





haml-coffee


1.14.1 • Public • Published 9 years ago

 [Readme](#)

 [Code](#) Beta

 [3 Dependencies](#)

 [41 Dependents](#)

 [70 Versions](#)

Haml Coffee Templates build passing

Haml Coffee is a JavaScript templating solution that uses **Haml** as markup, understands inline **CoffeeScript** and generates a JavaScript function that renders to HTML. It can be used in client-side JavaScript applications that are using **Backbone.js**, **Spine.js**, **JavaScriptMVC**, **KnockoutJS** and others, or on the server-side in frameworks like **Express**.

You can try Haml Coffee online by visiting [Haml Coffee Online](#).

Installation

Haml Coffee is available in NPM and can be installed with:

```
$ npm install haml-coffee
```

Please have a look at the **CHANGELOG** when upgrading to a newer Haml Coffee version with `npm update`.

Integration

There are different packages available to integrate Haml Coffee into your workflow:

Editor

- [CoffeeScriptHaml](#) Syntax highlighting for .hamlc files in Sublime Text.

Node.JS

- [grunt-haml](#) for projects using [Grunt](#).
- [hem-haml-coffee](#) for projects using [Hem](#).
- [stitch-haml-coffee](#) for projects using [Stitch](#).
- [Mincer](#) the Sprockets inspired web assets compiler.

Ruby/Rails

- [haml_coffee_assets](#) for projects using Rails.
- [guard-haml-coffee](#) for projects using [Guard](#).

Browser

- [Haml Coffee compiler \(minified\)](#) for compiling in the browser.

The browser distribution doesn't come bundled with CoffeeScript, so you'll have to make sure you've included it before requiring haml-coffee.

Compile Haml Coffee

Using the API

You can compile a Haml Coffee template to a JavaScript function and execute the function with the locals to render the HTML. The following code

```
hamlc = require 'haml-coffee'
tmpl = hamlc.compile '%h1= @title'
html = tmpl title: 'Haml Coffee rocks!'
```

will create the HTML `<h1>Haml Coffee rocks!</h1>`.

The `compile` function can take the compiler options as second parameter to customize the template function:

```
hamlc.compile '%h1= @title'
  cleanValue: false
  escapeHtml: false
```

See the [compiler options](#) for detailed information about all the available options and browse the [codo](#) generated [Haml Coffee API documentation](#).

Using with Express

You can configure [Express](#) to use Haml Coffee as template engine.

Express 3

Starting with version 1.4.0, Haml Coffee has support for Express 3 and can be registered as view engine as follows:

```
express = require 'express'
app      = express()

app.engine 'hamlc', require('haml-coffee').__express
```

Alternatively you can also use [consolidate.js](#) to register the engine:

```
express = require 'express'
cons     = require 'consolidate'
app      = express()

app.engine 'hamlc', cons['haml-coffee']
```

Express 2

Starting with version 0.5.0, Haml Coffee has support for Express 2 and can be registered as view engine as follows:

```
express = require 'express'

app = express.createServer()
app.register '.hamlc', require('haml-coffee')
```

Alternatively you can also use [consolidate.js](#) to register the engine:

```
express = require 'express'
cons     = require 'consolidate'

app = express.createServer()
app.register '.hamlc', cons['haml-coffee']
```

Express Usage

Layouts

Express 2 uses a layout file `layout.hamlc` by default and you have to insert the rendered view body into the layout like this:

```
!!!
%head
  %title Express App
%body
  != @body
```

Now you can create a Haml Coffee view

```
%h1= "Welcome #{ @name }"
%p You've rendered your first Haml Coffee view.
```

that you can render with:

```
app.get '/', (req, res) ->
  res.render 'index.hamlc', name: 'Express user'
```

Express 3 has removed layout support, but you can get it back by installing [express-partials](#) and configure it as middleware:

```
partials = require 'express-partials'
app.use partials()
```

Default template engine

It's possible to use Haml Coffee as the default template engine by setting the `view engine` :

```
app.configure ->
  app.set 'view engine', 'hamlc'
```

which allows you to omit the `.hamlc` extension when rendering a template:

```
app.get '/', (req, res) ->
  res.render 'index', name: 'Express user'
```

Compiler options

With Express 3, you can set global compiler options by using `app.locals` :

```
app.locals.uglify = true
```

which is the same as:

```
res.render view, { uglify: true }
```

See the **compiler options** for detailed information about all the available options.

Using the CLI tool

After the installation you will have a `haml-coffee` binary that can be used to compile single templates and even compile multiple templates recursively into a single file.

```
$ haml-coffee
Usage: node haml-coffee
```

Options:

<code>-i, --input</code>	Either a file or a directory name to be compiled
<code>-o, --output</code>	Set the output filename
<code>-n, --namespace</code>	Set a custom template namespace
<code>-t, --template</code>	Set a custom template name
<code>-b, --basename</code>	Ignore file path when generate the template name
<code>-e, --extend</code>	Extend the template scope with the context
<code>-r, --render</code>	Render to standalone HTML

The following section describes only the options that are unique to the command line tool.

You can see all the available options by executing `haml-coffee --help` and have a look at the **compiler options** for detailed information about all the options.

The `input` and `output` are optional and you can also directly redirect the streams.

Input filename

You can either specify a single template or a directory with the `-i / --input` argument. When you supply a directory, templates are being searched recursively:

```
$ haml-coffee -i template.haml
```

This will generate a template with the same name as the file but the extension changed to `.jst`. The above command for example would generate a template named `template.jst`.

A valid Haml Coffee template must have one of the following extensions: `.haml`, `.html.haml`, `.hamlc` or `.html.hamlc`.

Output filename

You can specify a single output file name to be used instead of the automatic generated output file name with the `-o / --output` argument:

```
$ haml-coffee -i template.haml -o t.js
```

This creates a template named `t.js`. You can also set a directory as input and give an output file name for concatenating all templates into a single file:

```
$ haml-coffee -i templates -o all.js
```

This will create all the templates under the `templates` directory into a single, combined output file `all.js`.

Template namespace

Each template will register itself by default under the `window.HAML` namespace, but you can change the namespace with the `-n / --namespace` argument:

```
$ haml-coffee -i template.haml -n exports.JST
```

Template name

Each template must have a unique name under which it can be addressed. By default the template name is derived from the template file name by stripping off all extensions and remove illegal characters. Directory names are converted to nested namespaces under the default namespace. For example, a template named `user/show-admin.html.haml` will result in a template that can be accessed by `window.HAML['user/show_admin']`.

Given the `-b / --basename` argument, the deduced template name will not include the path to the template. For example, a template named `user/show-admin.html.haml` will result in a template that can be accessed by `window.HAML['show_admin']` instead of `window.HAML['user/show_admin']`.

With the `-t / --template` argument you can set a template name manually:

```
$ haml-coffee -i template.haml -n exports.JST -t other
```

This will result in a template that can be accessed by `exports.JST['other']`.

Extend the template scope

By extending the template scope with the context, you can access your context data without `@` or `this`:

```
%h2= title
```

This effect is achieved by using the **with** statement. Using `with` is forbidden in ECMAScript 5 strict mode.

Stream redirection

You can use Haml Coffee on the command line to enter a template and stop it with `Ctrl-D`:

```
$ haml-coffee -p amd
%h1 Hello AMD
^D
```

which will output the AMD module source code to the console. You either have to set the placement option to `amd` or give it a template name like

```
$ haml-coffee -t name
%p JST rocks!
^D
```

which will output the JST source code. Now you can also redirect files like:

```
$ haml-coffee -t name < input.hamlc > output.jst
```

Haml support

Haml Coffee implements the **Haml Spec** to ensure some degree of compatibility to other Haml implementations and the following sections are fully compatible to Ruby Haml:

- Plain text
- Multiline: `|`
- Element names: `%`
- Attributes: `{ }` or `()`
- Class and ID: `.` and `#`, implicit `div` elements
- Self-closing tags: `/`
- Doctype: `!!!`
- HTML comments: `/`, conditional comments: `/[]`, Haml comments: `-#`
- Running CoffeeScript: `-`, inserting CoffeeScript: `=`
- CoffeeScript interpolation: `#{ }`
- Whitespace preservation: `~`
- Whitespace removal: `>` and `<`
- Escaping `\`
- Escaping HTML: `&=`, unescaping HTML: `!=`
- Filters: `:plain`, `:javascript`, `:css`, `:cdata`, `:escaped`, `:preserve`
- Boolean attributes conversion
- Haml object reference syntax: `[]`

Please consult the official **Haml reference** for more details.

Haml Coffee supports both Ruby 1.8 and Ruby 1.9 style attributes. So the following Ruby 1.8 style attribute

```
%a{ :href => 'http://haml-lang.com/', :title => 'Haml home' } Haml
```

can also be written in Ruby 1.9 style:

```
%a{ href: 'http://haml-lang.com/', title: 'Haml home' } Haml
```

HTML style tags are also supported:

```
%a( href='http://haml-lang.com/' title='Haml home') Haml
```

Helpers

Haml Coffee supports a small subset of the Ruby Haml **helpers**. The provided helpers will bind the helper function to the template context, so it isn't necessary to use `=>` .

Surround

Surrounds a block of Haml code with strings, with no whitespace in between.

```
!= surround '(', ')', ->
  %a{:href => "food"} chicken
```

produces the HTML output

```
(<a href='food'>chicken</a>)
```

Succeed

Appends a string to the end of a Haml block, with no whitespace between.

```
click
!= succeed '.', ->
  %a{:href=>"thing"} here
```

produces the HTML output

```
click
<a href='thing'>here</a>.
```

Precede

Prepends a string to the beginning of a Haml block, with no whitespace between.

```
!= precede '*', ->
  %span.small Not really
```

produces the HTML output

```
*<span class='small'>Not really</span>
```

Object reference: []

Haml Coffee supports object references, but they are implemented slightly different due to the underlying runtime and different code style for CoffeeScript.

Square brackets contain a CoffeeScript object or class that is used to set the class and id of that tag. The class is set to the object's constructor name (transformed to use underlines rather than camel case) and the id is set to the object's constructor name, followed by the value of its `id` property or its `#to_key` or `#id` functions (in that order). Additionally, the second argument (if present) will be used as a prefix for both the id and class attributes.

For example:

```
%div[@user, 'greeting']
  Hello
```

is compiled to:

```
<div class='greeting_user' id='greeting_user_15'>
  Hello!
</div>
```

If the user object is for example a Backbone model with the id of 15. If you require that the class be something other than the underscored object's constructor name, you can implement the `#hamlObjectRef` function on the object:

```
:coffeescript
class User
  id: 23
```

```
hamlObjectRef: -> 'custom'
```

```
%div[new User()]  
  Hello
```

is compiled to:

```
<div class='custom' id='custom_23'>  
  Hello!  
</div>
```

Directives

Haml Coffee supports currently a single directive that extends the Haml syntax.

Include

You can use the `+include` directive to include another template:

```
%h1 Include  
+include 'partials/test'
```

This will look up the specified template and include it. So if the partial `partials/test` contains

```
%p Partial content
```

The final result will be

```
<h1>Include</h1>  
<p>Partial content</p>
```

CoffeeScript support

Haml and CoffeeScript are a winning team, both use indention for blocks and are a perfect match for this reason. You can use CoffeeScript instead of Ruby in your Haml tags and the attributes.

It's not recommended to put too much logic into the template.

Attributes

When you define an attribute value without putting it into quotes (single or double quotes), it's considered to be CoffeeScript code to be run at render time. By default, attributes values from CoffeeScript code are escaped before inserting into the document. You can change this behaviour by setting the appropriate compiler option.

HTML style attributes are the most limited and can only assign a simple variable:

```
%img(src='/images/demo.png' width=@width height=@height alt=alt)
```

Both the `@width` and `@height` values must be passed as locals when rendering the template and `alt` must be defined before the `%img` tag.

Ruby style tags can be more complex and can call functions:

```
%header  
  %user{ :class => App.currentUser.get('status') }= App.currentUser.getDisplayName()
```

Attribute definitions are also supported in the Ruby 1.9 style:

```
%header  
  %user{ class: App.currentUser.get('status') }= App.currentUser.getDisplayName()
```

More fancy stuff can be done when use interpolation within a double quoted attribute value:

```
%header  
  %user{ class: "#{ if @user.get('roles').indexOf('admin') is -1 then 'normal' else 'admin' }" }= @user.getDisplayName()
```

But think twice about it before putting such fancy stuff into your template, there are better places like models, views or helpers to put heavy logic into.

You can define your attributes over multiple lines and the next line must not be correctly indented, so you can align them properly:

```
%input#password.hint{ type: 'password', name: 'registration[password]',  
                      data: { hint: 'Something very important', align: 'left' } }
```

In the above example you also see the usage for generating HTML5 data attributes.

Running Code

You can run any CoffeeScript code in your template:

```
- for project in @projects
- if project.visible
  .project
    %h1= project.name
    %p&= project.description
```

There are several supported types to run your code:

- Run code without insert anything into the document: `-`
- Run code and insert the result into the document: `=`

All inserted content from running code is escaped by default. You can change this behaviour by setting the appropriate compiler option.

There are three variations to run code and insert its result into the document, two of them to change the escaping style chosen in the compile option:

- Run code and do not escape the result: `!=`
- Run code and escape the result: `&=`
- Preserve whitespace when insert the result: `~`

Again, please consult the official [Haml reference](#) for more details. Haml Coffee implements the same functionality like Ruby Haml, only for CoffeeScript.

Interpolation

If you use the CoffeeScript interpolation without explicitly run code with `=` and `-`, the interpolation output is not being escaped. You can manually force escaping by using the HTML escape reference `$e`:

```
%p
  foo #{ $e '<bar>' }
```

will be rendered to

```
<p>
  foo &lt;bar>;
</p>
```

Multiline code blocks

Running code must be placed on a single line and unlike Ruby Haml, you cannot stretch a it over multiple lines by putting a comma at the end.

However, you can use multiline endings `|` to stretch your code over multiple lines to some extend:

```
- links = {           |
  home: '/',          |
  docs: '/docs',      |
  about: '/about'     |
}                      |

%ul
- for name, link of links
  %li
    %a{ href: link }= name
```

Please note, that since the line is concatenated before the compilation, you cannot omit the curly braces and the commas in the above example, like you'd do in normal CoffeeScript code.

Therefore it's recommended to use the CoffeeScript filter to have real multiline code blocks:

```
:coffeescript
  links =
    home: '/'
    docs: '/docs'
    about: '/about'

%ul
- for name, link of links
  %li
    %a{ href: link }= name
```

Functions

You can also create functions that generate Haml:

```
- sum = (a, b) ->
  %div
    %span= a
    %span= b
    %span= a+b
= sum(1,2)
= sum(3,4)
```

or pass generated HTML output through a function for post-processing.

```
= postProcess ->
  %a{ href: '/' }
```

The content of the `:coffeescript` filter is run when the template is rendered and doesn't output anything into the resulting document. This comes in handy when you have code to run over multiple lines and don't want to prefix each line with `-`:

```
%body
:coffeescript
  tags = ['CoffeeScript', 'Haml']
  project = 'Haml Coffee'
  %h2= project
  %ul
    - for tag in tags
      %li= tag
```

Compiler options

The following section describes all the available compiler options that you can use through the JavaScript API, as Express view option or as argument to the command line utility.

The command line arguments may be slightly different. For example instead of passing `--escape-html=false` you have to use the `--disable-html-escaping` argument. You can see a list of all the command line arguments by executing `haml-coffee --help`.

HTML generation options

The HTML options change the way how the generated HTML will look like.

Output format

- Name: 'format'
- Type: String
- Default: html5

The Haml parser knows different HTML formats to which a given template can be rendered and it must be one of:

- xhtml
- html4
- html5

Doctype, self-closing tags and attributes handling depends on this setting. Please consult the official [Haml reference](#) for more details.

Uglify output

- Name: uglify
- Type: Boolean
- Default: false

All generated HTML tags are properly indented by default, so the output looks nice. This can be helpful when debugging. You can skip the indentation by setting the `uglify` option to false. This save you some bytes and you'll have increased rendering speed.

HTML escape

- Name: escapeHtml
- Type: Boolean
- Default: true

The reserved HTML characters `"`, `'`, `&`, `<` and `>` are converted to their HTML entities by default when they are inserted into the HTML document from evaluated CoffeeScript.

You can always change the escaping mode within the template to either force escaping with `&=` or force unescaping with `!=`.

Attributes escape

- Name: escapeAttributes

- Type: `Boolean`
- Default: `true`

All HTML attributes that are generated by evaluating CoffeeScript are also escaped by default. You can turn off HTML escaping of the attributes only by setting `escapeAttributes` to `false`. You can't change this behaviour in the template since there is no `Haml` markup for this to instruct the compiler to change the escaping mode.

Clean CoffeeScript values

- Name: `cleanValue`
- Type: `Boolean`
- Default: `true`

Every output that is generated from evaluating CoffeeScript code is cleaned before inserting into the document. The default implementation converts `null` or `undefined` values into an empty string and marks real boolean values with a hidden marker character. The hidden marker character is necessary to distinguish between String values like `'true'`, `'false'` and real boolean values `true`, `false` in the markup, so that a boolean attribute conversion can quickly convert these values to the correct HTML5/XHTML/HTML4 representation.

Preserve whitespace tags

- Name: `preserve`
- Type: `String`
- Default: `textarea,pre`

The `preserve` option defines a list of comma separated HTML tags that are whitespace sensitive. Content from these tags must be preserved, so that the indentation has no influence on the displayed content. This is simply done by converting the newline characters to their equivalent HTML entity.

Autoclose tags

- Name: `autoclose`
- Type: `String`
- Default: `meta,img,link,br,hr,input,area,param,col,base`

The autoclose option defines a list of tag names that should be automatically closed if they have no content.

Module loader support

- Name: `placement`
- Type: `String`
- Default: `global`

The `placement` option defines where the template function is inserted upon compilation.

Possible values are:

- `global`
Inserts the optionally namespaced template function into `window.HAML`.
- `'standalone'`
Returns the template function without wrapping it
- `amd`
Wraps the template function into a `define()` statement to allow async loading via AMD.

See AMD support for more information.

Module dependencies

- Name: `dependencies`
- Type: `Object`
- Default: `{ hc: 'hamlcoffee' }`

The `dependencies` option allows you to define the modules that must be required for the AMD template `define` function. The object key will be the function parameter name of the module the object value defines. See AMD support for more information.

Data attribute hyphenation

- Name: `hyphenateDataAttrs`
- Type: `Boolean`
- Default: `true`

Convert underscores to hyphens for data attribute keys, see [the Ruby Haml reference](#).

Custom helper function options

Haml Coffee provides helper functions for HTML escaping, value cleaning and whitespace preservation, which must be available at render time. By default every generated template function is self-contained and includes all of the helper functions.

However you can change the reference to each helper function by providing the appropriate compiler option and there are good reasons to do so:

- You want to reduce the template size and provide all the helpers from a central place.

- You want to customize a helper function to better fit your needs.

To change these functions, simply assign the new function name to one of the following options:

- `customHtmlEscape` : Escape the reserved HTML characters into their equivalent HTML entity.
- `customPreserve` : Converting newlines into their HTML entity.
- `customFindAndPreserve` : Find whitespace sensitive tags and preserve their content.
- `customCleanValue` : Clean the value that is returned after evaluating some inline CoffeeScript.
- `customSurround` : Surrounds a block of Haml code with strings, with no whitespace in between.
- `customSucceed` : Appends a string to the end of a Haml block, with no whitespace between.
- `customPrecede` : Prepends a string to the beginning of a Haml block, with no whitespace between.
- `customReference` : Creates the Haml object reference.

The `customSurround`, `customSucceed` and `customPrecede` are bound to the template context.

You can find a default implementation for all these helper functions in [Haml Coffee Assets](#).

AMD support

- Global dependencies
- Trivial dependency detection

Haml Coffee has built in AMD support by setting the `placement` option to `amd`. This will generate a module definition for the JavaScript template. The `dependencies` options can be used to provide a mapping of module names to parameters. To illustrate this, the default value will result in the following module declaration:

```
define ['hamlcoffee'], (hc) ->
```

When the template contains a `require` call in the form of

```
- require 'module'
- require 'deep/nested/other'
```

it will be added to the module definition list

```
define ['hamlcoffee', 'module', 'deep/nested/other'], (hc, module, other) ->
```

allowing you to render a partial template:

```
!= module()
!= other()
```

Of course the `require` call can have different quotes or parentheses, allowing you to directly require and render:

```
!= require("another/other")()
```

Module dependency

By default Haml Coffee AMD templates depend on the `hamlcoffee` module that provides the client side helpers needed to render the template. You need to supply your own module, but you can grab a copy from [Haml Coffee Assets AMD helpers](#) and adapt it to your needs. Another option is to remove the module from the **module dependencies**, but that's usually not what you want because it duplicates the needed function within every template.

Static HTML

By default Haml Coffee outputs pre-compiled templates for rendering dynamic content.

There is a `render` helper that can be used as follows to get the standalone HTML rendering of a HAML template.

```
hamlc = require 'haml-coffee'
text = hamlc.render '%h1= @title', {title: 'hi'}
text
'<h1>hi</h1>'
```

Development information

Haml Coffee uses **Grunt** for development, which you can install with **NPM**:

```
$ npm install
```

and run Grunt to automatically run the Jasmine specs on file modification:

Changelog

Feel free to take a look at the crispy **changelog** instead of crawling through the commit history.

Related projects

Haml Coffee in the Rails asset pipeline:

- [haml-coffee-assets](#)

Authors

- [Michael Kessler \(@netzpirat, FlinkFinger\)](#)
- [Sebastion Deutsch \(@sippndipp, 9elements\)](#)
- [Jan Varwig \(@agento, 9elements\)](#)

Contributors

See all contributors on [the contributor page](#).

License

(The MIT License)

Copyright (c) 2011 9elements, 2011-2013 Michael Kessler

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the 'Software'), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED 'AS IS', WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Keywords

none

Install

```
> npm i haml-coffee
```

Repository

[github.com/netzpirat/haml-coffee](#)

Homepage

[github.com/netzpirat/haml-coffee#readme](#)

Weekly Downloads



Version	License
1.14.1	none
Issues	Pull Requests
6	6

Last publish
9 years ago

Collaborators



[👉 Try on RunKit](#)

[🚩 Report malware](#)



Support

[Help](#)

[Advisories](#)

[Status](#)

[Contact npm](#)

Company

[About](#)

[Blog](#)

[Press](#)

Terms & Policies

[Policies](#)

[Terms of Use](#)

[Code of Conduct](#)

[Privacy](#)