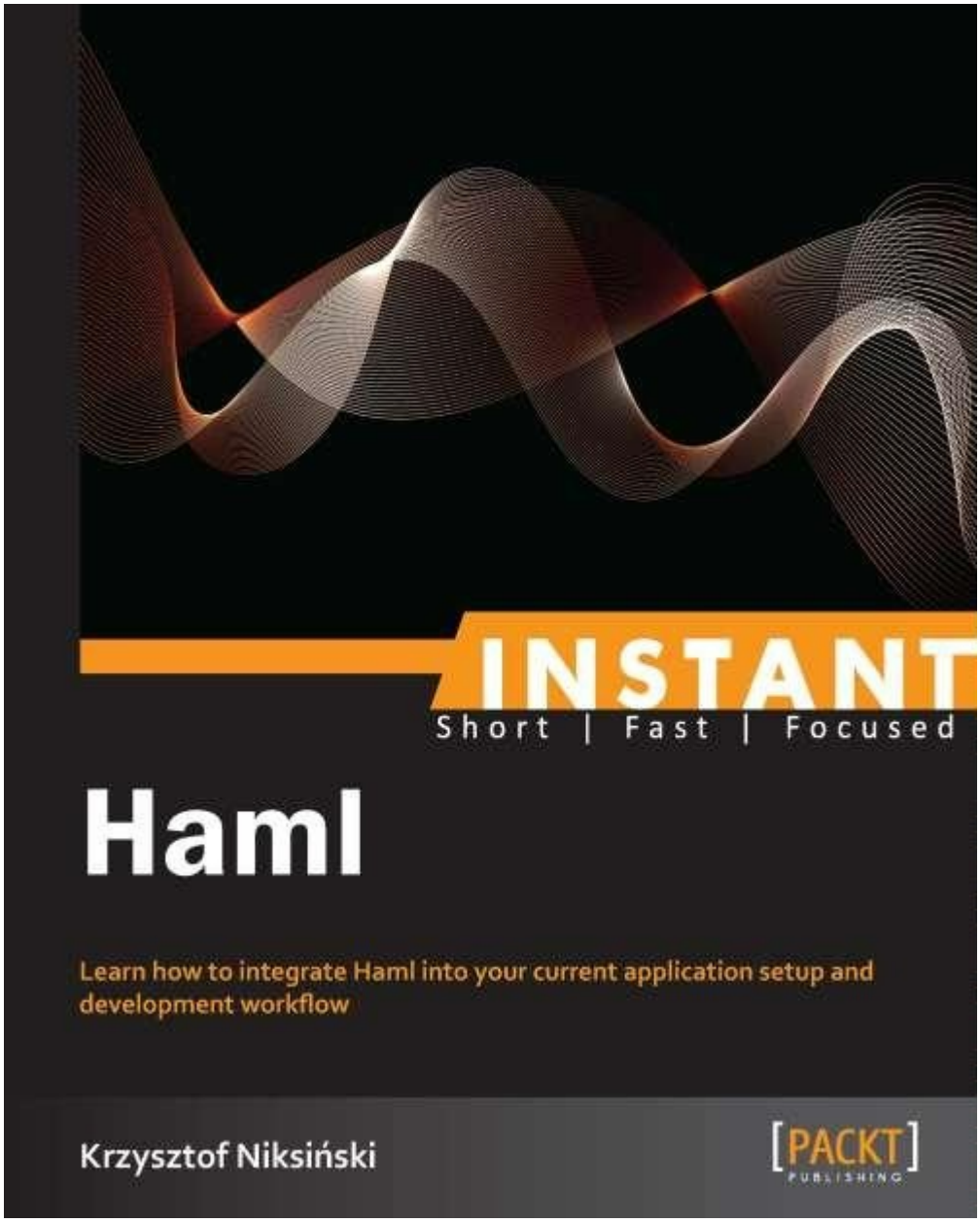

INSTANT
Short | Fast | Focused
Haml
Learn how to integrate Haml into your current application setup and development workflow
Krzysztof Niksiński
[PACKT]
PUBLISHING

Table of Contents

Instant Haml

Instant Haml

First published: September 2013

Production Reference: 1190913

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78328-377-4

www.packtpub.com

Credits

Author

Krzysztof Niksiński

Reviewer

Satish Talim

Acquisition Editor

Rubal Kaur

Commissioning Editor

Llewellyn Rozario

Technical Editors

Veena Pagare

Jinesh Kampani

Project Coordinator

Romal Karani

Proofreader

Lesley Harrison

Production Coordinator

Shantanu Zagade

Cover Work

Shantanu Zagade

Cover Image

Valentina Dsilva

About the Author

Krzysztof Niksiński is a Ruby On Rails developer from Warsaw, Poland. He has been a Ruby On Rails freelancer since 2003 and regrets that he started using HAML only four years ago in his projects, because it would have saved his lot of time and efforts. He loves to create web applications, which use multiple technologies in synergy, as well as configure all the underlying layers of the application servers, databases, and operating systems. Playing with all the systems that make Internet work is his daily bread and butter. He grew up in Warsaw, where he attended the Warsaw University of Technology. Later, he started to work as an Internet Systems administrator in a bank, where he learned all the details of Internet services and protocols. He later fell in love with Ruby and started his own Ruby On Rails development company. At the time of this writing he also works as a Middleware Infrastructure Administrator for Acxiom, a marketing technology and services company. Krzysztof spends his free time creating electronic music, playing basketball with his friends, or training his dog, Goofy.

I'd like to thank my awesome girlfriend Ania for her support in helping me to make this book possible.

About the Reviewer

Satish Talim is a Senior Software Consultant based in Pune, India with over 35 + years of IT experience. Ruby has become his language of choice since 2005. He regularly teaches Ruby programming, JRuby, Sinatra, and other languages at RubyLearning.org. In 2008, he was declared as Ruby's Top Teacher by RubyInside and also won the Shorty Award for Education. He now works on Open Source projects at JoshSoftware Pvt. Ltd., Pune.

www.packtpub.com

Support files, eBooks, discount offers and more

You might want to visit www.packtpub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packtpub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <service@packtpub.com> for more details.

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

Free Access for Packt account holders

If you have an account with Packt at www.packtpub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Chapter 1. Instant Haml

Welcome to the *Instant Haml*. This book has been especially created to provide you with all the information that you need to get started with Haml and boost your Ruby on Rails application development speed and efficiency. You will learn the basics of Haml, get started with creating your first Haml templates, and discover some tips and tricks for using Haml. You will also learn how to convert your existing Ruby on Rails application to use Haml. To make most of this book, you should have some experience with Ruby on Rails framework.

This book contains the following sections:

So, what is Haml? helps you find out what Haml actually is, what you can do with it, and why it's so great.

Installation helps you learn how to download and install Haml with the minimum fuss and then set it up so that you can use it as soon as possible.

Quick start – Using Haml will show you how to perform one of the core tasks of Haml; creating and converting view templates of an example Ruby on Rails application. Follow the steps to convert existing templates to Haml, learn how to create a template from scratch, create tags, add attributes to tags, implement nesting using whitespace, integrate Haml with your current programming editor, switch your Ruby on Rails application to use Haml in all view generators and make Haml play nicely with the Twitter Bootstrap framework.

Top 6 features you need to know about helps you learn how to perform eight tasks with the most important features of Haml. By the end of this section you will be able to generate HTML elements and attributes, set HTML doctypes, add comments, use Haml filters, use Ruby evaluation and Ruby blocks in Haml, generate HTML plain text using Haml, use multiline strings, preserve whitespace and use Haml helpers. You will also learn how to change the Haml configuration options.

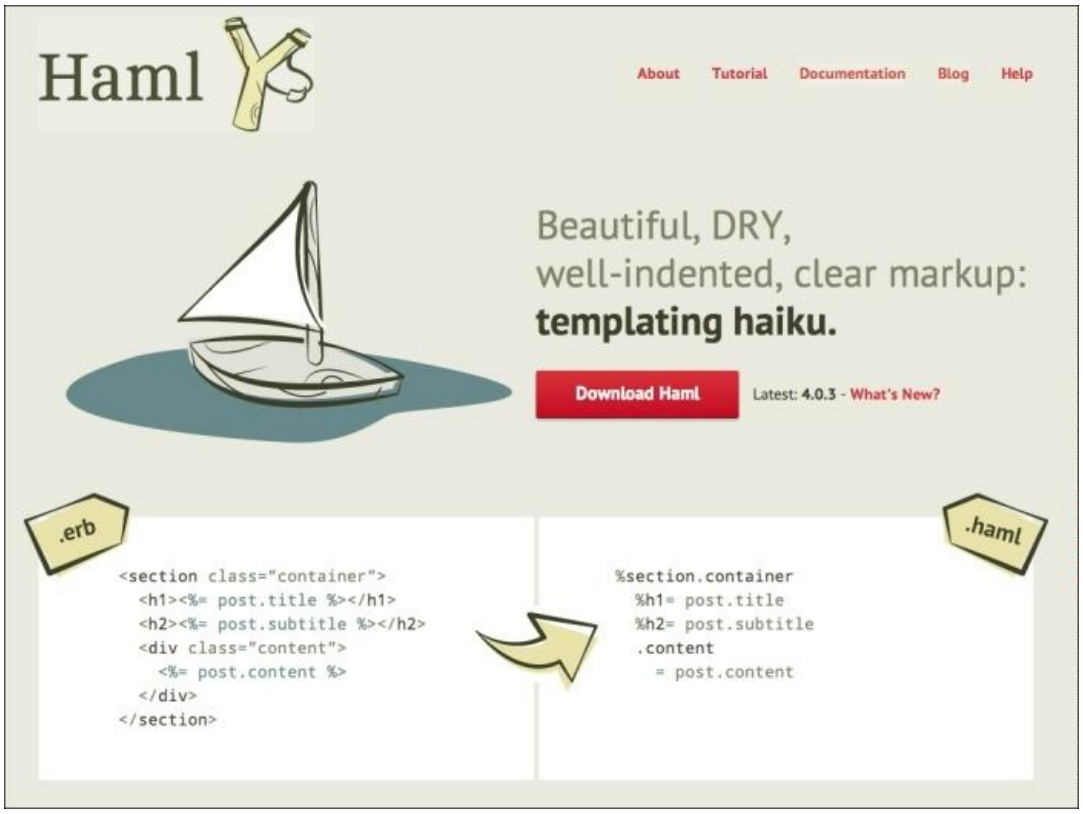
People and places you should get to know discusses the Haml technology, which is still being updated. This section provides you with many useful links to the project page and forums, as well as a number of helpful articles, tutorials, blogs and Twitter feeds that will help you stay current and learn more about Haml.

So, what is Haml?

Haml is **HTML abstraction markup language**. In other words, it's a language you can use instead of HTML to create HTML files. It's designed

to make markup simple, easy to write, clean, readable, and adhering to the DRY principles.

Haml is easy to learn and is well documented. It saves your time because there are 30-40 percent less characters in Haml code than in ERB and you don't have to write closing tags. This makes it incredibly easy to nest elements and declare the `div` tags. It emphasizes using semantic tags because there is so much less markup. The difference between ERB and Haml is illustrated in the following screenshot:



Haml has been created by *Hampton Catlin* in May 2006, and then maintained and developed by *Nathan Weizenbaum* for many years. The official implementation has been built for Ruby with plugins for Ruby on Rails. There are also implementations for other languages (Python, PHP, Perl, Java, and more).

It is most widely used to replace the built-in templating engine (ERB) of Ruby on Rails, but it can only be used with Ruby (to generate static sites) or with the Sinatra framework.

You can integrate it with your existing Ruby on Rails application easily, as ERB and Haml view templates can co-exist. You can then switch the Rails templating engine of your application to Haml, convert all the existing ERB files to Haml, or convert individual files manually.

It is best experienced by taking it for a *test drive* to see if it fits your coding style. Check it out and if you like it use it because of that, not because of the hype.

Installation

The following steps will show you how to install Haml and get it set up for use in your coding environment.

Step 1 – what do I need?

Before starting, please check the requirements listed as follows:

- Ruby (the updated version, for examples in this book, we are going to use Version 1.9.3-p392).
- Rubygems (the updated version, we are going to use 2.0.3).
- Bundler gem, to install all the required gem dependencies.
- Rails framework (for Ruby on Rails integration).
- Make sure you install the latest version of a modern browser (Firefox, Internet Explorer, Chrome, Safari, or Opera). The author suggests Google Chrome (Version 27.0.1453.116 has been used for examples in this book).
- An editor of your choice to edit text files or a programming editor. Haml is supported out of the box in many editors and can be added as a package/plugin in many others (Sublime Text 2 is used in this book).

> **Note**
>
> For a quick setup, you can use RailsInstaller, which is available from http://railsinstaller.org. It will set up Ruby and Rails for you.

Step 2 – downloading and installing Haml

The simplest way to try Haml is just to install the Haml gem using the following gem command:

```
gem install haml
```

Step 3 – converting an Haml file to an HTML file using the command line Haml utility to verify that it works

Let's test if our Haml installation is working correctly. We will create a simple Haml file and convert it to an HTML file using the command-line

`haml` utility. Use the following code to create the test Haml file (you can copy/paste it to your editor):

```
!!!
%html
  %head
    %body
      %h1 Hello world!
      %p First paragraph in Haml
      .test-div This is how you create a div with
a class
```

OSX/Linux

Perform the following steps to convert an Haml file to an HTML file for OSX/Linux systems:

1. Open your favorite editor.
2. Create a new file and paste the previous code.
3. Save the file and name it `test.haml`.
4. Open a terminal and using the `cd` command, change the current working directory to the directory that contains the file you just created.
5. Use the following command to convert the file:

```
haml test.haml test.html
```

Windows

Perform the following steps to convert an Haml file to an HTML file for Windows systems:

1. Open your favorite editor.
2. Create a new file and paste the previous code.
3. Save the file, name it `test.haml`.
4. Open a command prompt window (`cmd.exe`) and using the `cd` command, change the current working directory to the directory that contains the file you just created.
5. Use the following command to convert the file:

```
haml test.haml test.html
```

This should create the `test.html` file with following contents:

```
<!DOCTYPE html>
<html>
```

```
  <head>
    <body>
      <h1>Hello world!</h1>
      <p>First paragraph in Haml</p>
      <div class='test-div'>This is how you
create a div with a class</div>
    </body>
  </head>
</html>
```

We have used just a basic `haml` command syntax, which is `haml [input file] [output` file], what it does is convert the input haml file to an HTML file. If you omit the output file parameter, the command will output the HTML to `stdout`. If you want to learn more about all the options of the `haml` command use `haml -h` to see the help.

And that's it

By this point, you have a working installation of Haml and are able to convert the Haml files to HTML using the command-line utility.

Assuming that you have successfully set up Haml, it's time to learn how to create the basic parts of an HTML file in an Haml file.

Quick start – using Haml

Step 1 – integrating with Rails and creating a simple view file

Let's create a simple rails application and change one of the view files to Haml from ERB:

1. Create a new rails application named `blog`:

```
rails new blog
```

2. Add the Haml gem to this application's `Gemfile` and run it:

```
bundle install
```

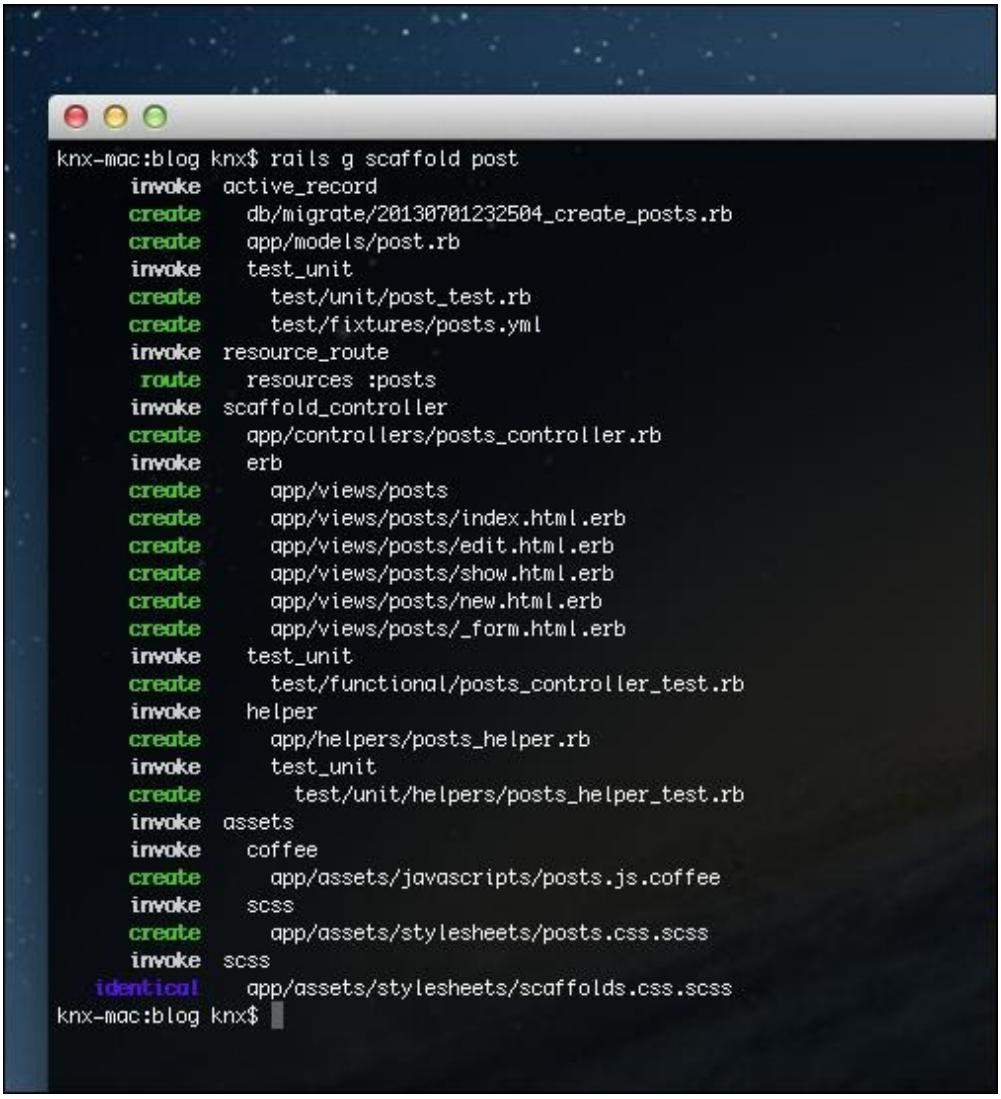
3. When the gem has been added, generate some views that we can convert to Haml and learn about its basic features. Run the Rails scaffold generator to create a scaffold named `post`.

```
rails g scaffold post
```

4. After this, you need to run the migrations to create the database tables:

```
rake db:migrate
```

   You should get an output as shown in the following screenshot:



Note

Our application is not yet generating Haml views automatically. We will switch it to this mode in the next steps.

The `index.html.erb` file that has been generated and is located in `app/views/posts/index.html.erb` looks as follows:

```
<h1>Listing posts</h1>

<table>
  <tr>
    <th></th>
    <th></th>
    <th></th>
  </tr>
```

```
<% @posts.each do |post| %>
  <tr>
    <td><%= link_to 'Show', post %></td>
    <td><%= link_to 'Edit', edit_post_path(post)
%></td>
    <td><%= link_to 'Destroy', post,
method: :delete, data: { confirm: 'Are you
sure?' } %></td>
  </tr>
<% end %>
</table>

<br>

<%= link_to 'New Post', new_post_path %>
```

Let's convert it to an Haml view step-by-step. First, let's understand the basic features of Haml:

- Any HTML tags are written with a percent sign and then the name of the tag
- Whitespace (tabs) is being used to create nested tags
- Any part of an Haml line that is not interpreted as something else is taken to be plain text
- Closing tags as well as end statements are omitted for Ruby blocks

Knowing the previous features we can write the first lines of our example view in Haml. Open the `index.html.erb` file in an editor and replace `<h1>`, `<table>`, `<th>`, and `<tr>` as follows:

- `<h1>Listing posts</h1>` can be written as `%h1 Listing posts`
- `<table>` can be written as `%table`
- `<tr>` becomes `%tr`
- `<th>` becomes `%th`

After those first replacements our view file should look like:

```
%h1 Listing posts
%table
  %tr
    %th
    %th
    %th

<% @posts.each do |post| %>
  <tr>
    <td><%= link_to 'Show', post %></td>
    <td><%= link_to 'Edit', edit_post_path(post)
%></td>
```

```
    <td><%= link_to 'Destroy', post,
method: :delete, data: { confirm: 'Are you
sure?' } %></td>
  </tr>
<% end %>

<br>

<%= link_to 'New Post', new_post_path %>
```

Please notice how `%tr` is nested within the `%table` tag using a tab and also how `%th` is nested within `%tr` using a tab.

Next, let's convert the Ruby parts of this view. Ruby is evaluated and its output is inserted into the view when using the equals character. In ERB we had to use `<%=`, whereas in Haml, this is shortened to just a `=`. The following examples illustrate this:

- `<%= link_to 'Show', post %>` becomes `= link_to 'Show', post` and all the other `<%=` parts are changed accordingly
- The equals sign can be used at the end of the tag to insert the Ruby code within that tag
- Empty (void) tags, such as `<br>`, are created by adding a forward slash at the end of the tag

Please note that you have to leave a space after the equals sign. After the changes are incorporated, our view file will look like:

```
%h1 Listing posts
%table
  %tr
    %th
    %th
    %th
<% @posts.each do |post| %>
  %tr
    %td= link_to 'Show',post
    %td= link_to 'Edit',edit_post_path(post)
    %td= link_to
'Destroy',post,method: :delete,data: { confirm:
'Are you sure?' }
%br/
= link_to 'New Post',new_post_path
```

The only thing left to do now is to convert the Ruby block part: `<% @posts.each do |post| %>`. Code that needs to be run, but does not generate any output, is written using a hyphen character. Here is how this conversion works:

- Ruby blocks do not need to be closed, they end when the indentation decreases
- HTML tags and the Ruby code that is nested within the block are indented by one tab more than the block
- `<% @posts.each do |post| %>` becomes `-@posts.each do |post|`
- Remember about a space after the hyphen character

After we replace the remaining part in our view file according to the previous rules, it should look as follows:

```
%h1 Listing posts
%table
  %tr
    %th
    %th
    %th
- @posts.each do |post|
  %tr
    %td= link_to 'Show', post
    %td= link_to 'Edit', edit_post_path(post)
    %td= link_to 'Destroy', post,
method: :delete, data: { confirm: 'Are you
sure?' }
%br/
= link_to 'New Post', new_post_path
```

Save the view file and change its name to `index.html.haml`. This is now an Haml-based template. Start our example Rails application and visit `http://localhost:3000/posts` to see the view being rendered by Rails, as shown in the following screenshot:

Step 2 – switching Rails application to use Haml as the templating engine

In the previous step, we have enabled Haml in the test application. However, if you generate new view files using any of the Rails built-in generators, it will still use ERB.

Let's switch the application to use Haml as the templating engine.

Edit the blog application `Gemfile` and add a gem named `haml-rails` to it. You can add it to the `:development` group because the generators are only used during development and this functionality is not needed in production or test environments.

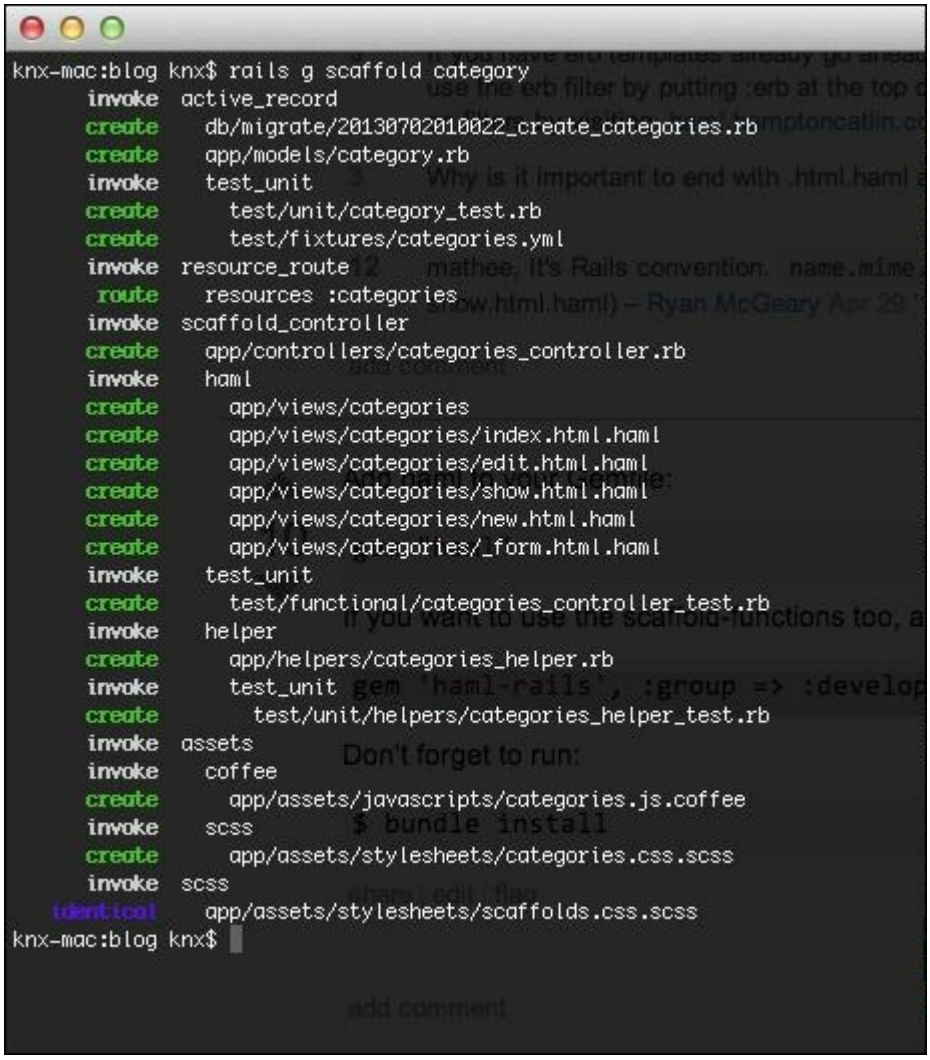
Our application `Gemfile` now looks as shown in the following code:

```
source 'https://rubygems.org'

gem 'rails', '3.2.13'

gem 'sqlite3'
gem 'haml'
gem 'haml-rails', :group => :development

group :assets do
  gem 'sass-rails',   '~> 3.2.3'
  gem 'coffee-rails', '~> 3.2.1'
  gem 'uglifier', '>= 1.0.3'
end

gem 'jquery-rails'
```

Then run following bundle command to install the gem:

```
bundle install
```

Let's say the posts in our application need to have categories. Run the scaffold generator to create some views for categories. This generator will create views using Haml, as shown in the following screenshot:



Please note that new views have a `.html.haml` extension and are using Haml. For example, the `_form.html.haml` view for the form looks as follows:

```
= form_for @category do |f|Asjd12As
  - if @category.errors.any?
    #error_explanation
      %h2= "#{pluralize(@category.errors.count,
"error")} prohibited this category from being
saved:"
      %ul
        - @category.errors.full_messages.each do
|msg|
          %li= msg
  .actions
    = f.submit 'Save'
```

Note

There are two very useful shorthand notations for creating a `<div>` tag with a class or a `<div>` tag with an ID.

To create a div with an ID, use the hash symbol followed by the name of the ID. For example, `#error_explanation` will result in `<div id="error_explanation">`.

To create a `<div>` tag with a class attribute, use a dot followed by the name of the class. For example, `.actions` will create `<div class="actions">`.

Step 3 – converting existing view templates to Haml

Our example blog app still has some leftover templates which are using ERB as well as an `application.html.erb` layout file. We would like to convert those to Haml. There is no need to do it all individually, because there is a handy gem which will automatically convert all the ERB files to Haml, shown as follows:

1. Let's install the `html2haml` gem:

```
gem install html2haml
```

2. Using the `cd` command, change the current working directory to the `app` directory of our example application and run the following bash command to convert all the ERB files to Haml (to run this command you need a bash shell. On Windows, you can use the embedded bash shell which ships with GitHub for Windows, Cygwin bash, MinGW bash, or the MSYS bash shell which is bundled with Git for Windows).

```
for file in $(find . -type f -name
\*.html.erb); do
  html2haml -e ${file} "$(dirname
${file})/$(basename ${file} .erb).haml";
done
```

3. Then to remove the ERB files and run this command:

```
find ./ -name *erb | while read line; do rm
$line; done
```

4. Those two Bash snippets will first convert all the ERB files recursively in the `app` directory of our application and then remove the remaining ERB view templates.

Step 4 – using attributes

The HTML tag attributes are written using Ruby's hash syntax. All the Ruby logic that works in Ruby's hashes, will be evaluated. You can even insert local variables. This hash is placed just after the tag name. For example, consider the following hash syntax:

```
%script{:type => "text/javascript", :src  =>
"javascripts/script_#{2 + 7}"}
```

Will be rendered as:

```
<script src='javascripts/script_9'
type='text/javascript'></script>
```

The `:class` and `:id` attributes can be specified as an array. Elements will be joined using a space character for `:class` and an underscore for `:id`.

```
%li{:id => [@item.name, @item.id], :class =>
[@item.type, @item.category]}
```

The Previous cose is same as the following:

```
%li{:id => #{@item.name}_#{@item.id}, :class =>
#{@item.type}_#{@item.category}
```

Assuming that `@item.name = "book"`, `@item.id = 2837773`, `@item.type ="literature"`, and `@item.category = "books"`, the previous code may be rendered as:

```
<li id="book_2837773" class="literature
books"></li>
```

Boolean attributes (the ones which have no value and matter only for whether present or not present) such as `checked` and `selected` are written using a Boolean value as follows:

```
%input{:selected => true}
```

The previous code can be rendered as:

```
<input selected>
```

When we use `false` as the value, the `<input>` tag will be rendered without the `selected` attribute, that is, as `<input>`.

HTML5 custom `data` attributes can be written using a hash as a value of the `:data` attribute. Each of the key/value pair will render a data attribute with the key as the name. For example, consider the following code:

```
<li class="user" data-name="John"
data-city="Boston"
  data-lang="js" data-food="Bacon">
```

The previous code can be written as:

```
%li{:class =>
"user", :data=>{:name=>"John", :city=>"Boston", :lang=>"js", :food=>"Baco
```

Because the `<div>` elements are used often, they are the default elements. As mentioned before, there is a shorthand notation for them.

> **Note**
>
> A `#` item will be compiled as `<div id="item"></div>`.
>
> `.item` will compile to `<div class"item"></div>`.
>
> Class names can be joined with a dot.
>
> `.item.btn.btn-small` will compile to `<div class="item btn btn-small"></div>`.

Step 5 – integrating Haml with your current programming editor

Some editors offer support for Haml out of the box and in others it is required to install a separate plugin/package/bundle to use it effectively. Let's review how to enable Haml in the most popular editors.

Sublime Text 2

Sublime Text 2 is one of the most popular programming editors right now. It can be downloaded and evaluated for free, however, if you plan to use it

continuously, you will have to purchase a license (at the time of writing the license cost was 70 dollars).

In Sublime Text 2, we can use an adapted TextMate bundle. It is available on GitHub (https://github.com/phuibonhoa/handcrafted-haml-textmate-bundle), but it's best installed using Sublime Text 2 package manager—Package Control (http://wbond.net/sublime_packages/package_control). Assuming that you have Sublime Text 2 installed, the following are the steps to install this package:

1. Go to Sublime's Package Control installation page and follow the instructions to install Package Control (http://wbond.net/sublime_packages/package_control/installation).
2. Open **Command Pallete** by pressing *Ctrl* + *Shift* + *P* (for Windows and Linux systems) or *command* + *shift* + *P* (OSX).
3. Start typing `Package` to reveal all the Package Control commands.
4. Select **Install package**.
5. In the new list that has appeared, type `Haml` to reveal all packages with `Haml` in their names.
6. Select the one that is named just `Haml`—Haml bundle for TextMate, and press *Enter*.
7. Restart Sublime Text 2.

This will give you the features explained in the following section.

Improved language

The following are the editor's features:

- Text inside Ruby, ERB, JavaScript, Sass, and CSS filters is now properly recognized, so you get all the syntax with the highlighted snippets, commands, and so on.
- Added built-in HTML recognition. This is helpful when you include HTML in your Haml.
- Ruby code inside `#{}` is now recognized as embedded Ruby.
- Updated language so that `<` and `>` are recognized as Haml constants.

Added snippets and commands

The following are the features:

- *command* + *alt* + *C* converts selection from HTML to Haml (and tries to convert ERB to Haml as well, although this feature is still in beta)
- ^ + > inserts `#{}` and toggles between `#{Ruby code}` and `#{ Ruby code }`
- Added snippets for attributes (*:Tab* and =>)
- Added full trigger snippets for common HTML elements (that is, `table`, `br`, `div`, `h1`, `h2`, and so on)

- *command* + */* uses rails comments `-#` instead of HTML comments
- *command* + *alt* + *X* escapes HTML special characters

TextMate

TextMate support for Haml is added using the same TextMate bundle that was described previously for Sublime Text 2. The features gained will be the same, but the installation procedure is different. The following are the steps for installation, assuming that you have TextMate and git installed:

1. Open a terminal.
2. Using the `cd` command, change the current working directory to the `Bundles` directory:

```
$ cd ~/Library/Application\
Support/TextMate/Bundles/
```

3. Use the `git clone` command to download the bundle:

```
$ git clone
git://github.com/phuibonhoa/handcrafted-haml-
textmate-bundle.git Haml.tmbundle
```

4. Force TextMate to reload bundles:

```
$ osascript -e 'tell app "TextMate" to reload
bundles'
```

Vim

Haml syntax highlighting can be added using an Haml runtime which is available at https://github.com/tpope/vim-haml.

To install it, download and extract it in `~/.vim or ~\vimfiles`.

If you are using MacVim, it does support Haml syntax highlighting out of the box.

Rubymine

Rubymine supports Haml out of the box.

Emacs

Emacs syntax highlighting for Haml and syntax-aware indentation is available through `haml-mode` which is available from GitHub (https://github.com/nex3/haml-mode).

Ensure that `haml-mode.el` is in a directory on your loading path, and add the following to your `~/.emacs` or `~/.emacs.d/init.el`:

```
(require 'haml-mode)
```

You will also need to ensure the `ruby-mode` is installed; the version in Emacs 24 is sufficient.

Coda

There is a plugin which enables Haml syntax highlighting for Coda. It is available from GitHub (https://github.com/gf3/haml.mode).

You can install it by checking out the repository into `~/Library/Application Support/Coda/Modes`; be sure to name the directory as `HAML.mode`.

Step 6 – Haml and Twitter Bootstrap framework

The `twitter-bootstrap-rails` gem supports Haml view generators if the `haml-rails` gem is installed. See step 2 of *Switching Rails application to use Haml as the templating engine* for instructions on how to install the `haml-rails` gem. For example, for our `Blog` application, we can quickly enable Bootstrap and make it overwrite our posts scaffold with Bootstrap-enabled Haml view templates. To do this, follow these steps:

1. Install the `twitter-bootstrap-rails` gem. Include the following lines in the `Gemfile` and run the `bundle install` command:

```
gem "therubyracer"
gem "less-rails"
gem "twitter-bootstrap-rails"
```

2. Run the `bootstrap:install` generator to install CSS stylesheets in our application:

```
rails generate bootstrap:install less
```

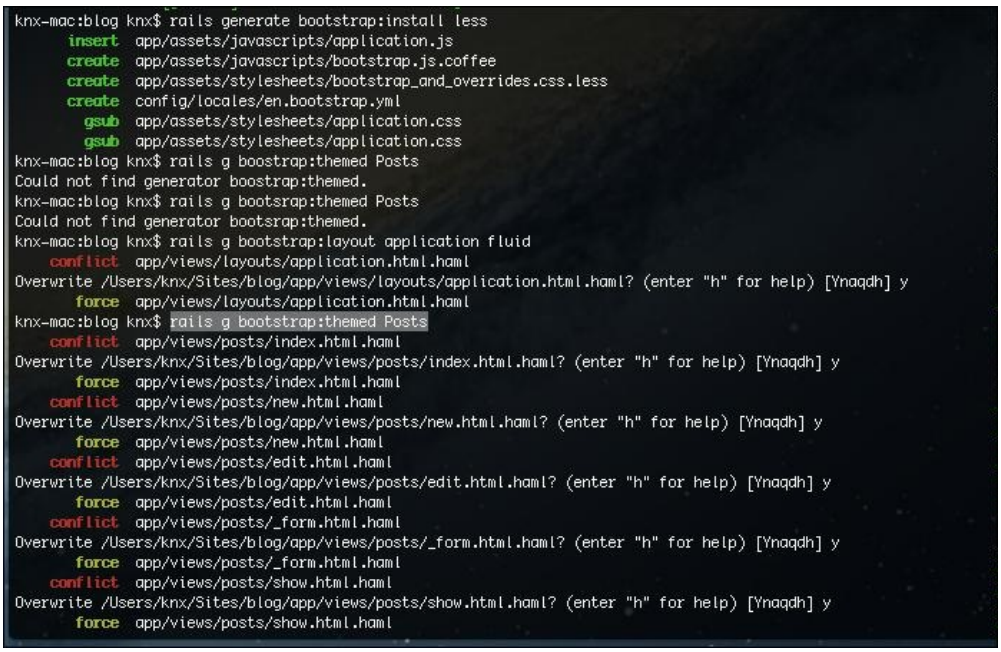
3. Run the `bootstrap:layout` generator to quickly create a Bootstrap-enabled application layout template:

```
rails g bootstrap:layout application fluid
```

4. Then run the `bootstrap:themed` generator to overwrite the `Posts` scaffold views with Bootstrap-enabled Haml-based views:

```
rails g bootstrap:themed Posts
```

You should get an output as shown in the following screenshot:



Top 6 features you'll want to know about

After you are familiar with the basics of Haml usage, you will probably soon want to know how to write HTML tags in Haml in more detail. In the next pages, we will go through all the most important things you can do with Haml to speed up your Rails development. First, you will learn how to create all the HTML tags and will learn about various shortcuts in creating HTML documents. Next, you will learn about commenting the Ruby code, evaluating Ruby, and using filters and helpers. At the end, you will learn about Haml's configuration options.

Using HTML tags and attributes

Let's review how to create basic HTML tags and various types of attributes, ids and classes.

Creating basic tags

To create any HTML tag, use a percent sign followed by the tag name. This will also take care of creating the closing tag for you. Nesting of elements or whole sections is specified by their indentation. For example:

```
%table
  %tr
    %td Hi, I'm a in a table!
```

Will be rendered as:

```
<table>
  <tr>
    <td>Hi, I'm in a table!</td>
  </tr>
</table>
```

Inserting HTML attributes

HTML attributes are specified using Ruby's hash syntax (it does support Ruby 1.9 style hashes). Local variables and Ruby evaluation can be used. For example, consider the following syntax:

```
    %a{:class => "carousel-control left", :href
=> "#carousel", :data => {:slide => "prev"}}
&lsaquo;
```

The previous code will be rendered as:

```
<a class="carousel-control left"
data-slide="prev" href="#carousel">>&lsaquo;</a>
```

Another example with Ruby evaluation (this example assumes that you are using it inside Rails view template), is as follows:

```
%li{class: "#{'active' if page.current?}"}
```

The previous code will be rendered as:

```
<li class="active"></li>
```

If `page.current?` returns `true`. If it evaluates to `false`, then the class attribute will be ignored, and not rendered at all.

You can specify class attributes as a Ruby array. This array's elements will be automatically joined with either space or an underscore (space for class and underscore for ids). For example, consider the following code (assuming that you have two Ruby arrays defined as: `@ids = ["first","second"]` and `@classes = ["class1","class2"]`):

```
%div{:id => @ids, :class => @classes}
```

The previous code will be rendered as:

```
<div id="first_second" class="class1
class2"></div>
```

The following is an example which you would use in a real Rails view template (assuming that the `@page` object is defined):

```
%body{id: [@page.type, @page.class], class:
[@page.mobile?, @page.setting]}
```

Will be rendered as (assuming that those variables and method contain/return values):

```
<body id="type1_marketing" class="mobile
normal"></body>
```

It is also possible to use a Ruby method that returns a hash value and use it in place of the attribute's hash contents (see Haml:Helpers description in

the *Using helpers and the ActionView extensions* section for some useful predefined helper methods).

For example, consider you have the following method defined in your helper:

```
def div_hash
  {id: "this_is_the_div", class: "classy"}
end
```

Then you can use it like:

```
%footer{div_hash}
```

It will be rendered as:

```
<footer id="this_is_the_div"
class="classy"></footer>
```

Using Boolean attributes

Some attributes do not need to have a value set—they only can be present or not present—for example, the `selected` attribute. You can write them by setting their value to `true`. Set their value to `false` when you do not want to render them at all. Consider the following:

```
%option{value:"car",selected: true} Car
```

The previous code will compile to:

```
<option selected value='car'>Car</option>
```

Inserting HTML5 data attributes

HTML5's custom data attributes can be rendered by using a `:data` key in the attribute's hash. This key should have another hash as its value, with the data attribute names and values. For example, (assuming that the `@user` object is defined) consider the following code:

```
%input{type: "text", name: "interest_tokens",
data: {load: @user.interests}}
```

Assuming that @user.interests returns an empty array. It will compile to:

```
<input type="text" name="interest_tokens"
data-load="[]">
```

Appending class and ID names

The `class` and `id` attributes can be appended to any tag name using the dot and percentage sign (the same as in CSS). They are placed immediately after the tag name and before the hash attribute. For example, consider the following code:

```
%div#tags
  %span.badge important
```

The previous code will compile to:

```
<div id="tags">
  <span class="badge">important</span>
</div>
```

Multiple classes can be appended as follows:

```
%span.text-info.smaller.answer-info
```

The previous code will compile to:

```
<span class="text-info smaller
answer-info"></span>
```

Self-closing tags

Self-closing tags (also called empty/void tags), can be written using a forward slash character placed at the end of the tag name. The most common example is the `br` tag:

```
%br/
```

The previous tag will compile to:

```
<br>
```

Using object references in class and ID names

Sometimes you need to add a class and ID to a tag, which are taken directly from a Ruby object. This can be accomplished using Haml object reference. You can put the Ruby object in square brackets directly after the tag name. The class will be set to the object's class and the ID will be set to

the object's class followed by the object ID (joined with an underscore). For example, consider the following code in your controller:

```
def index
  @users = User.all
end
```

And consider the following code in the view file:

```
- @users.each do |user|
  %tr[user]
    %td= user.name
```

The previous two code snippets will compile to:

```
<tr class="user" id="user_15">
  <td>Chris</td>
</tr>
```

> **Note**
>
> Everything in an Haml line which is not interpreted as a tag or attribute or anything else, will be rendered as plain text.

Generating doctypes

The page doctype is generated when you use the triple exclamation mark characters. The exact doctype that is generated depends on the `:format` option (see the *Changing Haml options* section for more details). When in its default setting, which is `:html5`, the `!!!` characters will generate `<!DOCTYPE html>`. For example, consider the following code:

```
!!!
  %html
    %head
      %title Homepage
    %body
      %h1 Welcome
      %p Welcome on our page
```

The previous code will be rendered as:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Homepage</title>
  </head>
```

```
  <body>
    <h1>Welcome</h1>
    <p>Welcome on our page</p>
  </body>
</html>
```

Adding comments to the markup

There are two types of comments in Haml. One will be rendered as commented-out HTML, with the `<!--` characters, and the other one will not be rendered at all.

Using HTML comments

Adding comments in parts of markup using HTML comments is done by using the forward slash character. You can place it at the beginning of the line that you would like to comment out, or at the beginning of the indented section of the code to comment out the whole section. Those comments will be visible in the HTML source. For example, consider the following code:

```
%badgerbadgerbadger
  /This is a badger element
  and here comes a snake!
```

The previous code will be rendered as:

```
<badgerbadgerbadger>
  <!-- This is a badger element -->
  and here comes a snake!
</badgerbadgerbadger>
```

To wrap a whole indented section of code at once, use the forward slash as shown in the following code:

```
  %p This will be commented out
  %badgerbadgerbadger
    and here comes a snake!
```

The previous code will be rendered as:

```
<!--
  <p> This will be commented out</p>
  <badgerbadgerbadger>
    and here comes a snake!
  </badgerbadgerbadger>
-->
```

If you need to use the Internet Explorer conditional comments, use the square brackets as shown in the following code:

```
/[if IE]
  %a{href:
https://www.google.com/intl/en/chrome/browser/ }
Get Chrome
```

The previous code will be rendered as:

```
<!--[if IE]>
  <a
href="https://www.google.com/intl/en/chrome/browser/">Get
Chrome</a>
<![endif]-->
```

Using Haml/Ruby comments

This is the second type of the comments that will not render any output in the HTML file and will be only visible in the Haml source. They are created by using a hyphen followed by a pound sign. For example, consider the following code:

```
%h1 Pages#about
-# Fill this page with content
%p Find me in app/views/pages/about.html.haml
```

The previous code will compile to:

```
<h1>Pages#about</h1>
<p> Find me in app/views/pages/about.html.haml</p>
```

Evaluating Ruby code

Basically, you can insert Ruby in two ways into Haml. One way will generate output into the resulting HTML file and the other one will just execute the Ruby code and not generate any output at all. This is equivalent to ERB `<%=` and `<%-` tags. Let's review those options in more detail.

Inserting Ruby code

Evaluation of Ruby code by inserting Ruby code is done using equals character. The Ruby code that is inserted this way will be run and its output will be inserted into the resulting HTML file. For example, consider the following code:

```
%h1= "main page".upcase
```

The previous code will be rendered as:

```
<h1>MAIN PAGE</h1>
```

You can also nest the Ruby code result into a tag, using indentation (assuming that `search_path` returns `"/search"`):

```
%li.active
 = link_to "Search", search_path
```

The previous code will be rendered as:

```
<li class="active"><a
href="/search">Search</a></li>
```

The equals character is also commonly used to create Ruby blocks that generate output. For example, in a Rails view, you can create Ruby blocks using `simple_form`:

```
= simple_form_for @question,
url: :questions, :html => {:class=>
"form-horizontal"} do |f|
```

Running Ruby code

Running Ruby code to execute it without generating any output is done using the hyphen character. The following code will not render anything:

```
  "This string will not be rendered"
```

The following code will set the variable class to contain the `"user"` string, but will not render any output:

```
  @class = "user"
```

Setting variables is the most common use case for this. A more complicated example is as follows:

```
- menu_links = {home: home_path, search:
search_path, sign_out: sign_out_path}
```

The previous will not render any output, but will set the `menu_links` variable to contain the hash.

Another common use is to create Ruby control structures (`if`, `case`, and so on). They are closed automatically based on the indentation (just as with HTML tags in Haml). For example, consider the following code:

```
- if true
%div True
- else
  %div False
```

The previous code will be rendered as:

```
<div>True</div>
```

A real life example could look as follows:

```
  %td.span2
    - if current_user.has_answered?(question.id)
        %span{:class=> "re-answer-button"}
          = link_to t(".re-answer"),
edit_answer_path(question.id), :class=> "btn"
    -else
      = link_to "Answer",
new_answer_path(question.id), :class=> "btn"
```

Assuming that `current_user.has_answered?` returns `true`, the previous code will be rendered as:

```
<td class="span2">
  <span class="re-answer-button">
  <a class="btn"
href="/answer/edit/23">Re-answer</a>
  </span>
</td>
```

Using Ruby interpolation

Ruby code can be inserted into Haml template just like in Ruby strings, by using `#{}` characters:

```
%p Welcome, #{current_user.nickname}
```

Is the same as:

```
%p= "Welcome, #{current_user.nickname}
```

And will compile to (assuming that current_user.nickname returns "Chris"):

```
<p>Welcome, Chris</p>
```

Escaping/unescaping

By default, Haml will sanitize any HTML-sensitive characters in the output. The backslash is used when you need to escape some characters, which would normally be processed. For example, (if the `@page.name` variable has `"Home page"` as value) consider the following code:

```
%nav
  = @page.name
  \= @page.name
```

The previous code will compile to the following because Haml will escape the `=` and `@` characters in the third line:

```
<nav>
  Home page
  = @page.name
</nav>
```

If you would like to disable the escaping, you can use the exclamation mark character. For example, consider the following code:

```
!= "This is a link tag: <a></a>"
```

The previous code will be rendered as:

```
This is a link tag: <a></a>
```

Without unescaping, this would be rendered as:

```
This is a link tag: &lt;a&gt;/&lt;a&gt;
```

Using filters

Filters are programs outside HAML, which can be used to process parts of text from an Haml file and their output will be rendered in HTML.

The type of the filter to use is specified by using a colon character followed by the name of the filter. Input text to be processed is designated by indenting. For example, consider the following code:

```
.content
  :markdown
    #Welcome
    * First item
    * Second item
    * Third item
```

The previous code will be rendered as:

```
<div class="content">
<h1>Welcome</h1>
<ul>
  <li>First item</li>
  <li>Second item</li>
  <li>Third item</li>
  </ul>
</div>
```

Ruby code can be interpolated inside the filter. This can be useful, for example, for passing contents of some variable through the filter:

```
- @articles.each do |article|
  %article.post
    %header=article.name
    :textile
      #{article.content}
    %footer
```

In this example, the `article.content` value will be passed through the `:textile` filter.

There are many more filters available such as `:cdata` (surrounds with CDATA tags), `:coffe` (for coffescript), `:css`, `:erb`, `:javascript`, `:less`, `:plain` (for large blocks of text that does not have to be parsed), `:preserve` (to preserve whitespace), `:ruby`, `:scss` and `:sass`.

Creating a custom filter

Haml allows the creation of custom Ruby filters. Let's walk over the steps to see how it can be done:

1. Create filter code: Create a file named `custom.rb` and save it in the `lib/haml/filters` directory inside your application directory tree

(create those directories if they don't exist). Paste the following code in the file:

```
module Haml::Filters::Custom
  include Haml::Filters::Base

  def render(text)
   #this is the actual filtering method, use
it to process
#the input text and return the output
  end

end
```

Replace the inside of the `render(text)` method with your code.

2. Integrate it with Rails: Create a file in `/config/initializers/` and paste the following code in it:

```
# config/initializers/custom.rb
require 'haml/filters/custom'
```

3. Use it in the view: Use your filter in the Haml templates, same as with any of the built-in filters:

```
:custom
  -# text to filter
```

Creating multiline attributes, Ruby code, and strings

Long HTML attributes lines of Ruby code or text can be spread out to multiple lines of Haml.

Attributes

Attributes can be divided into multiple lines by placing newlines after the commas. For example:

```
%html{:xmlns => "http://www.w3.org/1999/xhtml",
      "xml:lang" => "en",
      :lang => "en"}
```

Ruby code

Ruby code can be stretched to multiple lines by placing newlines just after the commas to make it more readable:

```
= f.input :interests_tokens,
```

```
            :placeholder =>
t(".optional_question_keywords"),
            :label=> t(".keywords"),
            :required => false,
            :input_html => { :data => {load:
@question.interests} }
```

Strings

Strings can be divided into multiple lines by using a pipe character. It is placed at the end of a line and the last line also must end with `|`. For example, consider the following code:

```
%p
  You may opt out of providing any such
information by |
simply choosing not to input the relevant data, |
  but certain information is compulsory if you
want to |
use certain parts of our Service. Some
information |
  which you voluntarily provide (e.g photos) may
constitute|
  "sensitive data" e.g by revealing your ethnic |
  origin, religion and/or sexual orientation. By
providing |
such data you specifically consent to our
processing it |
in accordance with the policy. |
```

Using helpers and the ActionView extensions

There are some useful methods, which can be used from any Haml template. They are useful in whitespace manipulation, appending text to Haml output, automatic class names generation, and lists rendering. Let's review some of the most useful ones:

- `surround`: It surrounds Haml output with text. When ran with two arguments, the first one will be added before the output and the second one after the output. When ran with one argument, it will be used on both sides. For example:

  ```
  = surround "(",")" do
    I will be surrounded
  ```

  The previous code will be rendered as:

  ```
  (I will be surrounded)
  ```

- `precede`: It adds text before Haml output and expects one argument.
- `succeed`: It adds text after Haml output and expects one argument.
- `find_and_preserve`: It converts newlines to HTML whitespace escape code to prevent them from breaking the indentation; it is useful on dynamically generated `<textarea>` and `<pre>` tags. There is also a shortcut for this method.

  `~` works just like =, except that it runs `Haml::Helpers#find_and_preserve` on its input. For example:

  ```
  ~ "Foo\n<pre>Bar\nBaz</pre>"
  ```

  The previous code is the same as:

  ```
  =
  find_and_preserve("Foo\n<pre>Bar\nBaz</pre>")
  ```

  And the same is rendered as:

  ```
  Foo
  <pre>Bar
  Baz</pre>
  ```

- `list_of`: This method is helpful during generation of lists. It takes an `Enumerable` object and iterates over it, yielding each element to an Haml block and surrounding the result in the `<li>` tags. This can be used, for example, when generating a navigation menu list:

  ```
  = list_of
  ["Home":"/","About":"/pages/about","FAQ":"/pages/faq","Contact":"/co
  {class: "nav} do |title,link|
    %a{href: link}= title
  ```

  The previous code will compile to:

  ```
  <li class="nav">
    <a href="/">Home</a>
  </li>
  <li class="nav">
    <a href="/pages/about">About</a>
  </li>
  <li class="nav">
    <a href="/pages/faq">FAQ</a>
  </li>
  <li class="nav">
    <a href="/contact">Contact</a>
  </li>
  ```

- `page_class`: It returns a string, which contains the name of the current controller and the action. It is useful for automatic generation of class names. For example:

```
%body{class: page_class}
```

  When used in a controller named `welcome` and action named `index`, the previous code will compile to:

```
<body class="welcome index"></body>
```

Changing Haml options

There are various configuration options, which can be set in a Rails initializer. For example:

```
# config/initializers/haml.rb
Haml::Template.options[:format] = :html4
```

The previous option will set the appropriate doctype to be generated from `!!!`.

Encoding of the HTML output can be specified manually using the `:encoding` option. This only works when using Ruby 1.9 or later. By default, the encoding generated by Haml is the same as used in the Haml view template.

To see the list of all Haml options go to http://haml.info/docs/yardoc/Haml/Options.html.

Using HAML outside of Rails

There is one issue with HAML—it is often presented in a way (like on it's official website), that is geared towards Rails developers and if you are thinking of creating a non-Rails site, you may get scared. Let's clear this unfortunate misconception.

When your project is static and is on HTML-only site, you may use HAML to drastically speed up your development. As a bonus there are many tools, which not only simplify development but also add many useful features to the workflow. They allow you to speed up prototype development, use project templates, replace bloated Content Management Systems, and preview your site in real time.

The tools for standalone HAML development

If you are working on a Mac machine, you are in luck, because there are many cool tools which will help you. We won't review them in great detail here, but you can check their web pages for more information.

Hammer

Hammer is a great Mac application, for static site development, which also has built-in HAML support. It is not an editor and allows you to use the editor of your choice, instead of focusing on HAML, SASS, and CoffeeScript compilation. It optimizes your code by compression and concatenation. It allows you to use special tags for HTML includes (such as in PHP) automatic file paths and variables. It also allows you to publish the site quickly for review and preview it in the browser. As of the time of writing, it's available in the AppStore for 23.99 dollars. You can check it at http://hammerformac.com. You may also want to check its free companion app, Anvil. It allows you to quickly manage and preview local websites (no cumbersome Apache configuration necessary)—http://anvilformac.com.

Codekit

Another application for Mac users is Codekit (28 dollars, from http://incident57.com/codekit).

Apart from doing the same things that Hammer does, it also optimizes your site images to reduce their size, minifies JavaScript and CoffeeScript, offers team collaboration features, and more.

LiveReload

LiveReload is more lightweight and more platform-independent tool which spares you manually refreshing the browser, and also compiles HAML and many more languages.

The Mac version is available from AppStore for 9.99 dollars and the Windows version is currently in alpha stage and can be freely downloaded for testing from the website: http://livereload.com.

StaticMatic, Jekyl, and Middleman

If GUI is not your thing, or you are on a Windows or a Linux computer, you may consider using StaticMatic, Jekyl, or Middleman. Those are free Ruby gems, which not only compile HAML for you but also will allow you to stick to a robust web development workflow. We have chosen Middleman for our example, as it offers most features. StaticMatic is simplest and Jekyl is geared towards blog creation.

Streamline static site development with Middleman

Middleman (http://middlemanapp.com) is a modern and well-designed static site generator, which takes many useful conventions from Rails, but still is simple to use and does not require extensive Ruby knowledge. Middleman is installed using RubyGems. You can install it using the following command:

```
gem install middleman
```

There are three phases to creating a website. First you run the middleman `init` command to create the site skeleton. Let's run it to create a skeleton for our landing page site. We will use a built-in HTML5 template. The following code will use the HTML5 boilerplate to create our website:

```
middleman init landing_page -template=html5
```

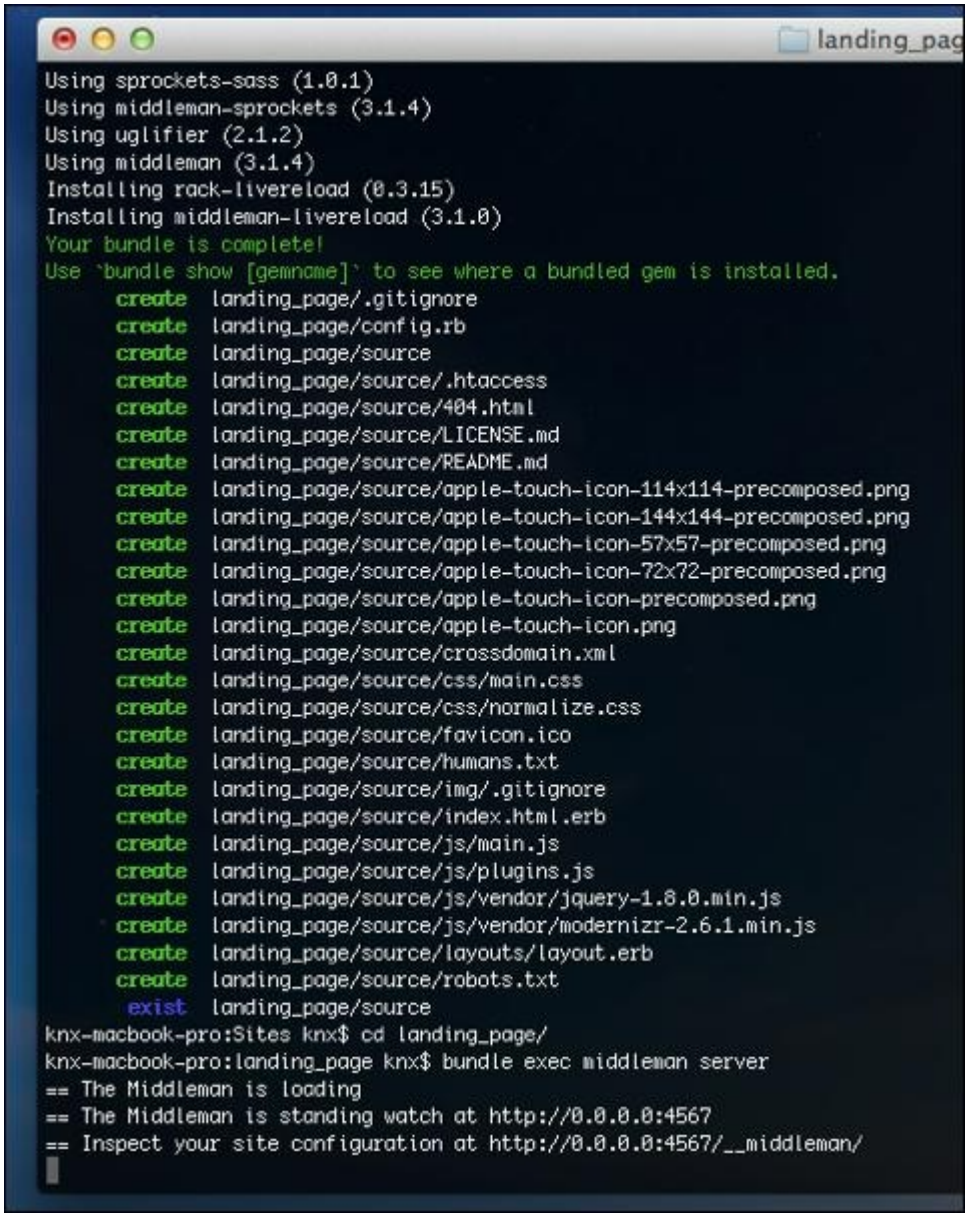
After the command completes its execution, you can change your directory to `landing_page` and you will have the complete site skeleton inside. It will contain the `source` folder, which will be holding your HAML files before they will be compiled. This folder will contain separate folders for javaScript, CSS, and images. There is also a `config.rb` file, which contains many settings which you can tweak to enable more complex features such as CSS and JavaScript minifying, or asset directory names.

Middleman separates your development and production files from the start. Your HAML files will be compiled to HTML and not included in production.

Next, we will start the preview development server, which will allow us to see the changes in real-time in a browser. Inside the `landing_page` directory run the following command:

```
bundle exec middleman server
```

You should see similar output, which gives you the preview URL in which you can view your HAML files rendered to HTML. In our example the URL is `http://0.0.0.0:4567`:



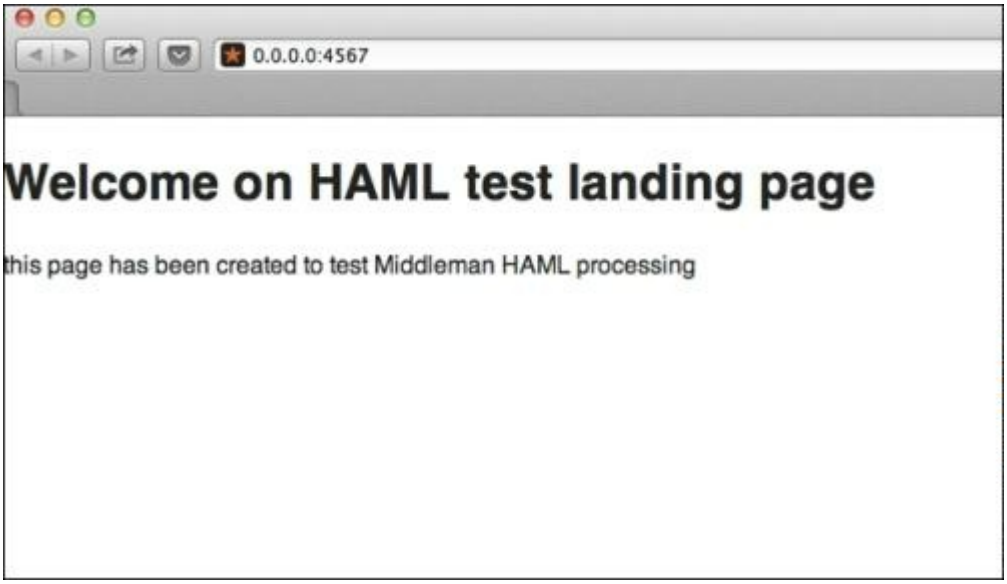
Middleman borrows many useful convention from Rails. One of them is the layouts concept. Basically you have a layout file which is located in `./source/layouts/layout.erb` and this layout will be applied automatically to all your pages. If you view the `layout.erb` source file, you can see a `<%= yield %>` statement. This is where your page content will be injected. The only page that has been generated so far is `./source/index.html.erb` page. We will remove it and create an Haml page instead to test the Haml rendering capabilities of Middleman.

Please remove the `./source/index.html.erb` file and create `./source/index.haml.erb` instead. Type some Haml code inside or paste the following code:

```
%h1 Welcome on HAML test landing page
```

```
%p this page has been created to test Middleman
HAML processing
```

If you now visit the preview URL, you should see the following page (or similar, if you have not pasted the Haml code from example, but typed your own):



This means that the HAML source page has been successfully rendered to HTML.

Exporting static site

One last thing that you will usually do is export your site to a static bundle, which can be deployed on the production server. This is done by running the build command.

```
bundle exec middleman build
```

This will create the `./build` directory inside of your site directory. This build directory will contain all of the site static assets, HAML processed to HTML, images, javaScripts, and CSS. You can then deploy it to your production server manually or use some other deployment method (many of these are described on the Middleman website).

We have only scratched the surface of what Middleman can do to make your static site development much more organized and efficient. I encourage you to read the *Getting Started* guide at http://middlemanapp.com/getting-started/ if you want to know more.

People and places you should get to know

If you need help with Haml, then here are some people and places which will prove invaluable.

Official sites

- Homepage: http://haml.info
- Manual and documentation: http://haml.info/docs.html
- Blog: http://blog.haml.info
- Source code: https://github.com/haml/haml

Articles and tutorials

- The official tutorial: http://haml.info/tutorial.html

Community

- Official mailing list: http://groups.google.com/group/haml
- Official forums: https://groups.google.com/forum/?fromgroups#!forum/haml
- Official IRC channel: `#haml` on `irc.freenode.net`
- User FAQ: http://haml.info/docs/yardoc/file.FAQ.html

Twitter

- *Norman Clarke*, the author of *Haml Spec* is on Twitter as `@compay`: http://twitter.com/compay