MUSTACHE(5) MUSTACHE MANUAL MUSTACHE(5)

## NAME

**mustache** - Logic-less templates.

## SYNOPSIS

A typical Mustache template:

```
Hello {{name}}
You have just won {{value}} dollars!
{{#in_ca}}
Well, {{taxed_value}} dollars, after taxes.
{{/in_ca}}
```

Given the following hash:

```
{
  "name": "Chris",
  "value": 10000,
  "taxed_value": 10000 - (10000 * 0.4),
  "in_ca": true
}
```

Will produce the following:

```
Hello Chris
You have just won 10000 dollars!
Well, 6000.0 dollars, after taxes.
```

## DESCRIPTION

Mustache can be used for HTML, config files, source code - anything. It works by expanding tags in a template using values provided in a hash or object.

We call it "logic-less" because there are no if statements, else clauses, or for loops. Instead there are only tags. Some tags are replaced with a value, some nothing, and others a series of values. This document explains the different types of Mustache tags.

## TAG TYPES

Tags are indicated by the double mustaches. **{{person}}** is a tag, as is **{{#person}}**. In both examples, we'd refer to **person** as the key or tag key. Let's talk about the different types of tags.

### Variables

The most basic tag type is the variable. A **{{name}}** tag in a basic template will try to find the **name** key in the current context. If there is no **name** key, the parent contexts will be checked recursively. If the top context is reached and the **name** key is still not found, nothing will be rendered.

All variables are HTML escaped by default. If you want to return unescaped HTML, use the triple mustache: **{{{name}}}**.

You can also use **&** to unescape a variable: **{{& name}}**. This may be useful when changing delimiters (see "Set Delimiter" below).

By default a variable "miss" returns an empty string. This can usually be configured in your Mustache library. The Ruby version of Mustache supports raising an exception in this situation, for instance.

Template:

```
* {{name}}
* {{age}}
* {{company}}
* {{{company}}}
```

Hash:

```
{
  "name": "Chris",
  "company": "<b>GitHub</b>"
}
```

Output:

```
* Chris
*
* &lt;b&gt;GitHub&lt;/b&gt;
* <b>GitHub</b>
```

**Sections**

Sections render blocks of text one or more times, depending on the value of the key in the current context.

A section begins with a pound and ends with a slash. That is, **{{#person}}** begins a "person" section while **{{/person}}** ends it.

The behavior of the section is determined by the value of the key.

**False Values or Empty Lists**

If the **person** key exists and has a value of false or an empty list, the HTML between the pound and slash will not be displayed.

Template:

```
Shown.
{{#person}}
  Never shown!
{{/person}}
```

Hash:

```
{
  "person": false
}
```

Output:

```
Shown.
```

**Non-Empty Lists**

If the **person** key exists and has a non-false value, the HTML between the pound and slash will be rendered and displayed one or more times.

When the value is a non-empty list, the text in the block will be displayed once for each item in the list. The context of the block will be set to the current item for each iteration. In this way we can loop over collections.

Template:

```
{{#repo}}
  <b>{{name}}</b>
{{/repo}}
```

Hash:

```
{
  "repo": [
```

```
      { "name": "resque" },
      { "name": "hub" },
      { "name": "rip" }
    ]
  }
```

Output:

```
  <b>resque</b>
  <b>hub</b>
  <b>rip</b>
```

**Lambdas**

When the value 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 - the lambda should do that on its own. In this way you can implement filters or caching.

Template:

```
  {{#wrapped}}
    {{name}} is awesome.
  {{/wrapped}}
```

Hash:

```
  {
    "name": "Willy",
    "wrapped": function() {
      return function(text, render) {
        return "<b>" + render(text) + "</b>"
      }
    }
  }
```

Output:

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

**Non-False Values**

When the value is non-false but not a list, it will be used as the context for a single rendering of the block.

Template:

```
{{#person?}}
  Hi {{name}}!
{{/person?}}
```

Hash:

```
{
  "person?": { "name": "Jon" }
}
```

Output:

```
Hi Jon!
```

**Inverted Sections**

An inverted section begins with a caret (hat) and ends with a slash. That is **{{^person}}** begins a "person" inverted section while **{{/person}}** ends it.

While sections can be used to render text one or more times based on the value of the key, inverted sections may render text once based on the inverse value of the key. That is, they will be rendered if the key doesn't exist, is false, or is an empty list.

Template:

```
{{#repo}}
  <b>{{name}}</b>
{{/repo}}
{{^repo}}
  No repos :(
{{/repo}}
```

Hash:

```
{
  "repo": []
}
```

Output:

```
No repos :(
```

### Comments

Comments begin with a bang and are ignored. The following template:

```
<h1>Today{{! ignore me }}.</h1>
```

Will render as follows:

```
<h1>Today.</h1>
```

Comments may contain newlines.

### Partials

Partials begin with a greater than sign, like **{{> box}}**.

Partials are rendered at runtime (as opposed to compile time), so recursive partials are possible. Just avoid infinite loops.

They also inherit the calling context. Whereas in ERB you may have this:

```
<%= partial :next_more, :start => start, :size => size %>
```

Mustache requires only this:

```
{{> next_more}}
```

Why? Because the **next_more.mustache** file will inherit the **size** and **start** methods from the calling context.

In this way you may want to think of partials as includes, or template expansion, even though it's not literally true.

For example, this template and partial:

```
base.mustache:
<h2>Names</h2>
{{#names}}
  {{> user}}
{{/names}}

user.mustache:
<strong>{{name}}</strong>
```

Can be thought of as a single, expanded template:

```
<h2>Names</h2>
{{#names}}
  <strong>{{name}}</strong>
{{/names}}
```

### Set Delimiter

Set Delimiter tags start with an equal sign and change the tag delimiters from **{{** and **}}** to custom strings.

Consider the following contrived example:

```
* {{default_tags}}
{{=<% %>=}}
* <% erb_style_tags %>
<%={{ }}=%>
* {{ default_tags_again }}
```

Here we have a list with three items. The first item uses the default tag style, the second uses erb style as defined by the Set Delimiter tag, and the third returns to the default style after yet another Set Delimiter declaration.

According to ctemplates, this "is useful for languages like TeX, where double-braces may occur in the text and are awkward to use for markup."

Custom delimiters may not contain whitespace or the equals sign.

## COPYRIGHT



Original CTemplate by Google

## SEE ALSO

mustache(1), http://mustache.github.io/