to transform a string to uppercase. For example, *Stencil* to *STENCIL*.

```
{{ "Stencil"|uppercase }}
```

lowercase

The uppercase filter allows you to transform a string to lowercase. For example, *Stencil* to *stencil*.

```
{{ "Stencil"|lowercase }}
```

default

If a variable not present in the context, use given default. Otherwise, use the value of the variable. For example:

```
Hello {{ name|default:"World" }}
```

join

Join an array of items.

```
{{ value|join:", " }}
```

Note: The value MUST be an array.

For Developers

Resources to help you integrate Stencil into a Swift project.

Installation

Swift Package Mangaer

If you're using the Swift Package Manager, you can add `Stencil` to your dependencies inside `Package.swift`.

```
import PackageDescription

let package = Package(
  name: "MyApplication",
  dependencies: [
    .Package(url: "https://github.com/kylef/Stencil.git", majorVersion: 0, minor: 8),
  ]
)
```

CocoaPods

If you're using CocoaPods, you can add Stencil to your `Podfile` and then run `pod install`.

```
pod 'Stencil', '~> 0.8.0'
```

Carthage

Note: Use at your own risk. We don't offer support for Carthage and instead recommend you use Swift Package Manager.

1. Add `Stencil` to your `Cartfile`:

```
github "kylef/Stencil" ~> 0.8.0
```

2. Checkout your dependencies, generate the Stencil Xcode project, and then use Carthage to build Stencil:

```
$ carthage update
$ (cd Carthage/Checkouts/Stencil && swift package generate-xcodeproj)
$ carthage build
```

3. Follow the Carthage steps to add the built frameworks to your project.

To learn more about this approach see Using Swift Package Manager with Carthage.

Getting Started

The easiest way to render a template using Stencil is to create a template and call render on it providing a context.

```
let template = Template(templateString: "Hello {{ name }}")
try template.render(["name": "kyle"])
```

For more advanced uses, you would normally create an `Environment` and call the `renderTemplate` convinience method.

```
let environment = Environment()

let context = ["name": "kyle"]
try template.renderTemplate(string: "Hello {{ name }}", context: context)
```

Template Loaders

A template loader allows you to load files from disk or elsewhere. Using a `FileSystemLoader` we can easily render a template from disk.

For example, to render a template called `index.html` inside the `templates/` directory we can use the following:

```
let fsLoader = FileSystemLoader(paths: ["templates/"])
let environment = Environment(loader: fsLoader)

let context = ["name": "kyle"]
try template.renderTemplate(name: "index.html", context: context)
```

Template API

This document describes Stencils Swift API, and not the Swift template language.

Contents

Environment

An environment contains shared configuration such as custom filters and tags along with template loaders.

```
let environment = Environment()
```

You can optionally provide a loader or extensions when creating an environment:

```
let environment = Environment(loader: ..., extensions: [...])
```

Rendering a Template

Environment provides convinience methods to render a template either from a string or a template loader.

```
let template = "Hello {{ name }}"
let context = ["name": "Kyle"]
let rendered = environment.renderTemplate(string: template, context: context)
```

Rendering a template from the configured loader:

```
let context = ["name": "Kyle"]
let rendered = environment.renderTemplate(name: "example.html", context: context)
```

Loading a Template

Environment provides an API to load a template from the configured loader.

```
let template = try environment.loadTemplate(name: "example.html")
```

Loader

Loaders are responsible for loading templates from a resource such as the file system.

Stencil provides a `FileSytemLoader` which allows you to load a template directly from the file system.

FileSystemLoader

Loads templates from the file system. This loader can find templates in folders on the file system.

```
FileSystemLoader(paths: ["./templates"])
```

```
FileSystemLoader(bundle: [Bundle.main])
```

Custom Loaders

`Loader` is a protocol, so you can implement your own compatible loaders. You will need to implement a `loadTemplate` method to load the template, throwing a `TemplateDoesNotExist` when the template is not found.

```
class ExampleMemoryLoader: Loader {
  func loadTemplate(name: String, environment: Environment) throws -> Template {
    if name == "index.html" {
      return Template(templateString: "Hello", environment: environment)
    }

    throw TemplateDoesNotExist(name: name, loader: self)
  }
}
```

Context

A `Context` is a structure containing any templates you would like to use in a template. It's somewhat like a dictionary, however you can push and pop to scope variables. So that means that when iterating over a for loop, you can push a new scope into the context to store any variables local to the scope.

You would normally only access the `Context` within a custom template tag or filter.

Subscripting

You can use subscripting to get and set values from the context.

```
context["key"] = value
let value = context["key"]
```

`push()`

A `Context` is a stack. You can push a new level onto the `Context` so that modifications can easily be poped off. This is useful for isolating mutations into scope of a template tag. Such as `{% if %}` and `{% for %}` tags.

```
context.push(["name": "example"]) {
    // context contains name which is `example`.
}

// name is popped off the context after the duration of the closure.
```

`flatten()`

Using `flatten()` method you can get whole `Context` stack as one dictionary including all variables.

```
let dictionary = context.flatten()
```

Custom Template Tags and Filters

You can build your own custom filters and tags and pass them down while rendering your template. Any custom filters or tags must be registered with a extension which contains all filters and tags available to the template.

```
let ext = Extension()
// Register your filters and tags with the extension

let environment = Environment(extensions: [ext])
try environment.renderTemplate(name: "example.html")
```

Custom Filters

Registering custom filters:

```
ext.registerFilter("double") { (value: Any?) in
  if let value = value as? Int {
    return value * 2
  }
```

```
  return value
}
```

Registering custom filters with arguments:

```
ext.registerFilter("multiply") { (value: Any?, arguments: [Any?]) in
  let amount: Int

  if let value = arguments.first as? Int {
    amount = value
  } else {
    throw TemplateSyntaxError("multiple tag must be called with an integer argument")
  }

  if let value = value as? Int {
    return value * 2
  }

  return value
}
```

Custom Tags

You can build a custom template tag. There are a couple of APIs to allow you to write your own custom tags. The following is the simplest form:

```
ext.registerSimpleTag("custom") { context in
  return "Hello World"
}
```

When your tag is used via `{% custom %}` it will execute the registered block of code allowing you to modify or retrieve a value from the context. Then return either a string rendered in your template, or throw an error.

If you want to accept arguments or to capture different tokens between two sets of template tags. You will need to call the `registerTag` API which accepts a closure to handle the parsing. You can find examples of the `now`, `if` and `for` tags found inside Stencil source code.