Stencil Documentation

Release 0.7.0

Kyle Fuller

January 18, 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>
```

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

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

 do {
   let template = try Template(named: "template.html")
   let rendered = try template.render(context)
   print(rendered)
 } catch {
   print("Failed to render template \(error)")
 }
```

Contents:

CHAPTER 1

Templates

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

1.1 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 }}.
```

1.1.1 Filters

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

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

See *all builtin filters*.

1.2 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*.

1.3 Comments

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

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

CHAPTER 2

Built-in template tags and filters

2.1 Built-in Tags

2.1.1 `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

2.1.2 `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.

2.1.3 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 %}
```

2.1.4 now

2.1.5 include

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

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

The *include* tag requires a FileSystemLoader to be found inside your context with the paths, or bundles used to lookup the template.

```
let context = Context(dictionary: [
  "loader": FileSystemLoader(bundle: [NSBundle.mainBundle()])
])
```

2.1.6 extends

2.1.7 block

2.2 Built-in Filters

2.2.1 capitalize

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

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

2.2.2 uppercase

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

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

2.2.3 lowercase

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

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

2.2.4 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" }}
```

2.2.5 join

Join an array with a string.

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

Note: The value MUST be an array of Strngs and the separator must be a string.

CHAPTER 3

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 can initialise a `Context` with a `Dictionary`.

```
Context(dictionary: [String: Any]? = nil)
```

3.1 API

3.1.1 Subscripting

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

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

3.1.2 `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.
```

3.1.3 `flatten()`

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

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

CHAPTER 4

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 namespace which contains all filters and tags available to the template.

```
let namespace = Namespace()
// Register your filters and tags with the namespace
let rendered = try template.render(context, namespace: namespace)
```

4.1 Custom Filters

Registering custom filters:

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

  return value
}
```

Registering custom filters with arguments:

```
namespace.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
}
```

4.2 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:

```
namespace.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.