Ramaze

Michael Fellinger

May 14, 2008

A detailed in-depth walk through the features and behaviours of Ramaze. This document covers Ramaze::Controller, Ramaze::Helper, Ramaze::Template and Ramaze::Action and their interaction with each other.

Contents

1 Introduction

1.1 About Ramaze

Ramaze is a modular web application framework, it provides you with just about everything to make your daily development simple and fun.

Making programming fun is a concept popularized by the Ruby programming language in which Ramaze is written. This document assumes at least basic knowledge about Ruby. If you do not know what Ruby is yet, visit`http://ruby-lang.org` and find out, but beware, it may change your life.

1.1.1 Important Links

Naturally for an open source web framework, apart from reading the source most help can be found online. So here are some links to provide more information on all the topics covered in this book.

`http://ramaze.net` provides further documentation, screencasts, links and news.

`http://ramaze.googlegroups.com` hosts the mailing list for discussion with the developers and users of the framework.

1.2 About the author

Michael Fellinger (a.k.a manveru) is initiator and core developer of the Ramaze framework.

His programming language of choice is, as you might have guessed already, Ruby.

Michael lives in Tokyo/Japan since 2006, but is Austrian, whenever you find strange English in this article, blame it on the daily influence of Engrish and that German is his native language.

2 Ramaze

As mentioned above, Ramaze is a web application framework, but also features a readable open source codebase licensed under the Ruby license (optional GPLv2).

The strenghts of Ramaze, as described by its users, are a free style of development, enjoying all the benefits of the underlying Ruby programming language without having the library getting into your way of doing things but instead helping you along as you require it.

2.1 Installation Prerequisits

Ruby `http://ruby-lang.org`

Rubygems `http://rubygems.org`

rack `http://rack.rubyforge.org`

ramaze `http://ramaze.net`

2.2 Installing a specific version

Installing Ramaze should be very simple on most systems. I will cover a couple of ways here.

2.2.1 On Windows

2.2.2 On OSX

2.2.3 On Linux

Issue your package manager to install a package commonly called„rubygems"

ArchLinux

```
$ pacman -S rubygems
```

Debian

```
$ apt-get install rubygems
```

Now that you have rubygems installed, there are a few useful steps to make rubygems run without root permissions by installing them into your home or otherwise by you controlled directory.

I strongly recommend doing this as it simplifies further diving into the ruby ecosystem without having to use sudo or switch to root via su.

Simply add following lines to your ~/.bashrc or ~/.zshrc

```
export RUBYOPT=-rubygems
export GEM_HOME="$HOME/.gems"
export GEM_PATH="$GEM_HOME"
export PATH="$HOME/.gems/bin:$PATH"
```

Once you have done that you can simply run

```
gem install ramaze
```

And Ramaze will be installed.

2.2.4 On Linux (manual)

Download and install from http://rubygems.org

Visit `http://rubyforge.org/frs/?group_id=126`

and download the latest version as tgz.

2.3 Installing the development version

Ramaze is developed in a git repository, so if you want to get the latest source you can simply get your own copy. Please note that, since all our commits are checked by the spec suite before they get into the official repository, they are generally safe for production use and even recommended since recent security and code flaws have been applied.

Installing the gem is only recommended if you want to have a version of ramaze that„just works" or if you have only the ability to use ramaze through the gem (possible on some system setups).

2.3.1 Latest as tarball

You can download a tarball directly from github, the location of our Ramaze repository is at `http://github.com/manveru/ramaze`.

2.3.2 Getting git

Git is available on most Linux distributions

2.3.3 Clone the repository

Cloning the repository, a.k.a getting your own copy, is really simple once you have installed git.

```
$ git clone git://github.com/manveru/ramaze
```

2.3.4 Setting up your environment after installation

Now that you have the latest version of Ramaze, you have to tell Ruby how to find it. One of the simplest ways is to add a file called„ramaze.rb" in your site_ruby directory.

```
$ sudo echo 'require "/path/to/ramaze/lib/ramaze.rb"' > `ruby -e 'puts $:.grep(/s
```

This way, everytime you say„require 'ramaze"' in your code, it will first require the ramaze.rb in the site_ruby directory, which in turn requires the ramaze.rb from your development version.

3 Basic Usage

3.1 Hello, World!

A short introductionary example is always„Hello world", in Ramaze this looks like following.

```
require 'rubygems'
require 'ramaze'

class MainController < Ramaze::Controller
  def index
    "Hello, World"
  end
end

Ramaze.start
```

First we require rubygems, the package managing wrapper that allows us to require the ramaze library and framework. Next we define a Controller and method that will show up when accessing„http://localhost:7000/", 7000 being the default port of Ramaze.

3.2 Configuration

Configuration is mainly done by the Ramaze::Global singleton. It's an instance of Option::Holder, containing a rather large list of options and their defaults. Every option is documented in minimal style and the defaults should make clear in most cases how to set these options.

Configuration can be done either from Environment variables or command-line arguments or directly in your source.

Let's see the different ways, three variations of how to set your port in source first.

```
Ramaze::Global.port = 8080

Ramaze::Global.setup do |g|
  g.port = 8080
end

Ramaze.start :port => 8080
```

now by commandline arguments

```
$ ruby start.rb --port 8080
```

and finally using an environment variable

```
$ RAMAZE_PORT=8080 ruby start.rb
```

3.3 Application with multiple files

The convention for larger applications consists of a basic (voluntary) directory structure.

```
/
    start.rb
    ramaze.ru
    public/
        favicon.ico
        robots.txt
    model/
        user.rb
    view/
        index.xhtml
        user/
            index.xhtml
            view.xhtml
            new.xhtml
    controller/
        main.rb
        user.rb
```

Following this layout is fully up to you, but it is recommended to make it easier for other people to jump right into your code and understand things.

3.3.1 Some directory conventions in detail

There are two special cases you should be aware of before starting, /public and /view are defaults in Ramaze, set in Ramaze::Global.public_root and Ramaze::Global.view_root respectively, they are relative to Ramaze::Global.root

To illustrate this, see following example:

```
Ramaze::Global.root        # => '/home/manveru/demo'
Ramaze::Global.view_root   # => '/home/manveru/demo/view'
Ramaze::Global.public_root # => '/home/manveru/demo/public'
```

In practical terms this means that templates are found in the /view directory while static files are served from /public

The rest of the conventions are not connected to any defaults and you will have to manually require your .rb files from start.rb, or however your file is named that contains„Ramaze.start".

4 Ramaze::Controller

4.1 What, exactly, is a Controller anyway?

The first thing you will encounter when starting out with Ramaze is the Controller. This is because the Controller is the central structure in most web applications and for that reason you should know as much as possible about it.

To start out, let's take a look at the basic structure of a usual Controller with a hello world example.

```
class MainControler < Ramaze::Controller
  def index
    'Hello, World!'
  end
end
```

So we have a class MainController that inherits from Ramaze::Controller, I won't go into details yet what exactly this inheritance means, but it basically integrates your new class into your application.

What you see next is a method called index, containing a simple String.

To fully understand this snippet we will have to understand the concept of mapping in Ramaze. If you are familiar with other web frameworks, you will know about routing, and for those of you who don't, let me explain routing real quick, since it's the principle applied here as well.

So routing basically means, when you get a request from a client to "/user/songs", the routing will see if you have defined any routes matching this path and executes whatever you specified as the result.

What Ramaze does with mapping is a little bit more sophisticated in order to lift work off the shoulders of the programmer and only make him do manual routing to refine the results of this automatic procedure.

So given a Controller named UserController, Ramaze will route to it with the path „/user", UserNameController would be put at „/user_name" and so on. In this case we have picked the only exception, that is MainController and is mapped at „/" by default.

We will see in the next example how to change this default mapping, since defaults may be good, but nothing beats configurability when needed.

4.2 Remapping

In order to change the default mapping, you can simply use the Controller::map(path, [path2, ...]) method. This also allows you to map one Controller on multiple paths.

```
class MainController < Ramaze::Controller
  map '/article'

  def index
    "Hello, World!"
  end
end
```

This example simply shows how you can use the map method, but let's see a nicer way to do the same using the default mapping.

```
class ArticleController < Ramaze::Controller
  def index
    "Hello, World!"
  end
end
```

So instead of defining a Controller that is by default mapped to„/" and remapping to„/article", we can just name it ArticleController and Ramze will take the part of your classes name before„Controller" and convert it to a mapping by doing a simple snake case transformation.

Sticking to this kind of defaults makes your code a lot more readable, as people know which Controller will map to which URL without even checking first.

4.3 Controller class methods

4.3.1 Controller::startup

4.3.2 Controller::map

4.3.3 Controller::at

4.3.4 Controller::layout

4.3.5 Controller::template

4.3.6 Controller::check_path

4.3.7 Controller::engine

4.3.8 Controller::current

4.3.9 Controller::handle

4.3.10 Controller::relevant_ancestors

4.3.11 Controller::render

5 Ramaze::Helper

Helpers are modules for inclusion into controllers or other classes.

5.1 Using a helper

```
class MainController < Ramaze::Controller
  helper :formatting

  def index
    number_format(rand)
  end
end
```

5.2 Rules for helpers

Methods are private.

5.3 How to create your own helper

Creating helpers is simple.

```
module Ramaze
  module Helper
    module SimleyHelper
      FACES = {
        ':)' => '/smilies/smile.gif'
        ';)' => '/smilies/twink.gif'
      }
      REGEX = Regexp.union(*FACES.keys)

      def smile(string)
        string.gsub(REGEX){ FACES[$1] }
      end
    end
  end
end
```

5.3.1 Exposing methods to routing

By adding an helper module to the Ramaze::Helper::LOOKUP Set it's possible to add the module to the lookup for methods together with the Ramaze::Controller. Conflicts of method names in Helper and Controller will prefer the Controller, following the same rules as Ruby inheritance.

```
module Ramaze
  module Helper
    module Locale
      LOOKUP << self
```

```
      def locale(name)
        session[:LOCALE] = name
      end
    end
  end
end
```

In this example we expose the public method Locale#locale (Ruby methods are public by default). So in your application your can just use the helper and when the client visits the /locale/en route the session will reflect this choice.

Please note that this code doesn't include a redirect_referrer call since we may be using it within our own codebase in the middle of a method.

6 Ramaze::Action

Action is one of the core parts of Ramaze, a recipe for rendering combinations of controllers, methods and templates.

6.1 What is an Action

An Action, at the lowest level, is a Struct with following members:

6.1.1 Action#method

Refers to the name of the method to be invoked before a template is evaluated.

The return value of the method is kept as the result of Action#render if there is no Action#template.

6.1.2 Action#params

The parameter are part of the #__send__ to Action#method. You can visualize this as:

```
if method = action.method
  action.instance.__send__(method, *action.params)
end
```

In Lisp/Scheme terms, they are being applied.

Please note that the params are given as Strings due to the fact that they are extracted from the URI of the HTTP method and so Ramaze has no chance of determining what kind of Object this should represent.

6.1.3 Action#template

If there is a template found during the procedure of finding a fitting Action for the given request, this member is set to the relative path from your Global.view_root or Controller.view_root directory. The template is evaluated unless the templating engine is set to :None and the result of this evaluation is set as the response.body.

6.1.4 Action#controller

Points to the controller this Action operates on, also see Action#instance.

6.1.5 Action#path

A String represenation of the Action without params, generally it's the name of the method or (if no method but a template found) the name of the template without extension. It's used for different kinds of hooks.

6.1.6 Action#binding

The binding to the Action#instance, obtained by doing an instance_eval{ binding} on it. This is only done when the binding hasn't been set yet.

6.1.7 Action#engine

Refers to the templating engine that this Action is passed to when Action#render is called.

6.1.8 Action#instance

Instance of the Controller, lazily obtained through Action#controller.new on access of this member. Note that this is also invoked as a dependency of Action#binding.

6.2 Create an Action

Creating an Action can be verbose and is usually not required outside the direct proximity within the codeflow.

```
action = Ramaze::Action(:controller => MainController)
```

6.3 Render an Action

We did all this just to render an Action, so let's do that already.

```
class MainController < Ramaze::Controller
  def index
    "Hello, World!"
  end
end

Ramaze::Action(:controller => MainController, :method =>
:index).render
# => "Hello, World!"
```

7 Ramaze::Template

7.1 Engines

7.2 Ezamar

Ezamar is the default templating engine shipped with Ramaze, it's a very simple implementation but offers you most things you will demand from templating.

One of the advantages of Ezamar is quite fast execution based on the optimized String interpolation in Ruby.

Also, your templates will work out of the box, you don't have to install anything besides Ramaze.

7.2.1 Syntax

Output is done by using #{} syntax, the last expression inside the parenthesis is interpolated in the final template.

The PI (Processing Instruction) is <?r ?>, r standing for Ruby, In future more PIs may become available.

To summarize:

#{ (expr) }	Interpolates like in normal Ruby String and shows in result.
<?r (expr) ?>	Only execute; <expression> is sourrounded by semicolon.

7.2.2 Usage

As an example, let's see hello world using all basic features of this engine.

```
<html>
  <head><title>Hello, World!</title></head>
  <body>
    <h1>Hello, World!</h1>
    <?r 10.times do |i| ?>
      <p>#{ordinal(i)} times i've said hello</p>
    <?r end ?>
  </body>
</html>
```

7.3 Erubis

Erubis is an implementation of eRuby, giving you the well known ERB syntax.

It is very fast, around three times faster than ERB and even 10% faster than eruby.

Other features are

- Multi-language support (Ruby/PHP/C/Java/Scheme/Perl/Javascript)
- Auto escaping support
- Auto trimming spaces around '<% %>'
- Embedded pattern changeable (default '<% %>')

- Enable to handle Processing Instructions (PI) as embedded pattern (ex. '<?rb ... ?>')
- Context object available and easy to combine eRuby template with YAML datafile
- Print statement available
- Easy to extend and customize in subclass

Erubis is implemented in pure Ruby, so it works on most implemenations of the language. According to their dependencies it requires Ruby 1.8 or higher.

7.3.1 Syntax

Erubis provides you default ERB syntax, PIs are inside <% %>, output is done by <%= %>

Summary:

<% (expr) %>	Execute expression, ignore result
<%= (expr) %>	Interpolate last expressions result
<?rb (expr) %>	Execute expression, ignore result

7.3.2 Usage

```
<html>
  <head><title>Hello, World!</title></head>
  <body>
    <h1>Hello, World!</h1>
    <% 10.times do |i| %>
      <p><%= ordinal(i) %> times i've said hello</p>
    <% end %>
  </body>
</html>
```

7.4 Haml

Haml is a templating engine for (X)HTML, designed to make it both easier and more pleasant to code your documents.

It attempts to eliminate redundancy, reflecting the underlying structure that the document represents, and providing elegant, easily understandable but powerful syntax.

It's closely related to the Sass templating engine also supported by Ramaze.

Syntax

Haml syntax is more involved as it replaces conventional HTML tags completely and providing its own indentation-based alternative.

The way to create tags can be shortly summarized with

```
%tagname{ :attr1 => 'value1', :attr2 => 'value2' } Contents
# Equivalent in HTML:
<tagname attr1='value1' attr2='value2'>Contents</tagname>
```

There are shortcuts, especially for the <*div*> tag:

```
#foo Bar
# Equivalent in HTML:
<div id='foo'>Bar</div>
```

To nest tags, just adjust their indentation:

```
#foo Bar
  #bar Foo
# in HTML:
<div id='foo'>Bar
  <div id='bar'>Foo</div>
</div>
```

Which already shows you how this can be a very powerful way to write your templates.

An example embedding Ruby code:

```
%p Date/Time:
  - now = DateTime.now
  %strong= now
  - if now > DateTime.parse('December 31, 2006')
    = 'Happy new ' + 'year!'
```

As you can see, lines prefixed with '-' are executed but their results are not shown in the result, contrary to that, lines starting with '=' are executed and their result is interpolated in the template.

Haml doesn't only omit ending tags by indentation but the same mechanism also helps to the 'end' statements in embedded Ruby code. This way the HTML and Ruby languages are connected in their side-effects, providing you with a quite terse but still readable and DRY syntax to write your templates in.

There are downsides to this approach, as it's not possible to control how exactly the final document is generated, but Haml is gaining momentum in the Ruby community very fast despite that fact, as it's simply not important for most applications on the web.

Usage

```
%html
  %head
    %title Hello, World!
  %body
    %h1 Hello, World!
    - 10.times do |i|
      %p= "#{ordinal(i)} times i've said hello"
```

7.5 Sass

Haml is a templating engine for CSS, designed to make it both easier and more pleasant to create styles for your documents.

It attempts to eliminate redundancy, reflecting the underlying structure that CSS represents, and providing elegant, easily understandable but powerful syntax.

It's closely related to the Haml templating engine also supported by Ramaze.