Liquid for Designers

Jérôme Coupé edited this page 21 days ago · 31 revisions

There are two types of markup in Liquid: Output and Tag.

- Output markup (which may resolve to text) is surrounded by

```
{{ matched pairs of curly brackets (ie, braces) }}
```

- Tag markup (which cannot resolve to text) is surrounded by

```
{% matched pairs of curly brackets and percent signs %}
```

Output

Here is a simple example of Output:

```
Hello {{name}}
Hello {{user.name}}
Hello {{ 'tobi' }}
```

Advanced output: Filters

Output markup takes filters. Filters are simple methods. The first parameter is always the output of the left side of the filter. The return value of the filter will be the new left value when the next filter is run. When there are no more filters, the template will receive the resulting string.

```
Hello {{ 'tobi' | upcase }}
Hello tobi has {{ 'tobi' | size }} letters!
Hello {{ '*tobi*' | textilize | upcase }}
Hello {{ 'now' | date: "%Y %h" }}
```

Standard Filters

- `date` - reformat a date (syntax reference)
- `capitalize` - capitalize words in the input sentence
- `downcase` - convert an input string to lowercase
- `upcase` - convert an input string to uppercase
- `first` - get the first element of the passed in array
- `last` - get the last element of the passed in array
- `join` - join elements of the array with certain character between them
- `sort` - sort elements of the array
- `map` - map/collect an array on a given property

- `size` - return the size of an array or string
- `escape` - escape a string
- `escape_once` - returns an escaped version of html without affecting existing escaped entities
- `strip_html` - strip html from string
- `strip_newlines` - strip all newlines (\n) from string
- `newline_to_br` - replace each newline (\n) with html break
- `replace` - replace each occurrence *e.g.* `{{ 'foofoo' | replace:'foo','bar' }} #=> 'barbar'`
- `replace_first` - replace the first occurrence *e.g.* `{{ 'barbar' | replace_first:'bar','foo' }} #=> 'foobar'`
- `remove` - remove each occurrence *e.g.* `{{ 'foobarfoobar' | remove:'foo' }} #=> 'barbar'`
- `remove_first` - remove the first occurrence *e.g.* `{{ 'barbar' | remove_first:'bar' }} #=> 'bar'`
- `truncate` - truncate a string down to x characters. It also accepts a second parameter that will append to the string *e.g.* `{{ 'foobarfoobar' | truncate: 5, '.' }} #=> 'foob.'`
- `truncatewords` - truncate a string down to x words
- `prepend` - prepend a string *e.g.* `{{ 'bar' | prepend:'foo' }} #=> 'foobar'`
- `pluralize` - return the second word if the input is not `1`, otherwise return the first word *e.g.* `{{ 3 | pluralize: 'item', 'items' }} #=> 'items'`
- `append` - append a string *e.g.* `{{ 'foo' | append:'bar' }} #=> 'foobar'`
- `slice` - slice a string. Takes an offset and length, *e.g.* `{{ "hello" | slice: -3, 3 }} #=> llo`
- `minus` - subtraction *e.g.* `{{ 4 | minus:2 }} #=> 2`
- `plus` - addition *e.g.* `{{ '1' | plus:'1' }} #=> 2`, `{{ 1 | plus:1 }} #=> 2`
- `times` - multiplication *e.g* `{{ 5 | times:4 }} #=> 20`
- `divided_by` - integer division *e.g.* `{{ 10 | divided_by:3 }} #=> 3`
- `round` - rounds input to the nearest integer or specified number of decimals
- `split` - split a string on a matching pattern *e.g.* `{{ "a~b" | split:"~" }} #=> ['a','b']`
- `modulo` - remainder, *e.g.* `{{ 3 | modulo:2 }} #=> 1`
- `reverse` - reverse sort the passed in array

Tags

Tags are used for the logic in your template. New tags are very easy to code, so I hope to get many contributions to the standard tag library after releasing this code.

Here is a list of currently supported tags:

- **assign** - Assigns some value to a variable
- **capture** - Block tag that captures text into a variable
- **case** - Block tag, its the standard case...when block
- **comment** - Block tag, comments out the text in the block
- **cycle** - Cycle is usually used within a loop to alternate between values, like colors or DOM classes.
- **for** - For loop
- **break** - Exits a for loop
- **continue** Skips the remaining code in the current for loop and continues with the next loop
- **if** - Standard if/else block
- **include** - Includes another template; useful for partials
- **raw** - temporarily disable tag processing to avoid syntax conflicts.

- **unless** - Mirror of if statement

Comments

Any content that you put between `{% comment %}` and `{% endcomment %}` tags is turned into a comment.

```
We made 1 million dollars {% comment %} in losses {% endcomment %} this year
```

Raw

Raw temporarily disables tag processing. This is useful for generating content (eg, Mustache, Handlebars) which uses conflicting syntax.

```
{% raw %}
  In Handlebars, {{ this }} will be HTML-escaped, but {{{ that }}} will not.
{% endraw %}
```

If / Else

`if / else` should be well-known from any other programming language. Liquid allows you to write simple expressions in the `if` or `unless` (and optionally, `elsif` and `else`) clause:

```
{% if user %}
  Hello {{ user.name }}
{% endif %}


# Same as above
{% if user != null %}
  Hello {{ user.name }}
{% endif %}


{% if user.name == 'tobi' %}
  Hello tobi
{% elsif user.name == 'bob' %}
  Hello bob
{% endif %}


{% if user.name == 'tobi' or user.name == 'bob' %}
  Hello tobi or bob
{% endif %}


{% if user.name == 'bob' and user.age > 45 %}
  Hello old bob
{% endif %}


{% if user.name != 'tobi' %}
  Hello non-tobi
{% endif %}
```

```
# Same as above
{% unless user.name == 'tobi' %}
  Hello non-tobi
{% endunless %}



# Check for the size of an array
{% if user.payments == empty %}
   you never paid !
{% endif %}

{% if user.payments.size > 0  %}
   you paid !
{% endif %}



{% if user.age > 18 %}
   Login here
{% else %}
   Sorry, you are too young
{% endif %}



# array = 1,2,3
{% if array contains 2 %}
   array includes 2
{% endif %}



# string = 'hello world'
{% if string contains 'hello' %}
   string includes 'hello'
{% endif %}
```

Case Statement

If you need more conditions, you can use the `case` statement:

```
{% case condition %}
{% when 1 %}
hit 1
{% when 2 or 3 %}
hit 2 or 3
{% else %}
... else ...
{% endcase %}
```

Example:

```
{% case template %}

{% when 'label' %}
     // {{ label.title }}
{% when 'product' %}
     // {{ product.vendor | link_to_vendor }} / {{ product.title }}
{% else %}
     // {{page_title}}
{% endcase %}
```

Cycle

Often you have to alternate between different colors or similar tasks. Liquid has built-in support for such operations, using the `cycle` tag.

```
{% cycle 'one', 'two', 'three' %}
{% cycle 'one', 'two', 'three' %}
{% cycle 'one', 'two', 'three' %}
{% cycle 'one', 'two', 'three' %}
```

will result in

```
one
two
three
one
```

If no name is supplied for the cycle group, then it's assumed that multiple calls with the same parameters are one group.

If you want to have total control over cycle groups, you can optionally specify the name of the group. This can even be a variable.

```
{% cycle 'group 1': 'one', 'two', 'three' %}
{% cycle 'group 1': 'one', 'two', 'three' %}
{% cycle 'group 2': 'one', 'two', 'three' %}
{% cycle 'group 2': 'one', 'two', 'three' %}
```

will result in

```
one
two
one
two
```

For loops

Liquid allows `for` loops over collections:

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

When iterating a hash, `item[0]` contains the key, and `item[1]` contains the value:

```
{% for item in hash %}
  {{ item[0] }}: {{ item[1] }}
{% endfor %}
```

During every `for` loop, the following helper variables are available for extra styling needs:

```
forloop.length      # => length of the entire for loop
forloop.index       # => index of the current iteration
forloop.index0      # => index of the current iteration (zero based)
```

```
forloop.rindex      # => how many items are still left?
forloop.rindex0     # => how many items are still left? (zero based)
forloop.first       # => is this the first iteration?
forloop.last        # => is this the last iteration?
```

There are several attributes you can use to influence which items you receive in your loop

`limit:int` lets you restrict how many items you get. `offset:int` lets you start the collection with the nth item.

```
# array = [1,2,3,4,5,6]
{% for item in array limit:2 offset:2 %}
  {{ item }}
{% endfor %}
# results in 3,4
```

Reversing the loop

```
{% for item in collection reversed %} {{item}} {% endfor %}
```

Instead of looping over an existing collection, you can define a range of numbers to loop through. The range can be defined by both literal and variable numbers:

```
# if item.quantity is 4...
{% for i in (1..item.quantity) %}
  {{ i }}
{% endfor %}
# results in 1,2,3,4
```

A for loop can take an optional `else` clause to display a block of text when there are no items in the collection:

```
# items => []
{% for item in items %}
   {{ item.title }}
{% else %}
   There are no items!
{% endfor %}
```

Variable Assignment

You can store data in your own variables, to be used in output or other tags as desired. The simplest way to create a variable is with the `assign` tag, which has a pretty straightforward syntax:

```
{% assign name = 'freestyle' %}

{% for t in collections.tags %}{% if t == name %}
  <p>Freestyle!</p>
{% endif %}{% endfor %}
```

Another way of doing this would be to assign `true / false` values to the variable:

```
{% assign freestyle = false %}
```

```
{% for t in collections.tags %}{% if t == 'freestyle' %}
  {% assign freestyle = true %}
{% endif %}{% endfor %}

{% if freestyle %}
  <p>Freestyle!</p>
{% endif %}
```

If you want to combine a number of strings into a single string and save it to a variable, you can do that with the `capture` tag. This tag is a block which "captures" whatever is rendered inside it, then assigns the captured value to the given variable instead of rendering it to the screen.

```
{% capture attribute_name %}{{ item.title | handleize }}-{{ i }}-color{% endcaptur

<label for="{{ attribute_name }}">Color:</label>
<select name="attributes[{{ attribute_name }}]" id="{{ attribute_name }}">
  <option value="red">Red</option>
  <option value="green">Green</option>
  <option value="blue">Blue</option>
</select>
```