Documentation

Hummingbird Document… / Mustache Features

Article

Mustache Features

An overview of the features of swift-mustache.

Lambdas

The library provides support for mustache lambdas via the type `MustacheLambda`.

Rendering variables

The mustache manual section for mustache lambdas when rendered as variables states.

> **Manual**
>
> If any value found during the lookup is a callable object, such as a function or lambda, this object will be invoked with zero arguments. The value that is returned is then used instead of the callable object itself.
>
> An optional part of the specification states that if the final key in the name is a lambda that returns a string, then that string should be rendered as a Mustache template before interpolation. It will be rendered using the default delimiters (see Set Delimiter below) against the current context.

Swift Mustache supports both parts of the specification of lambdas when rendered as variables. Instead of a callable object, swift-mustache requires the type to be a `MustacheLambda` initialized with a closure that has no parameters.

> **Note**
>
> If the lambda is rendered as a variable and you supply a closure that accepts a `String` then the supplied `String` is empty.

Below we have a couple of examples of rendering mustache lambdas as variables. One returning a tuple and one returning a `String` which is then parsed as a template. If we have the following object

```
let object: [String: Any] = [
    "year": 1970,
    "month": 1,
    "day": 1,
    "time": MustacheLambda {
        (hour: 0, minute: 0, second: 0)
    },
    "today": MustacheLambda { _ in
        return "{{year}}-{{month}}-{{day}}"
    },
]
```

and the following mustache template

```
let mustache = """
    * {{time.hour}}
    * {{today}}
    """
let template = try MustacheTemplate(string: mustache)
```

then `template.render(object)` will output

```
* 0
* 1970-1-1
```

In this example the first part of the template calls lambda `time` and then uses `hour` from the return object. In the second part the `today` lambda returns a string which is then parsed as mustache and renders the year.

Rendering sections

The mustache manual section for mustache lambdas when rendered as a section states.

> **Manual**
>
> When any value found during the lookup is a callable object, such as a function or lambda, the object will be invoked and passed the block of text. The text passed is the literal block, unrendered. {{tags}} will not have been expanded.
>
> An optional part of the specification states that if the final key in the name is a lambda that returns a string, then that string replaces the content of the section. It will be rendered using the same delimiters as the original section content. In this way you can implement filters or caching.

Swift Mustache does not support the part of the specification of lambdas when rendered as sections pertaining to delimiters. As with variables, instead of a callable object, swift-mustache requires the type to be a `MustacheLambda` which can be initialized with either a closure that accepts a String or nothing. When the lambda is rendered as a section the supplied `String` is the contents of the section.

If we have an object as follows

```
let object: [String: Any] = [
  "name": "Willy",
  "wrapped": MustacheLambda { text in
    return "<b>" + text + "</b>"
  }
]
```

and the following mustache template

```
let mustache = "{{#wrapped}}{{name}} is awesome.{{/wrapped}}"
let template = try MustacheTemplate(string: mustache)
```

Then `template.render(object)` will output

```
<b>Willy is awesome.</b>
```

Here when the `wrapped` section is rendered the text inside the section is passed to the `wrapped` lambda and the resulting text passed back is parsed as a new template.

Template inheritance and parents

Template inheritance allows you to override elements of an included partial. It allows you to create a base page template, or parent as it is called in the mustache manual, and override elements of it with your page content. A parent that includes overriding elements is indicated with a `{{<parent}}`. Note this is different from the normal partial reference which uses >. This is a section tag so needs a ending tag as well. Inside the section the tagged sections to override are added using the syntax `{{$tag}}contents{{/tag}}`.

If your template is as follows

```
{{! mypage.mustache }}
{{<base}}
{{$head}}<title>My page title</title>{{/head}}
{{$body}}Hello world{{/body}}
{{/base}}
```

And you partial is as follows

```
{{! base.mustache }}
<html>
<head>
{{$head}}{{/head}}
</head>
<body>
```

```
{{$body}}Default text{{/body}}
</body>
</html>
```

You would get the following output when rendering `mypage.mustache`.

```
<html>
<head>
<title>My page title</title>
</head>
<body>
Hello world
</body>
```

Note the `{{$head}}` section in `base.mustache` is replaced with the `{{$head}}` section included inside the `{{<base}}` partial reference from `mypage.mustache`. The same occurs with the `{{$body}}` section. In that case though a default value is supplied for the situation where a `{{$body}}` section is not supplied.

Pragmas/Configuration variables

The syntax `{{% var: value}}` can be used to set template rendering configuration variables specific to Hummingbird Mustache. The only variable you can set at the moment is `CONTENT_TYPE`. This can be set to either to `HTML` or `TEXT` and defines how variables are escaped. A content type of `TEXT` means no variables are escaped and a content type of `HTML` will do HTML escaping of the rendered text. The content type defaults to `HTML`.

Given input object <>, template

```
{{%CONTENT_TYPE: HTML}}{{.}}
```

will render as `&lt;&gt;` and

```
{{%CONTENT_TYPE: TEXT}}{{.}}
```

will render as <>.

Transforms

Transforms are specific to this implementation of Mustache. They are similar to Lambdas but instead of generating rendered text they allow you to transform an object into another. Transforms are formatted as a function call inside a tag eg

```
{{uppercase(string)}}
```

They can be applied to variable, section and inverted section tags. If you apply them to a section or inverted section tag the transform name should be included in the end section tag as well eg

```
{{#sorted(array)}}{{.}}{{/sorted(array)}}
```

The library comes with a series of transforms for the Swift standard objects.

- String/Substring
  - capitalized: Return string with first letter capitalized
  - lowercase: Return lowercased version of string
  - uppercase: Return uppercased version of string
  - reversed: Reverse string
- Int/UInt/Int8/Int16…
  - equalzero: Returns if equal to zero
  - plusone: Add one to integer
  - minusone: Subtract one from integer
  - odd: return if integer is odd
  - even: return if integer is even
- Array
  - first: Return first element of array
  - last: Return last element of array

  - count: Return number of elements in array
  - empty: Returns if array is empty
  - reversed: Reverse array
  - sorted: If the elements of the array are comparable sort them
- Dictionary
  - count: Return number of elements in dictionary
  - empty: Returns if dictionary is empty
  - enumerated: Return dictionary as array of key, value pairs
  - sorted: If the keys are comparable return as array of key, value pairs sorted by key

If a transform is applied to an object that doesn't recognise it then `nil` is returned.

Sequence context transforms

Sequence context transforms are transforms applied to the current position in the sequence. They are formatted as a function that takes no parameter eg

```
{{#array}}{{.}}{{^last()}}, {{/last()}}{{/array}}
```

This will render an array as a comma separated list. The inverted section of the `last()` transform ensures we don't add a comma after the last element.

The following sequence context transforms are available

- first: Is this the first element of the sequence
- last: Is this the last element of the sequence
- index: Returns the index of the element within the sequence
- odd: Returns if the index of the element is odd
- even: Returns if the index of the element is even

Custom transforms

You can add transforms to your own objects. Conform the object to `MustacheTransformable` and provide an implementation of the function `transform`. eg

```
struct Object: MustacheTransformable {
    let either: Bool
    let or: Bool

    func transform(_ name: String) -> Any? {
        switch name {
        case "eitherOr":
            return either || or
        default:
            break
        }
        return nil
    }
}
```

When we render an instance of this object with `either` or `or` set to true using the following template it will render “Success”.

```
{{#eitherOr(object)}}Success{{/eitherOr(object)}}
```

With this we have got around the fact it is not possible to do logical OR statements in Mustache.

See Also

Related Documentation

`struct MustacheTemplate`

Class holding Mustache template

`struct MustacheLibrary`

Class holding a collection of mustache templates.

Mustache

Mustache Syntax

Overview of Mustache Syntax