Views / Templates

Each template language is exposed via its own rendering method. These methods simply return a string:

```
get '/' do
  erb :index
end
```

This renders views/index.erb.

Instead of a template name, you can also just pass in the template content directly:

```
get '/' do
  code = "<%= Time.now %>"
  erb code
end
```

Templates take a second argument, the options hash:

```
get '/' do
  erb :index, :layout => :post
end
```

This will render views/index.erb embedded in the views/post.erb (default is views/layout.erb, if it exists).

Any options not understood by Sinatra will be passed on to the template engine:

```
get '/' do
  haml :index, :format => :html5
end
```

You can also set options per template language in general:

```
set :haml, :format => :html5

get '/' do
  haml :index
end
```

Options passed to the render method override options set via set.

Available Options:

locals
: List of locals passed to the document. Handy with partials. Example: `erb "<%= foo="" %="">", :locals => {:foo => "bar"}`

default_encoding
: String encoding to use if uncertain. Defaults to `settings.default_encoding`.

views
: Views folder to load templates from. Defaults to `settings.views`.

layout
: Whether to use a layout (`true` or `false`). If it's a Symbol, specifies what template to use. Example: `erb :index, :layout => !request.xhr?`

content_type
: Content-Type the template produces. Default depends on template language.

scope
: Scope to render template under. Defaults to the application instance. If you change this, instance variables and helper methods will not be available.

layout_engine
: Template engine to use for rendering the layout. Useful for languages that do not support layouts otherwise. Defaults to the engine used for the template. Example: `set :rdoc, :layout_engine => :erb`

layout_options
: Special options only used for rendering the layout. Example: `set :rdoc, :layout_options => { :views => 'views/layouts' }`

Templates are assumed to be located directly under the ./views directory. To use a different views directory:

```
set :views, settings.root + '/templates'
```

One important thing to remember is that you always have to reference templates with symbols, even if they're in a subdirectory (in this case, use: :'subdir/template' or 'subdir/template'.to_sym). You must use a symbol because otherwise rendering methods will render any strings passed to them directly.

Literal Templates

```
get '/' do
  haml '%div.title Hello World'
end
```

Renders the template string.

Available Template Languages

Some languages have multiple implementations. To specify what implementation to use (and to be thread-safe), you should simply require it first:

```
require 'rdiscount' # or require 'bluecloth'
get('/') { markdown :index }
```

Haml Templates

Dependency	haml
File Extension	`.haml`
Example	`haml :index, :format => :html5`

Erb Templates

Dependency	erubis or erb (included in Ruby)
File Extensions	`.erb`, `.rhtml` or `.erubis` (Erubis only)
Example	`erb :index`

Builder Templates

Dependency	builder
File Extension	`.builder`
Example	`builder { |xml| xml.em "hi" }`

It also takes a block for inline templates (see example).

Nokogiri Templates

Dependency	nokogiri
File Extension	`.nokogiri`
Example	`nokogiri { |xml| xml.em "hi" }`

It also takes a block for inline templates (see example).

Sass Templates

Dependency	sass
File Extension	`.sass`
Example	`sass :stylesheet, :style => :expanded`

SCSS Templates

Dependency	sass
File Extension	`.scss`
Example	`scss :stylesheet, :style => :expanded`

Less Templates

Dependency	less
File Extension	`.less`
Example	`less :stylesheet`

Liquid Templates

Dependency	liquid
File Extension	`.liquid`
Example	`liquid :index, :locals => { :key => 'value' }`

Since you cannot call Ruby methods (except for yield) from a Liquid template, you almost always want to pass locals to it.

Markdown Templates

Dependency	Anyone of: RDiscount, RedCarpet, BlueCloth, kramdown, maruku
File Extensions	`.markdown`, `.mkd` and `.md`
Example	`markdown :index, :layout_engine => :erb`

It is not possible to call methods from markdown, nor to pass locals to it. You therefore will usually use it in combination with another rendering engine:

```
erb :overview, :locals => { :text => markdown(:introduction) }
```

Note that you may also call the markdown method from within other templates:

```
%h1 Hello From Haml!
%p= markdown(:greetings)
```

Since you cannot call Ruby from Markdown, you cannot use layouts written in Markdown. However, it is possible to use another rendering engine for the template than for the layout by passing the :layout_engine option.

Textile Templates

Dependency	RedCloth
File Extension	`.textile`
Example	`textile :index, :layout_engine => :erb`

It is not possible to call methods from textile, nor to pass locals to it. You therefore will usually use it in

combination with another rendering engine:

```
erb :overview, :locals => { :text => textile(:introduction) }
```

Note that you may also call the textile method from within other templates:

```
%h1 Hello From Haml!
%p= textile(:greetings)
```

Since you cannot call Ruby from Textile, you cannot use layouts written in Textile. However, it is possible to use another rendering engine for the template than for the layout by passing the :layout_engine option.

RDoc Templates

Dependency	RDoc
File Extension	`.rdoc`
Example	`rdoc :README, :layout_engine => :erb`

It is not possible to call methods from rdoc, nor to pass locals to it. You therefore will usually use it in combination with another rendering engine:

```
erb :overview, :locals => { :text => rdoc(:introduction) }
```

Note that you may also call the rdoc method from within other templates:

```
%h1 Hello From Haml!
%p= rdoc(:greetings)
```

Since you cannot call Ruby from RDoc, you cannot use layouts written in RDoc. However, it is possible to use another rendering engine for the template than for the layout by passing the :layout_engine option.

AsciiDoc Templates

Dependency	Asciidoctor
File Extension	`.asciidoc`, `.adoc` and `.ad`
Example	`asciidoc :README, :layout_engine => :erb`

Since you cannot call Ruby methods directly from an AsciiDoc template, you almost always want to pass locals to it.

Radius Templates

Dependency	Radius

File Extension	`.radius`
Example	`radius :index, :locals => { :key => 'value' }`

Since you cannot call Ruby methods directly from a Radius template, you almost always want to pass locals to it.

Markaby Templates

Dependency	Markaby
File Extension	`.mab`
Example	`markaby { h1 "Welcome!" }`

It also takes a block for inline templates (see example).

RABL Templates

Dependency	Rabl
File Extension	`.rabl`
Example	`rabl :index`

Slim Templates

Dependency	Slim Lang
File Extension	`.slim`
Example	`slim :index`

Creole Templates

Dependency	Creole
File Extension	`.creole`
Example	`creole :wiki, :layout_engine => :erb`

It is not possible to call methods from creole, nor to pass locals to it. You therefore will usually use it in combination with another rendering engine:

```
erb :overview, :locals => { :text => creole(:introduction) }
```

Note that you may also call the creole method from within other templates:

```
%h1 Hello From Haml!
%p= creole(:greetings)
```

Since you cannot call Ruby from Creole, you cannot use layouts written in Creole. However, it is possible to use another rendering engine for the template than for the layout by passing the :layout_engine option.

MediaWiki Templates

Dependency	WikiCloth
File Extension	`.mediawiki` and `.mw`
Example	`mediawiki :wiki, :layout_engine => :erb`

It is not possible to call methods from MediaWiki markup, nor to pass locals to it. You therefore will usually use it in combination with another rendering engine:

```
erb :overview, :locals => { :text => mediawiki(:introduction) }
```

Note that you may also call the mediawiki method from within other templates:

```
%h1 Hello From Haml!
%p= mediawiki(:greetings)
```

Since you cannot call Ruby from MediaWiki, you cannot use layouts written in MediaWiki. However, it is possible to use another rendering engine for the template than for the layout by passing the :layout_engine option.

CoffeeScript Templates

Dependency	CoffeeScript and a way to execute javascript
File Extension	`.coffee`
Example	`coffee :index`

Stylus Templates

Dependency	Stylus and a way to execute javascript
File Extension	`.styl`
Example	`stylus :index`

Before being able to use Stylus templates, you need to load stylus and stylus/tilt first:

```
require 'sinatra'
require 'stylus'
require 'stylus/tilt'

get '/' do
  stylus :example
```

```
end
```

Yajl Templates

Dependency	yajl-ruby
File Extension	`.yajl`
Example	`yajl :index, :locals => { :key => 'qux' }, :callback => 'present', :variable => 'resource'`

The template source is evaluated as a Ruby string, and the resulting json variable is converted using #to_json:

```
json = { :foo => 'bar' }
json[:baz] = key
```

The :callback and :variable options can be used to decorate the rendered object:

```
var resource = {"foo":"bar","baz":"qux"};
present(resource);
```

WLang Templates

Dependency	WLang
File Extension	`.wlang`
Example	`wlang :index, :locals => { :key => 'value' }`

Since calling ruby methods is not idiomatic in WLang, you almost always want to pass locals to it. Layouts written in WLang and yield are supported, though.

Accessing Variables in Templates

Templates are evaluated within the same context as route handlers. Instance variables set in route handlers are directly accessible by templates:

```
get '/:id' do
  @foo = Foo.find(params['id'])
  haml '%h1= @foo.name'
end
```

Or, specify an explicit Hash of local variables:

```
get '/:id' do
  foo = Foo.find(params['id'])
```

```
  haml '%h1= bar.name', :locals => { :bar => foo }
end
```

This is typically used when rendering templates as partials from within other templates.

Templates with yield and nested layouts

A layout is usually just a template that calls yield. Such a template can be used either through the :template option as described above, or it can be rendered with a block as follows:

```
erb :post, :layout => false do
  erb :index
end
```

This code is mostly equivalent to erb :index, :layout => :post.

Passing blocks to rendering methods is most useful for creating nested layouts:

```
erb :main_layout, :layout => false do
  erb :admin_layout do
    erb :user
  end
end
```

This can also be done in fewer lines of code with:

```
erb :admin_layout, :layout => :main_layout do
  erb :user
end
```

Currently, the following rendering methods accept a block: erb, haml, liquid, slim, wlang. Also the general render method accepts a block.

Named Templates

Templates may also be defined using the top-level template method:

```
template :layout do
  "%html\n  =yield\n"
end

template :index do
  '%div.title Hello World!'
end

get '/' do
```

```
  haml :index
end
```

If a template named "layout" exists, it will be used each time a template is rendered. You can individually disable layouts by passing :layout => false or disable them by default via set :haml, :layout => false:

```
get '/' do
  haml :index, :layout => !request.xhr?
end
```

Associating File Extensions

To associate a file extension with a template engine, use Tilt.register. For instance, if you like to use the file extension tt for Textile templates, you can do the following:

```
Tilt.register :tt, Tilt[:textile]
```

Adding Your Own Template Engine

First, register your engine with Tilt, then create a rendering method:

```
Tilt.register :myat, MyAwesomeTemplateEngine

helpers do
  def myat(*args) render(:myat, *args) end
end

get '/' do
  myat :index
end
```

Renders ./views/index.myat. See https://github.com/rtomayko/tilt to learn more about Tilt.

Using Custom Logic for Template Lookup

To implement your own template lookup mechanism you can write your own #find_template method:

```
configure do
  set :views [ './views/a', './views/b' ]
end

def find_template(views, name, engine, &block)
  Array(views).each do |v|
    super(v, name, engine, &block)
  end
end
```

Inline Templates

Templates may be defined at the end of the source file:

```
require 'sinatra'

get '/' do
  haml :index
end

__END__

@@ layout
%html
  = yield

@@ index
%div.title Hello world.
```

NOTE: Inline templates defined in the source file that requires sinatra are automatically loaded. Call enable :inline_templates explicitly if you have inline templates in other source files.