HTML Templating

Contents

Templating Options

Luminus comes with the Hiccup dependency. If you're familiar with Hiccup then you can start using it out of the box.

Hiccup uses standard Clojure data structures to represent its templates. On top of that, Hiccup provides a rich API of helper functions for generating common HTML elements.

Luminus also packages Selmer for more traditional style templating using plain text files as templates.

You can choose to use either templating engine or combine them. Alternatively, you can choose to use a different templating engine altogether. A couple of popular options are Enlive and Stencil.

HTML Templating Using Selmer

Selmer is a templating language similar to Django and Rails templates. If you're familiar with Django or similar templating languages you should feel right at home.

Creating Templates

By design, Selmer separates the presentation logic from the program logic. The templates are simply HTML files with additional template tags for dynamic elements. The dynamc elements in the template are resolved during the rendering step. Let's take a look at an example template below:

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UT
        <title>My First Template</title>
    </head>
    <body>
        <h2>Hello {{name}}</h2>
    </body>
</html>
```

The templates are rendered using a context represented by a map of key/value pairs. The context contains the variables that we'd like to render in our template at runtime. Above, we have a template representing a page that renders a single variable called `name`.

There are two functions for rendering templates called `render` and `render-file`. The `render` function accepts a string representing the template, while the `render-file` function accepts a string representing the path to the file containing the template.

If we saved the template defined above in a file called `index.html` then we could render it as follows:

```
(ns example.routes.home
  (:require [selmer.parser :refer [render-file]]))

(render-file "html/index.html" {:name "John"})
```

The `render-file` function expects the templates to be found at a path relative to the `resources` folder of the application.

Above, we passed in a string as the value for the variable `name`. However, we're not restricted to strings and can pass in any type we like. For example, if we pass in a collection we can iterate it using the `for` tag:

```
<ul>
{% for item in items %}
<li> {{item}} </li>
{% endfor %}
</ul>


(render-file "html/items.html" {:items (range 10)})
```

If an item happens to be a map, we can access the keys by their name as follows:

```
(render "<p>Hello {{user.first}} {{user.last}}</p>"
        {:user {:first "John" :last "Doe"}})
```

When no special processing is specified in the template then the `.toString` value of the parameter will be used.

By default Selmer caches the compiled template. A recompile will be triggered if the last modified timestamp of the files changes. Alternatively you can turn caching on and off calling `(selmer.parser/cache-on!)` and `(selmer.parser/cache-off!)` respectively.

Filters

Filters allow post-processing of the variables before they are rendered. For example, you can use a filter to convert the variable to upper case, compute a hash, or count the length. Filters are specified by using a `|` after the variable name, as can be seen below:

```
{{name|upper}}
```

The following built-in filters are available:

add

`(render "{{add_me|add:2:3:4}}" {:add_me 2})` => `11`

addslashes

Nota bene, the slashes aren't actually in the input string, but they *are* going to be in the input. Just trying to write valid Clojure code.

```
(render "{{name|addslashes}}" {:name "\"Russian tea is best tea\""})
=> "\"Russian tea is best tea\""
```

block.super

Can be used inside a block to insert the content from the parent block in its place

```
{% block foo %} {{block.super}} some content{% endblock %}
```

capitalize

```
(render "{{name|capitalize}}" {:name "russian tea is best tea"}) =>
"Russian tea is best tea"
```

center

`(render "{{name|center:20}}" {:name "bitemyapp"})` => `" bitemyapp "`

count

`(render "{{name|count}}" {:name "Yogthos"})` => `"7"`

```
(render "{{items|count}}" {:items [1 2 3 4]}) => "4"
```

date

```
(render "{{creation-time|date:\"yyyy-MM-dd_HH:mm:ss\"}}" {:created-at
(java.util.Date.)}) => "2013-07-28_20:51:48"
```

default

```
(render "{{name|default:"I <3 ponies"}}" {:name "bitemyapp"}) =>
"bitemyapp"

(render "{{name|default:"I <3 ponies"}}" {:name nil}) => "I <3 ponies"

(render "{{name|default:"I <3 ponies"}}" {:name []}) => "[]"

(render "{{name|default:"I <3 ponies"}}" {}) => "I <3 ponies"
```

default-if-empty

```
(render "{{name|default-if-empty:"I <3 ponies"}}" {:name "bitemyapp"})
=> "bitemyapp"

(render "{{name|default-if-empty:"I <3 ponies"}}" {:name nil}) => "I <3
ponies"

(render "{{name|default-if-empty:"I <3 ponies"}}" {:name []}) => "I <3
ponies"

(render "{{name|default-if-empty:"I <3 ponies"}}" {}) => "I <3 ponies"
```

double-format

```
(render "{{tis-a-number|double-format:2}}" {:tis-a-number 10.00001})
=> 10.00

(render "{{tis-a-number|double-format}}" {:tis-a-number 10.00001}) =>
10.0
```

first

```
(render "{{seq-of-some-sort|first}}" {:seq-of-some-sort [:dog :cat
:bird :bird :bird :is :the :word]}) => :dog
```

get-digit

returns digits from the right, 1 indexing on the last digit

```
(render "{{tis-a-number|get-digit:1}}" {:tis-a-number 12.34567}) => 7
```

hash

available hashes: `md5`, `sha`, `sha256`, `sha384`, `sha512`

```
(render "{{domain|hash:\"md5\"}}" {:domain "example.org"}) =>
"1bdf72e04d6b50c82a48c7e4dd38cc69"
```

join

```
(render "{{sequence|join}}" {:sequence [1 2 3 4]}) => "1234"
```

json

by default content will be escaped

```
(render "{{data|json}}" {:data [1 2 {:foo 27 :dan "awesome"}]}) => "
[1,2,{&amp;quot;foo&amp;quot;:27,&amp;quot;dan&amp;quot;:&amp;quot;awesome&amp;quot;}]"
```

if you wish to render it unescaped use the `safe` filter:

```
(render "{{f|json|safe}}" {:f {:foo 27 :dan "awesome"}})
```

last

```
(render "{{sequence|last}}" {:sequence 12.34567}) => 7

(render "{{sequence|last}}" {:sequence [1 2 3 4]}) => 4
```

length

```
(render "{{sequence|length}}" {:sequence [1 2 3 4]}) => 4
```

length-is

```
(render "{{sequence|length-is:4}}" {:sequence [1 2 3 4]}) => true
```

linebreaks

Single newlines become
, double newlines mean new paragraph. Content will be escaped by default.

```
(render "{{foo|linebreaks|safe}}" {:foo "\nbar\nbaz"}) => "<p><br
/>bar<br />baz</p>"
```

linebreaks-br

like `linebreaks` but doesn't insert `<p>` tags.

```
(render "{{foo|linebreaks-br|safe}}" {:foo "\nbar\nbaz"}) => "bar<br
/>baz"
```

linenumbers

Displays text with line numbers.

```
(render "{{foo|linenumbers" {:foo "foo\n\bar\nbaz"}) => "1. foo\n2.
\bar\n3. baz"
```

lower

```
(render "{{foo|lower}}" {:foo "FOOBaR"}) => "foobar"
```

pluralize

Returns the correct (English) pluralization based on the variable. This works with many words, but certainly not all (eg. foot/feet, mouse/mice, etc.)

```
(render "{{items|count}} item{{items|pluralize}}" {:items []}) => "0
items"

(render "{{items|count}} item{{items|pluralize}}" {:items [1]}) => "1
item"

(render "{{items|count}} item{{items|pluralize}}" {:items [1 2]}) => "2
items"

(render "{{fruit|count}} tomato{{fruit|pluralize:\"es\"}}" {:fruit
[]}) => "0 tomatoes"

(render "{{people|count}} lad{{people|pluralize:\"y\":\"ies\"}}"
{:people [1]}) => "1 lady"

(render "{{people|count}} lad{{people|pluralize:\"y\":\"ies\"}}"
{:people [1 2]}) => "2 ladies"
```

rand-nth

returns rand-nths value from a collection:

```
(render "{{foo|rand-nth}}" {:foo [1 2 3]}) => "2"
```

remove

removes specified characters from the string:

```
(render "{{foo|remove:\"aeiou\"}}" {:foo "abcdefghijklmnop"}) =>
"bcdfghjklmnp"
```

remove-tags

Removes the specified HTML tags from the string:

```
(render "{{ value|remove-tags:b:span }}" {:value "<b>
<span>foobar</span></b>"}) => "foobar"
```

safe

By default Selmer will HTML escape all variables, The `safe` filter exempts the variable from being html-escaped:

```
(render "{{data}}" {:data "<foo>"}) => "&lt;foo&gt;"

(render "{{data|safe}}" {:data "<foo>"}) => "<foo>"
```

sort

```
(render "{{ value|sort }}" {:value [1 4 2 3 5]}) => "(1 2 3 4 5)"
```

sort-by

```
(render "{{ value|sort-by:name }}" {:value [{:name "John"} {:name
"Jane"}]}) => "({:name &amp;quot;Jane&amp;quot;} {:name &amp;quot;John&amp;quot;})"
```

sort-reversed

same as sort, but in reverse order

sort-by-reversed

same as sort-by, but in reverse order

upper

```
(render "{{shout|upper}}" {:shout "hello"}) => "HELLO"
```

Defining Custom Filters

You can easily add your own filters using the `selmer.filters/add-filter!` function. The filter function should accept the element and return a value that will replace the original value.

```
(use 'selmer.filters)

(add-filter! :embiginate #(.toUpperCase %))
 (render "{{shout|embiginate}}" {:shout "hello"})

(add-filter! :count count)
(render "{{foo|count}}" {:foo (range 3)})
```

Filters can also be chained together as needed:

```
(add-filter! :empty? empty?)
(render "{{foo|upper|empty?}}" {:foo "Hello"})
```

Tags

Selmer provides two types of tags. The frist kind are inline tags such as the `extends` and `include` tags. These tags are self contained statements and do not require an end tag. The other type is the block tags. These tags have a start and an end tag, and operate on a block of text. An example of this would be the `if` ... `endif` block.

Let's take a look at the default tags:

include

replaces itself with the contents of the referenced template

```
{% include "path/to/comments.html" %}
```

optionally, you can supply default arguments any tags matching these will have the `default` filter applied using the value supplied:

```
{% include "html/inheritance/child.html" with name="Jane Doe"
greeting="Hello!" %}
```

block

Allows specifying a block of content that can be overwritten using the template inheritance discussed below.

```
{% block foo %}This text can be overridden later{% endblock %}
```

cycle

Will cycle through the supplied argument.

```
(render "{% for i in items %}<li class={% cycle \"blue\" \"white\" %}>
{{i}}</li>{% endfor %}" {:items (range 5)}) => "<li
class=\"blue\">0</li><li class=\"white\">1</li><li
class=\"blue\">2</li><li class=\"white\">3</li><li
class=\"blue\">4</li>"
```

extends

This tag is used to reference a parent template. The blocks in parents are recursively overridden by the blocks from child templates.

Note: child templates can **only** contain blocks. Any tags or text outside the blocks will beignored!

For example, say we have a base template called `base.html` and a child template `child.html`:

```
<html>
  <body>
  {% block foo %}This text can be overridden later{% endblock %}
  </body>
</html>
```

```
{% extends "base.html" %}
{% block foo %}
  <p>This text will override the text in the parent</p>
{% endblock %}
```

if

It's an `if` – only render the body if the conditional is true.

```
{% if condition %}yes!{% endif %}

{% if condition %}yes!{% else %}no!{% endif %}
```

filters work for the conditions:

```
(add-filter! :empty? empty?)
(render "{% if files|empty? %}no files{% else %}files{% endif %}"
  {:files []})
```

ifequal

Only render the body if the two args are equal (according to clojure.core/=).

```
{% ifequal foo bar %}yes!{% endifequal %}

{% ifequal foo bar %}yes!{% else %}no!{% endifequal %}

{% ifequal foo "this also works" %}yes!{% endifequal %}
```

for/endfor *block*

for

Render the body one time for each element in the list. Each render will introduce the following variables into the context:

```
forloop.first

forloop.last

forloop.counter

forloop.counter0

forloop.revcounter

forloop.revcounter0

forloop.length

{% for x in some-list %}element: {{x}} first? {{forloop.first}} last?
{{forloop.last}}{% endfor %}
```

you can also iterate over nested data structures, eg:

```
{% for item in items %} <tr><td>{{item.name}}</td><td>{{item.age}}
</td></tr> {% endfor %}
```

now

renders current time

```
(render (str "{% now \"" date-format "\"%}") {}) => "\"01 08 2013\""
```

comment

ignores any content inside the block

```
(render "foo bar {% comment %} baz test {{x}} {% endcomment %} blah"
{}) => "foo bar baz test blah"
```

firstof

renders the first occurance of supplied keys that doesn't resolve to false:

```
(render "{% firstof var1 var2 var3 %}" {:var2 "x" :var3 "not me"}) =>
"x"
```

style

The style tag will generate an HTML script tag and prepend the value of the `servlet-context` key to the URI. When `servlet-context` key is not present then the original URI is set.

```
(render "{% style \"/css/screen.css\" %}" {:servlet-context "/myapp"})
=>

"<link href=\"/myapp/css/screen.css\" rel=\"stylesheet\"
type=\"text/css\" />"
```

script

The script tag will generate an HTML script tag and prepend the value of the `servlet-context` key to the URI. When `servlet-context` key is not present then the original URI is set.

```
(render "{% script \"/js/site.js\" %}" {:servlet-context "/myapp"}) =>

"<script src=\"/myapp/js/site.js\" type=\"text/javascript\"></script>"
```

verbatim

prevents any tags inside from being parsed:

```
(render "{% verbatim %}{{if dying}}Still alive.{{/if}}{% endverbatim
%}" {}) => "{{if dying}}Still alive.{{/if}}"
```

with

injects the specified keys into the context map:

```
 (render "{% with total=business.employees|count %}{{ total }}{% endwit
          {:business {:employees (range 5)}})
```

```
=> "5 employees"
```

Defining Custom Tags

In addition to tags already provides you can easily define custom tags of your own. This is done by using the `add-tag!` macro. Let's take a look at a couple of examples to see how it works:

```
 (use 'selmer.parser)

 (add-tag! :foo
   (fn [args context-map]
     (str "foo " (first args))))

 (render "{% foo quux %} {% foo baz %}" {})
```

```
(add-tag! :bar
  (fn [args context-map content]
    (str content))
  :baz :endbar)

(render "{% bar %} some text {% baz %} some more text {% endbar %}" {}
```

As can be seen above, the tag is defined by providing a keyword specifying the tag name followed by the handler and any closing tags.

When there are no closing tags the tag will not have any content. The handler for such tags accepts the arguments defined in the tag and the context map.

When closing tags are present then the content for each block will be keyed on the opening tags. The content will be a map containing the `:args` and `:content` keys associated with each block.

Template inheritance

Selmer templates can refer to other templates using the `block` tag. There are two ways to refer to a template. We can either use the `extends` tag or the `include` tag for this.

EXTENDING TEMPLATES

When we use the `extends` tag, the current template will use the template it's extending as the base. Any blocks in the base template with the names matching the current template will be overwritten.

The content of the child template **must** be encapsulated in blocks. Any content outside the blocks present in the parent templates will be ignored.

Let's take a look at a concrete example. First, we'll define our base template and call it `base.html`:

```
<!DOCTYPE html>
<head>
    <link rel="stylesheet" href="style.css" />
    <title>{% block title %}My amazing site{% endblock %}</title>
</head>

<body>
    <div id="content">
        {% block content %}{% endblock %}
    </div>
</body>
</html>
```

Then we'll create a new template called `home.html` that will extend `base.html` as follows:

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

{% block content %}
  {% for entry in entries %}
    <h2>{{ entry.title }}</h2>
    <p>{{ entry.body }}</p>
  {% endfor %}
{% endblock %}
```

When the `home.html` is rendered the `content` block will display the entries. However, since we did not define a block for the title, the one from `base.html` will be used.

Note that you can chain extended templates together. In this case the latest occurrence of a block tag will be the one that's rendered.

INCLUDING TEMPLATES

The `include` tag allows including blocks from other templates in the current template. Let's take a look at an example. Let's say we have a `base.html` template that includes templates named `register.html` and `home.html`, then defines blocks called `register` and `home`:

```
<!DOCTYPE html>
<head>
    <link rel="stylesheet" href="style.css" />
    <title>{% block title %}My amazing site{% endblock %}</title>
</head>

<body>
  <div id="content">
    {% if user %}
    {% include "<templates path>/home.html" %}
    {% else %}
    {% include "<templates path>/register.html" %}
    {% endif %}
  </div>
</body>
</html>
```

We can now define the content for these blocks in separate template files called `register.html`:

```
{% block register %}
<form action="/register" method="POST">
  <label for="id">user id</label>
  <input id="id" name="id" type="text"></input>
  <input pass="pass" name="pass" type="text"></input>
  <input type="submit" value="register">
</form>
{% endblock %}
```

and `home.html`:

```
{% block home %}
<h1>Hello {{user}}</h1>
```

```
{% endblock %}
```

When the `base.html` is rendered it will replace the `register` and `home` include tags with the content from the templates they are referencing.

For more details please see the official documentation.

HTML Templating Using Hiccup

Hiccup is a popular HTML templating engine for Clojure. The advantage of using Hiccup is that we can use the full power of Clojure to generate and manipulate our markup. This means that you don't have to learn a separate DSL for generating your HTML with its own rules and quirks.

In Hiccup, HTML elements are represented by Clojure vectors and the structure of the element looks as following:

```
[:tag-name {:attribute-key "attribute value"} tag-body]
```

For example, if we wanted to create a div with a paragraph in it, we could write:

```
[:div {:id "hello", :class "content"} [:p "Hello world!"]]
```

which corresponds to the following HTML:

```
<div id="hello" class="content"><p>Hello world!</p></div>
```

Hiccup provides shortcuts for setting the id and class of the element, so instead of what we wrote above we could simply write:

```
[:div#hello.content [:p "Hello world!"]]
```

Hiccup also provides a number of helper functions for defining common elements such as forms, links, images, etc. All of these functions simply output vectors in the format described above. This means that if a function doesn't do what you need, you can either write out the literal form for the element by hand or take its output and modify it to fit your needs. Each function which describes an HTML element can also take optional map of attributes as its first parameter:

```
(image {:align "left"} "foo.png")
```

This would result in the following HTML:

```
<img align=\"left\" src=\"foo.png\">
```

However, it is best practice to use CSS for the actual styling of the elements in order to keep the structure separate from the representation.

Forms and Input

Hiccup also provides helpers for creating HTML forms, here's an example:

```
(form-to [:post "/login"]
  (text-field {:placeholder "screen name"} "id")
  (password-field {:placeholder "password"} "pass")
  (submit-button "login"))
```

The helper takes a vector with the type of the HTTP request specified as a keyword followed by the URL string, the rest of the arguments should be HTML elements. The above will generate the following HTML:

```
<form action="/login" method="POST">
  <input id="id" name="id" placeholder="screen name" type="text" />
  <input id="pass" name="pass" placeholder="password" type="password"
  <input type="submit" value="login" />
</form>
```

Finally, Luminus template provides a helper function under the `<yourapp>.util` namespace called `md->html`, this function will read a markdown file relative to `resources/public/` folder and return an HTML string. This can be used in conjunction with Hiccup functions, eg:

```
(:require [<yourapp>.util :as util])
...
(html [:div.container [:p (util/md->html "/md/paragraph.md")]])
```

The markdown generation is done by markdown-clj, please see the Github page for details on supported syntax.
