Journey to Ramaze

Michael Fellinger
<m.fellinger@gmail.com>
version 0.2, May 2009

Table of Contents

JavaScript must be enabled in your browser to display the table of contents.

1. Journey to Ramaze
2. Preface
    1. 1. Ruby
        1. 1.1. Features
    2. 2. Short history of web development
    3. 3. About the "Journey to Ramaze"
    4. 4. Relevant links
    5. 5. About the author
3. Tutorial introduction to Ramaze
    1. 1. Installation
        1. 1.1. Checking for and Installing Ruby
        2. 1.2. Installing Ramaze
    2. 2. Hello, World!
    3. 3. The ramaze command
        1. 3.1. Create
        2. 3.2. Start
        3. 3.3. Stop
        4. 3.4. Restart
        5. 3.5. Console
4. Middleware
    1. 1. Building middleware
    2. 2. Ramaze modes
        1. 2.1. The :dev and :live modes
        2. 2.2. Your own modes
    3. 3. Rack middleware
        1. 3.1. Rack::Lint
        2. 3.2. Rack::CommonLogger
5. Configuration
    1. 1. Basic options
6. Sessions
    1. 1. What are sessions
    2. 2. Usage example
    3. 3. Access
    4. 4. Cookies
        1. 4.1. History
        2. 4.2. Structure
        3. 4.3. Usage
        4. 4.4. Limitations

5. 5. Configuration of session cookies
    1. 5.1. Expiration
6. 6. Concurrency
7. 7. Flash
    1. 7.1. Implementation

7. Helpers
    1. 1. Usage
    2. 2. Making your own
    3. 3. Naming
    4. 4. Helper file lookup
    5. 5. Methods during action creation
    6. 6. Default helpers
    7. 7. Aspect helper
        1. 7.1. Using before_all
    8. 8. CGI helper
        1. 8.1. Encoding text for use in URIs with url_encode
        2. 8.2. Decoding text from URIs with url_decode
    9. 9. Flash helper
    10. 10. Link helper
    11. 11. Redirect helper
    12. 12. Render helper

8. Testing
    1. 1. About testing
    2. 2. Ramaze and testing
    3. 3. Testing your application
    4. 4. Bacon
        1. 4.1. Usage
        2. 4.2. Shared contexts
    5. 5. Innate::Mock
    6. 6. Rack::Test
    7. 7. Hpricot
    8. 8. Webrat
    9. 9. Mechanize
    10. 10. Examples
        1. 10.1. GET
        2. 10.2. POST
        3. 10.3. Multipart POST
        4. 10.4. Working with models

9. Actions
    1. 1. History
    2. 2. Structure
    3. 3. Creating actions
        1. 3.1. The hard way
        2. 3.2. The easy way

10. Views
    1. 1. History
    2. 2. Structure
        1. 2.1. The index template
        2. 2.2. Faking depth

            3. 2.3. Inline templates
        3. 3. Path lookup
        4. 4. Configuration
            1. 4.1. Global configuration
            2. 4.2. Controller configuration
        5. 5. Content representations
11. Layouts
        1. 1. History
        2. 2. Usage
            1. 2.1. Layout from template
            2. 2.2. Layout from method
            3. 2.3. Layout directories per controller
        3. 3. Structure
        4. 4. Path lookup
            1. 4.1. Inline templates
        5. 5. Configuration
        6. 6. Content representations
        7. 7. Implementation
12. Rack spec
        1. 1. Rack applications
            1. 1.1. The Environment
                1. 1.1.1. The Input Stream
                2. 1.1.2. The Error Stream
            2. 1.2. The Response
                1. 1.2.1. The Status
                2. 1.2.2. The Headers
                3. 1.2.3. The Content-Type
                4. 1.2.4. The Content-Length
                5. 1.2.5. The Body
            3. 1.3. Thanks
13. Glossary

Preface

Ramaze is a simple but powerful web application development framework. This book is an in-depth walk-through of Ramaze's features and behavior.

Ramaze is a modular web application framework. It provides you with just about everything you need to make your daily web development simple and fun. Making programming fun is a concept popularized by the Ruby programming language.

Ramaze is written in Ruby. This book assumes at least basic knowledge about Ruby. If you do not know what Ruby is yet, visit the official Ruby programming language website and find out; but beware, it may change your life. It changed mine for sure.

Ramaze features a readable open source codebase licensed under the Ruby license (optionally GPL version 2).

The strength of Ramaze, as described by its users, is a free style of development, affording all the

benefits of the underlying Ruby programming language. It gets out of your way as you do things, helping you along as you require it.

1. Ruby

It makes sense to write a little about Ruby first before starting our journey to Ramaze.

Ruby was created in 1995 by the Japanese developer Yukihiro Matsumoto. It wasn't until 2000, when the first Programming Ruby was published, that Ruby gained any kind of popularity outside of Japan.

1.1. Features

The manpage of Ruby lists the features of the language as follows:

Interpretive

> Ruby is an interpreted language, so you don't have to recompile programs written in Ruby to execute them.

Variables have no type (dynamic typing)

> Variables in Ruby can contain data of any type. You don't have to worry about variable typing. Consequently, it has a weaker compile time check.

No declaration needed

> You can use variables in your Ruby programs without any declarations. Variable names denote their scope, local, global, instance, etc.

Simple syntax

> Ruby has a simple syntax influenced slightly from Eiffel.

No user-level memory management

> Ruby has automatic memory management. Objects no longer referenced from anywhere are automatically collected by the garbage collector built into the interpreter.

Everything is an object

> Ruby is a purely object-oriented language, and was so since its creation. Even such basic data as integers are seen as objects.

Class, inheritance, and methods

> Of course, as an object-oriented language, Ruby has such basic features like classes, inheritance, and methods.

Singleton methods

Ruby has the ability to define methods for certain objects. For example, you can define a press-button action for certain widget by defining a singleton method for the button. Or, you can make up your own prototype based object system using singleton methods, if you want to.

Mix-in by modules

Ruby intentionally does not have multiple inheritance as it is a source of confusion. Instead, Ruby has the ability to share implementations across the inheritance tree. This is often called *Mix-in*.

Iterators

Ruby has iterators for loop abstraction.

Closures

In Ruby, you can objectify the procedure.

Text processing and regular expressions

Ruby has a bunch of text processing features like in Perl.

Bignums

With built-in bignums, you can for example calculate `factorial(400)`.

Exception handling

As in Java™.

Direct access to the OS

Ruby can use most UNIX system calls, often used in system programming.

Dynamic loading

On most UNIX systems, you can load object files into the Ruby interpreter on-the-fly.

2. Short history of web development

Well until 2005 Ruby didn't get much mainstream use. That changed when Ruby on Rails started attracting web developers that were increasingly annoyed by Java and PHP.

Although Ruby on Rails has achieved widespread popularity even outside of the realms of web programming, it was by no means the first web framework to be written in Ruby.

One of the surviving frameworks from the time before Rails is IOWA, which is focused on discrete reusable components rather than MVC. Although it hasn't seen many changes since 2007, it's still one of the most solid implementations of a web framework and used in various contexts.

Another one is Nitro, which was first released in 2004. Nitro had a large impact on the Ramaze project. It was the first web framework that tried to enable people to write their applications in a multi-paradigm style, utilizing whatever is best suited for the task at hand. It gained some popularity among the Ruby community but eventually died a slow death.

3. About the "Journey to Ramaze"

This book is the result of a long time spent researching, specifying, and describing the exact behaviours of Ramaze.

This book is available as HTML and PDF, the asciidoc source can be found in ramaze-book on Github.

4. Relevant links

Naturally, since Ramaze is an open source web framework, most help can be found on the web. Here are some links which provide more information on all the topics covered in this book.

- The offical homepage for the ramaze project: ramaze.net.
- The ramaze googlegroup mailing list where most developers using Ramaze are subscribed and share help and information.
- Ramaze on Github provides immediate access to the source and commit-history of the project.
- A wiki to exchange information and sketch out documentation can be found at wiki.ramaze.net
- The automatically generated API documentation for the latest release is at doc.ramaze.net
- The place to report all issues, proposals, and bugs is bugs.ramaze.net

5. About the author

Michael Fellinger (a.k.a. manveru) is the creator and a core developer of the Ramaze project.

He started programming in middle-school using QBASIC. Since then he learned various programming languages, among them Smalltalk, Dylan, Neko, Forth, NASM, PHP, and works on his own languages for fun.

His programming language of choice is, as you might have guessed already, Ruby, having worked with it since 2005.

Around 1999, Michael started creating websites, and was using PHP for several years. He was attracted to Ruby by the Ruby on Rails web framework, but was disappointed by the lack of freedom given to developers.

After some searching, he finally began working with the Nitro web framework, and was offered a job in Japan soon after finishing school. He spent an extensive amount of time studying and using Ruby and Nitro until the Nitro project was discontinued. He then applied his knowledge of Ruby and started the Ramaze project in the spirit of Nitro, which tried to be a tool that may be utilized in any way a developer wishes, not commanding any one true way.

Michael has been living in Tokyo, Japan since 2006, but is of Austrian nationality, speaking German, English and currently learning Japanese.

You can reach him by mailing to <m.fellinger@gmail.com>

Tutorial introduction to Ramaze

1. Installation

The easiest way to get to know Ramaze is by example, so we will start out with that, covering installation and a few simple applications so you can get a feel for it.

In order to use Ramaze, you will have to install it on your system. This is usually fairly straightforward.

1.1. Checking for and Installing Ruby

Before we install Ramaze, we need to ensure your system has Ruby, since Ramaze is written in the Ruby programming language. Ruby doesn't ship by default with most systems, so you will likely have to install it first.

Check your system for Ruby with this command:

```
$ ruby --version
```

If you see output similar to this, you are good to go:

```
ruby 1.9.1p0 (2009-01-30 revision 21907) [i686-linux]
```

If you see a "command not found" error, or you see a version less than 1.8.5, you'll need to install or upgrade Ruby.

For this book we assume an installation of Ruby 1.9.x, which ships with RubyGems, the package manager for Ruby libraries and applications.

The reasoning behind using 1.9.x is that this book might take some time to get finished, and by then I hope that all major development in Ruby will happen on the basis of the 1.9 spec.

For some reason people have the impression that Ramaze will only work on 1.9, we still keep it backwards-compatible to 1.8.6 for some time.

Linux users can generally use their package manager to install Ruby. Otherwise, you can obtain Ruby from the official Ruby programming language website, where you will find instructions for installation on most systems.

1.2. Installing Ramaze

Once Ruby is installed correctly, you can install Ramaze simply by:

```
$ gem install ramaze
```

This will take care of installing all dependencies as well. There are a lot of libraries that we will install in the course of this book, but installing them is usually just a command away, so we defer it to when we actually need them.

Ramaze is a project that is developed in an open manner by the community. In order to work together we utilize git for revision control. You can obtain your own copy of the repository if you are interested in helping the development or simply would like to browse through the project history. We will cover this subject in developing Ramaze.

2. Hello, World!

A short introductory example is always "Hello world". In Ramaze this looks like following.

```
require 'rubygems'
require 'ramaze'

class Hello < 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 greet us when accessing *http://localhost:7000/*, `7000` being the default port of Ramaze.

To start this application we can now simply:

```
$ ruby start.rb
```

That will start an instance of Ruby and start the WEBrick HTTP server that ships with Ruby.

This will output something along the lines of:

```
delta ~/tmp/tutorial % ruby start.rb
W [2009-04-01 15:35:24 $17562]  WARN | : No explicit root folder found, assuming
it is .
D [2009-04-01 15:35:24 $17562] DEBUG | : Using webrick
I [2009-04-01 15:35:24 $17562]  INFO | : WEBrick 1.3.1
I [2009-04-01 15:35:24 $17562]  INFO | : ruby 1.9.2 (2009-03-02) [i686-linux]
D [2009-04-01 15:35:24 $17562] DEBUG | : TCPServer.new(0.0.0.0, 7000)
D [2009-04-01 15:35:24 $17562] DEBUG | : Rack::Handler::WEBrick is mounted on /.
I [2009-04-01 15:35:24 $17562]  INFO | : WEBrick::HTTPServer#start: pid=17562
port=7000
```

Now you can open your browser, and go to http://localhost:7000/.

Sometimes you will get an error when starting Ramaze that looks like:

```
delta ~/tmp/tutorial % ruby start.rb
/usr/lib/ruby19/1.9.1/webrick/utils.rb:73:in `initialize': Address already in use
- bind(2) (Errno::EADDRINUSE)
  from /usr/lib/ruby19/1.9.1/webrick/utils.rb:73:in `new'
  from /usr/lib/ruby19/1.9.1/webrick/utils.rb:73:in `block in create_listeners'
  from /usr/lib/ruby19/1.9.1/webrick/utils.rb:70:in `each'
  from /usr/lib/ruby19/1.9.1/webrick/utils.rb:70:in `create_listeners'
  from /usr/lib/ruby19/1.9.1/webrick/server.rb:74:in `listen'
  from /usr/lib/ruby19/1.9.1/webrick/server.rb:62:in `initialize'
  from /usr/lib/ruby19/1.9.1/webrick/httpserver.rb:24:in `initialize'
```

```
  from /home/manveru/c/rack/lib/rack/handler/webrick.rb:9:in `new'
  from /home/manveru/c/rack/lib/rack/handler/webrick.rb:9:in `run'
  from /home/manveru/c/innate/lib/innate/adapter.rb:66:in `start_webrick'
  from /home/manveru/c/innate/lib/innate/adapter.rb:40:in `start'
  from /home/manveru/c/innate/lib/innate.rb:139:in `start!'
  from /home/manveru/c/innate/lib/innate.rb:135:in `start'
  from start.rb:10:in `<main>'
zsh: exit 1     ruby start.rb
```

This means that you already have a server running on this port, and you will have to use another port to run your application or shutdown the application occupying the port.

If you don't know what is running on this port, you can find out by using the *netstat* command.

The invocation of *netstat* differs on Windows an Linux, we'll show Linux first.

```
delta ~ % netstat -lntp
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
PID/Program name
tcp        0      0 0.0.0.0:7000            0.0.0.0:*               LISTEN
11910/ruby
```

With this information you know the Process ID, and can, for example `kill 11910` (send TERM signal to the process) or `pkill ruby` (send TERM signal to all ruby processes) on Linux.

On Windows, *netstat* has to be called like this:

```
C:\> netstat -a -b -o -p TCP
Active Connections
 Proto  Local Address          Foreign Address        State           PID
 TCP    localhost:7000         localhost:0            LISTENING       17806
[ruby.exe]
```

Now you can use `taskkill /PID 17806` or `taskkill /IM <process_name>` to get rid of this process.

3. The ramaze command

Ramaze ships with an executable called *ramaze*. This little tool helps you creating, controlling, and debugging your applications.

It tightly integrates with both Ramaze and the Rack executable *rackup*, so you will be able to take advantage of both.

From now on we will call the *ramaze* executable *bin/ramaze*, that's what it is commonly called in the Ramaze community as well.

Many commands of *bin/ramaze* rely on a so-called pidfile, which is a tiny file that only contains the process-ID of a currently running Ramaze application. The name of this file defaults to the name of the directory your application is in, suffixed with *.pid*. So if you made an application called *blog* in a *blog* directory, the pidfile will be called *blog.pid*. If you see such a file, it usually means that the application is already running in the background. You can see whether that is the case:

```
delta ~ramaze/blog % ps `cat blog.pid`
  PID TTY      STAT   TIME COMMAND
32578 ?        SNl    0:00 /usr/bin/ruby /home/manveru/bin/rackup config.ru -P
```

```
blog.pid -D
```

Future versions of `bin/ramaze` might provide a `ramaze status` command, that will show you similar information.

3.1. Create

To create a new application with a basic set of files and directories that get you started in no time, we just have to issue a single command.

`ramaze create PROJECT` creates a new prototype Ramaze application in a directory named *PROJECT* in the current directory. `ramaze create foo` would make *./foo* containing an application prototype. Rack options are meaningless here.

```
$ ramaze create blog
```

What it does is simply copying a bunch of files from *lib/proto* in the ramaze library to the argument given. So you will end up with a new directory called *blog* which contains a bunch of files.

The contents of this directory are:

```
blog
|-- config.ru
|-- controller
|   |-- init.rb
|   `-- main.rb
|-- layout
|   `-- default.xhtml
|-- model
|   `-- init.rb
|-- public
|   |-- css
|   |   `-- screen.css
|   |-- dispatch.fcgi
|   |-- favicon.ico
|   |-- js
|   |   `-- jquery.js
|   `-- ramaze.png
|-- spec
|   `-- main.rb
|-- start.rb
`-- view
    `-- index.xhtml
```

We will take a look at these files and their purpose soon, but first we'll cover other commands of *bin/ramaze*.

3.2. Start

Start an instance of your application with `ramaze start`. Supply a pidfile name if you do not want it to use the default (PROJECT.pid).

```
$ ramaze start
```

To start your application in the background you should daemonize it with the `-D` argument. This will

put the PID of the instance into the pidfile. Please note that this is not available on Windows, because of limitations of the platform.

```
$ ramaze start -D
```

3.3. Stop

Stop a running instance of this application with `ramaze stop`. Supply a pidfile name if you started it with a pidfile other than the default (PROJECT.pid).

3.4. Restart

Stop a running instance of this application, then starts it back up with `ramaze restart`. Pidfile (if supplied) is used for both stop and start.

```
$ ramaze restart -D
```

3.5. Console

Starts an irb console with app.rb (and irb completion) loaded. This command ignores rack options. At the moment, you will have to call `Ramaze.setup_dependencies` after starting the session to get full functionality, as it will not actually start your application, this should be fixed in the future. We might add support for hooking into an already running application (as that would be vastly superior for live debugging).

```
$ ramaze console
```

An example session:

```
MainController.instance_methods(false)
self
# main
Ramaze.setup_dependencies
# [Innate::Cache, Innate::Node, Ramaze::Controller, Ramaze::Plugin]
MainController.instance_methods(false)
Ramaze::Mock.get('/notemplate').headers
# {"Content-Type"=>"text/html", "Content-Length"=>"994"}
Ramaze::Mock.get('/notemplate').status
# 200
```

Middleware

Ramaze relies to great extend on so-called middleware, that are small pieces of code chained together with a simple interface to handle requests and serve responses. The concept of middlewares was introduced by WSGI, and consequently also used in the design of Rack. There are already lots of middlewares available for different purposes, from serving static files to handling authentication.

1. Building middleware

An example for a middleware would be:

```ruby
class Smile
  def initialize(app)
    @app = app
  end

  def call(env)
    status, header, body = @app.call(env)
    header['Content-Type'] = 'text/plain'
    header['Content-Length'] = '2'
    return status, header, [':)']
  end
end
```

This discards any response body it receives and replaces it with a smiley. Of course that's a rather silly example, but it shows you the basic structure. A middleware has to respond to `#call`, just like a Proc. That's the reason why a small functional application you can run on Rack may even look like following:

```ruby
app = lambda{|env|                                                  # <1>
  [200,                                                             # <2>
    {'Content-Type' => 'text/plain, 'Content-Length' => '2'},       # <3>
    [':)']] }                                                       # <4>

Rack::Handler::WEBrick.run(app, :Port => 7000)                      # <5>
```

In this case we don't even bother with an inner application, so this is not a middleware but an endpoint. Using the curry functionality of Ruby 1.9, we can even make a middleware out of a normal Proc.

```ruby
hello = lambda{|env| [200, {}, ['Hello', 'World', '!']] }

reverse_middleware = lambda{|app, env|
  status, header, body = app.call(env)
  new_body = []
  body.each{|a| new_body << a.to_s.reverse }
  new_body.reverse!
  [status, header, new_body]
}

app = reverse_middleware.curry[hello]

Rack::Handler::WEBrick.run(app, :Port => 7000)
```

This would return a body containing *!dlroWolleH*, it's almost equivalent with the `Smile` middleware we built above, but works completely without creating a class.

Note Yukihiro Matsumoto sees the Proc#curry method as a curiosity at best. I don't think I can agree with that.

2. Ramaze modes

As said above, Ramaze uses a lot of middleware to provide flexible functionality that can be maintained, improved, or replaced independently without even touching the Ramaze codebase.

The simplest way to do this is to change the so-called `mode` Ramaze runs in. A `mode` describes a set of middlewares and gives it a unique name.

At the moment there are only two modes builtin: `:live` and `:dev`, the `:dev` mode is the default and should be changed upon (or before) deployment as it has a huge impact on the overall performance of Ramaze.

This option can be changed via `Ramaze.options.mode`, changing it will trigger an internal rebuild of the Dispatcher, so you can switch between modes during runtime, although this might be used only rarely it can be useful for example to adapt your application to peak-load by changing to a more aggressive caching strategy.

It is recommended that you maintain your own set of middlewares for larger applications so you can influence the behaviour better. We will show later how to do this in just a few lines of code.

The harder part of using your own set of middlewares is in figuring out which functionality you need and in which order it should be applied, something we will try to help you with as well.

2.1. The :dev and :live modes

In `:dev` mode, short for *development mode*, Ramaze will try to make your developing as effortless as possible. It automatically reloads your source code in the background on requests, and validates your application's behaviour against the Rack specification using `Rack::Lint`.

This may not sound like much, but `Rack::Lint` alone usually causes a major impact in performance (your application might run around 2-3x slower). Once your application is developed, you can disable this functionality safely.

In case you do some really crazy things with Ramaze, you should implement this while using the Lint to ensure conformance to the spec and avoid trouble during deployment. Once you are ready to deploy, Lint will have made sure that you can do safely and won't hit any problems.

Although the source-code reloaders are built to perform as well as possible, they still have a noticeable impact and increase the amount of disk I/O significantly. You certainly don't want to have your hardware die earlier on you than necessary, so we recommend to avoid source-reloading when you don't use it.

The middlewares used during `:dev` mode are:

- `m.use Rack::Lint`
- `m.use Rack::CommonLogger, Ramaze::Log`
- `m.use Rack::ShowExceptions`
- `m.use Rack::ShowStatus`
- `m.use Rack::RouteExceptions`
- `m.use Rack::ConditionalGet`
- `m.use Rack::ETag`
- `m.use Rack::Head`
- `m.use Ramaze::Reloader`
- `m.run Ramaze::AppMap`

The middlewares used during `:live` mode are:

- `m.use Rack::CommonLogger, Ramaze::Log`
- `m.use Rack::RouteExceptions`
- `m.use Rack::ShowStatus`
- `m.use Rack::ConditionalGet`
- `m.use Rack::ETag`
- `m.use Rack::Head`
- `m.run Ramaze::AppMap`

2.2. Your own modes

As mentioned above, add your own modes is a piece of cake once you know how it's being done. Let me illustrate that with a little example of an anonymous mode.

```
Ramaze.start do |mode|
  mode.use Rack::CommonLogger
  mode.use Rack::ShowStatus
  mode.use Rack::RouteExceptions
  mode.use Rack::Head
  mode.use Rack::ETag
  mode.use Rack::ConditionalGet
  mode.use Rack::ContentLength
  mode.run Ramaze::AppMap
end
```

This is an exact replica of the `:live` mode, but now you can shuffle middlewares around, insert other ones you like or need, or remove some you won't need.

If you want to keep a couple of middlewares handy for fast switching you can give them names, that's just as easy, I'll skip the actual mode of the block, it's identical to the above example using `Ramaze::start`:

```
Ramaze.middleware :moon do |mode|
  # ...
end

Ramaze.middleware :earth do |mode|
  # ...
end

Ramaze.middleware :mars do |mode|
  # ...
end

Ramaze.options.mode = :earth
```

Now that we have multiple modes in place we can freely switch between them and tweak each as we need.

3. Rack middleware

The two main repositories with middlewares ready to use are the rack on Rubyforge and rack-contrib on Github projects. Obviously, all the middlewares in Rack are available to you out of the box, as Ramaze depends on it, but many more are in rack-contrib, and you can simply install the *rack-contrib* gem and `require 'rack/contrib'` in your application.

As the number of middlewares is constantly growing, I'll just present you with some that are commonly used, and leave it up to you to discover more.

3.1. Rack::Lint

Lint validates your application and the requests and responses according to the Rack spec. The current Rack spec can always be found on the Rack homepage or may be generated by issuing `rake SPEC` in the Rack source.

The spec covers aspects like valid response body, correct response status, proper response headers, etc. The full spec with some annotations can be found in the Rack chapter.

3.2. Rack::CommonLogger

The CommonLogger middleware forwards every request to the given app, and logs a line in the Apache common log format to the given logger, or `rack.logger` by default.

In Ramaze we pass `Ramaze::Log`, an instance of `Ramaze::LogHub`, so we integrate cleanly with any middleware that is not aware of Ramaze logging.

An example of the log output would be:

```
127.0.0.1 - - [29/Apr/2009 18:57:40] "GET /admin/edit/proto%2Fwelcome HTTP/1.1"
500 230231 0.6686
127.0.0.1 - - [29/Apr/2009 18:57:42] "GET / HTTP/1.1" 200 1301 0.0831
127.0.0.1 - - [29/Apr/2009 18:57:44] "GET /favicon.ico HTTP/1.1" 304 - 0.0070
127.0.0.1 - - [29/Apr/2009 18:57:47] "GET /test HTTP/1.1" 200 1301 0.1751
127.0.0.1 - - [29/Apr/2009 18:58:46] "GET / HTTP/1.1" 200 1301 0.0531
127.0.0.1 - - [29/Apr/2009 18:58:46] "GET / HTTP/1.1" 200 1301 0.0710
```

Configuration

Configuration is a big aspect of Ramaze, although it tries to offer your sane defaults out of the box, it still gives you many ways to influence just about every behaviour.

Most commonly, you won't have to configure that much, Ramaze follows the "convention over configuration" principle to some extent, but never forces a particular way of doing things on you.

The center of all configuration is in `Ramaze.options`, an instance of `Ramaze::Options`. The configuration is self-documenting, by requiring a bit of description when you first define an option. You may use this also to add custom options to your own application, common usage of this would be credentials for an administrator, SMTP options for a mailer, or things like a page title or number of items displayed on the front-page.

I will first introduce you to some options that are most used in Ramaze, changing the web-server and port, changing your caching backend, and cookie expiration times.

There is much more that we don't cover here, I invite you to a short IRB session that shows you how to find out more.

```
Ramaze.options.each_option{|key, value| puts("%-20s: %s" % [key, value[:doc]]) }
response            :
setup               : Will send ::setup to each element during Innate::start
cache               : Innate::Cache::Memory
session             :
log_hub             :
roots               : The directories this application resides in
helpers_helper      :
trap                : Trap this signal to issue shutdown, nil/false to disable
trap
adapter             :
views               : Directories containing the view templates
middleware_compiler : The compiler for middleware
app                 :
prefix              : Prefix used to create relative links
started             : Innate::start will not start an adapter if true
layouts             : Directories containing the layout templates
mode                : Indicates which default middleware to use, (:dev|:live)
publics             : The directories containing static files to be served
```

Quite the information overload to get started, so we'll pick a few options for further inspection.

1. Basic options

Something you will notice is that some options don't have a documentation, this is because they were automatically inserted and are actually their own instance of `Ramaze::Option` living on another class. So the `:session` lives in `Ramaze::Session::options`, `:adapter` is on `Ramaze::Adapter::options`, and so on.

```
adapter = Ramaze.options.adapter

adapter.port # => 7000
adapter.get(:port) # => {:value=>7000, :doc=>"Port for the server"}

adapter.handler # => :webrick
adapter.get(:handler) # => {:value=>:webrick, :doc=>"Web server to run on"}
```

Now, that was not so bad. How about inspecting some of the options directly on `Ramaze::options` now.

```
options = Ramaze.options

options.get(:cache) # => {:names=>{:value=>[:session, :view], :doc=>"Assign a
cache to each of these names on Innate::Cache::setup"}}
options.get(:layouts) # => {:value=>["layout"], :doc=>"Directories containing the
layout templates"}
options.get(:middleware_compiler) # => {:value=>Ramaze::MiddlewareCompiler,
:doc=>"The compiler for middleware"}
options.get(:prefix) # => {:value=>"/", :doc=>"Prefix used to create relative
links"}
options.get(:publics) # => {:value=>["public"], :doc=>"The directories containing
static files to be served"}
options.get(:roots) # => {:value=>["."], :doc=>"The directories this application
```

```
resides in"}
options.get(:started) # => {:value=>false, :doc=>"Innate::start will not start an
adapter if true"}
options.get(:trap) # => {:value=>"SIGINT", :doc=>"Trap this signal to issue
shutdown, nil/false to disable trap"}
options.get(:views) # => {:value=>["view"], :doc=>"Directories containing the view
templates"}

setup = options.get(:setup)
setup[:doc] # => "Will send ::setup to each element during Innate::start"
setup[:value] # => [Innate::Cache, Innate::Node, Ramaze::Controller,
Ramaze::Plugin]

mode = options.get(:mode)
mode[:doc] # => "Indicates which default middleware to use, (:dev|:live)"
mode[:value] # => :dev
```

OK, enough of that, let's see how we can change some of these options:

```
Ramaze.options.adapter.port # => 7000
Ramaze.options.adapter.port = 8080
Ramaze.options.adapter.port # => 8080

Ramaze.options.mode # => :dev
Ramaze.options.mode = :live
Ramaze.options.mode # => :live
```

Not so hard as well.

As you can see, options are laid out in a tree structure, where options can be nested within options. Ramaze uses this functionality to keep the configuration for specific parts of the source close to the source itself, instead of defining everything in one place and then sending you around hunting for the places where that option might be used (that was the way we did it before 2009.04).

Sessions

1. What are sessions

The term session is used for data associated with a specific client (browser). The most common way is to send a cookie to the browser, which sends it back on every request. This enables us to keep client state between requests.

Sessions are an essential part of most dynamic web applications, so Ramaze tries to make it as comfortable as possible to use them to their full potential.

In this chapter we will see how and when sessions are initialized, and how you can use them in your Controllers, Views, and Models. We will also go into some of the lower-level parts in Rack, which will show you how you can manage cookies yourself, which might make sense in very lightweight applications.

As usual, we start with a small example before we dive deeper.

2. Usage example

In this example we show you how to use sessions within the Controller, this is the most common usage and essential for further understanding.

The example consists of a very simple counter, every time someone visits your site for the first time, it goes up, but it will not increase on subsequent visits. This is usually called a counter for unique page views.

```
require 'ramaze'

class Counter < Ramaze::Controller
  map '/'
  @@counter = 0

  def index
    "You are visitor number #{@@counter}"
  end

  private

  def count_visit
    return if session[:counted]
    @@counter += 1
    session[:counted] = true
  end

  before_all{ count_visit }
end

Ramaze.start
```

So, before every action, we call `count_visit`, which checks whether a specific value is set in the session associated with the client making the request. If you are using a normal browser, with cookies enabled, you will always get *You are visitor number 1*. However, if you disable cookies, or delete the cookie associated with localhost, or make a request with another browser (or curl), the counter will increase further.

So to cheat and increase your counter you can simply write a script that just doesn't send a cookie back to the client like following.

```
require 'open-uri'

100_000.times do
  open('http://localhost:7000/')
end
```

There are some ways to close this loophole, but for now I'll leave that as an exercise for the reader.

3. Access

You may access the session for the currently active request from anywhere in your application, either via `Trinity` or by calling `Ramaze::Current.session`.

This will come in handy, for example if you need to obtain the currently active language for a user, or

to determine whether the user has certain privileges in your Model.

As for all things related to `Trinity`, be aware that you will not be able to have access to the session if you are in another thread as Ramaze uses threads to contain state about the request/response cycle.

If you want to carry over state into a background job you have to use `Ramaze.defer` like following:

```
class Register < Ramaze::Controller
  def register
    # assume we register a new user here

    session[:email] = user.email
    session[:username] = user.name

    Ramaze.defer{
      mail(session[:email], session[:username], 'Welcome')
    }
  end
end
```

This will spawn a new Thread, and copy all thread variables over, so Ramaze will know which request/response cycle spawned the Thread and can access all of the state.

4. Cookies

Before we dive deeper into sessions we will have to understand cookies.

HTTP cookies are small text files containing data used by websites, and are stored on the user's computer.

Cookies are set when the server sends a *Set-Cookie* header in a response. Afterwards, the exact data the server sent is transmitted back to the server as the value of the *HTTP_COOKIE* in every request the client makes.

4.1. History

HTTP cookies have a long history back to the beginnings of the WWW, they were used in other contexts for applications before, but only after adoption in Netscape and Internet Explorer they opened a vast potential for web-applications as we know them today where state plays a major role.

The term "HTTP cookie" derives from "magic cookie", a packet of data a program receives but only uses for sending it again, possibly to its origin, unchanged. Magic cookies were already used in computing when Lou Montulli had the idea of using them in Web communications in June 1994. At the time, he was an employee of Netscape Communications, which was developing an e-commerce application for a customer. Cookies provided a solution to the problem of reliably implementing a virtual shopping cart.

Together with John Giannandrea, Montulli wrote the initial Netscape cookie specification the same year. Version 0.9beta of Mosaic Netscape, released on October 13, 1994, supported cookies. The first actual use of cookies (out of the labs) was made for checking whether visitors to the Netscape Web site had already visited the site. Montulli applied for a patent for the cookie technology in 1995; it was granted in 1998. Support for cookies was integrated in Internet Explorer in version 2, released in October 1995.

The introduction of cookies was not widely known to the public, at the time. In particular, cookies were

accepted by default, and users were not notified of the presence of cookies. Some people were of the existence of cookies as early as the first quarter of 1995, but the general public learned about them after the Financial Times published an article about them on February 12, 1996. In the same year, cookies received a lot of media attention, especially because of potential privacy implications. Cookies were discussed in two U.S. Federal Trade Commission hearings in 1996 and 1997.

The development of the formal cookie specifications was already ongoing. In particular, the first discussions about a formal specification started in April 1995 on the www-talk mailing list. A special working group within the IETF was formed. Two alternative proposals for introducing state in HTTP transactions had been proposed by Brian Kristol himself, soon decided to use the Netscape specifications as a starting point. On February 1996, the working group identified third-party cookies as a considerable privacy threat. The specification produced by the group was eventually published as RFC 2109 in February 1997. It specifies that third-party cookies were either not allowed at all, or at least not enabled by default.

At this time, advertising companies were already using third-party cookies. The recommendation about third-party cookies of RFC 2109 was not followed by Netscape and Internet Explorer. RFC 2019 was followed by RFC 2965 in October 2000.

HTTP_cookie
— Wikipedia

4.2. Structure

Let's inspect the headers sent back when we try to access the *index* action from the example above.

```
delta ~ % curl -I localhost:7000/
HTTP/1.1 200 OK
Content-Length: 24
Content-Type: text/html
Server: WEBrick/1.3.1 (Ruby/1.9.2/2009-03-02)
Date: Thu, 02 Apr 2009 11:17:15 GMT
Connection: Keep-Alive
Set-Cookie:
innate.sid=d686bd7b9a4b4b8ccc2cf5b1d4f6a47d315ef375acb91a73caa34a825cf19f1b;
path=/; expires=Tue, 19-Jan-2038 03:14:07 GMT
```

You see that Ramaze sends the *Set-Cookie* header automatically because we try to store data in the session in our Controller.

The value of the *Set-Cookie* header consists of `key=value` pairs, where value usually contains following elements:

domain	Domain that set the cookie
path	Relative path below which the cookie is used
expires	Point in time when the cookie expires. If no time is given then the cookie will be deleted once the browser closes.
secure	**TODO**

4.3. Usage

Now that we have a better knowledge of what cookies are and what they are supposed to do, let's see how to use them within Ramaze.

```
require 'ramaze'

class CookieCounter < Ramaze::Controller
  map '/'

  def index
    "This is your visit number #{@count}"
  end

  private

  def count_visit
    counter = request.cookies['counter'].to_i
    @count = counter + 1
    response.set_cookie('counter', @count)
  end

  before_all{ count_visit }
end

Ramaze.start
```

In this example we don't count unique page views, but rather keep track of the number of visits a single browser has made so far.

We are using methods provided by Rack, you can find them in `Rack::Request` and `Rack::Response`. Ramaze uses the same methods to set cookies for sessions.

`Rack::Respone#set_cookie` also allows a Hash as argument for the additional key/value pairs that we showed above.

```
set_cookie('counter',
           :domain => nil,
           :path => '/',
           :secure => true,
           :expires => (Time.now + 60 * 60),
           :httponly => false,
           :value => 1)
```

Note The only required argument is `:value`, all the others are optional and the cookie is valid without them.

Usually you won't need to care about cookies at all, and excessive usage of them is considered bad practice.

4.4. Limitations

There are several limitations imposed on cookies.

According to the HTTP cookie specifications, browsers should be able to store a minimal amount of cookies. In particular, an internet browser is expected to be able to store at least 300 cookies of four kilobytes each, and at least 20 cookies per server or domain.

Since around 2007, all major browsers store around 30-50 cookies per domain. But the size of each cookie still has severe limitations, Internet Explorer for example will limit the total size of all cookies for one domain to 4 kilobytes.

So it makes sense to utilize Sessions instead of a large amount of cookies if you want to store arbitrary amounts of data without worrying about it, all that is stored on the client is an UUID that associates their cookie with data stored on the server.

5. Configuration of session cookies

Now that we have a better understanding about cookies, let's investigate the way Ramaze utilizes cookies for the session, in particular how to change the default behaviour through configuration.

The configuration for sessions is kept at `Ramaze::Session.options`, which is also reachable through `Ramaze.options.session`.

The defaults are:

Key for the session cookie	`:key`	*innate.sid*
Domain the cookie relates to	`:domain`	`false`
Path the cookie relates to	`:path`	/
Use secure cookie	`:secure`	`false`
Time of cookie expiration	`:expires`	Tue Jan 19 12:14:07 +0900 2038

Please note that there is no way (yet) to disable sessions. If you do not want to have them, don't use them.

This contrasts with earlier releases of Ramaze, where sessions were always created as soon as a request was made unless you explicitly disabled sessions.

Ramaze since version 2009.04 initializes sessions in a lazy manner, only setting a session cookie the first time you explicitly store a value in the session. You can still try to read a value from an uninitialized session, which simply returns nil and won't initialize a session cookie either.

5.1. Expiration

The most important option here might be the time of expiration.

By default Ramaze uses the highest allowed value for a Time instance on 32bit systems, which is some

time in the year 2038.

The expiration time also decides the time-to-live of the server-side value of the session, which is expired once the cookie is set to expire.

So in order to keep your server from taking up too much memory when you expect a lot of unique visitors it is recommended to set this to a more reasonable value depending on your application.

6. Concurrency

A common problem when it comes to utilizing sessions is concurrency. If you have more than one instance of your application running to handle requests via a load-balancer it gets quite hard to predict which instance will handle requests for which client, and browsers may end up having multiple active sessions within the same application.

By default, sessions are stored in a conventional Hash that is kept in memory, this is the fastest way and allows for arbitrary data to be kept in the session value, such as instances of Proc or objects with a custom meta-classes.

However, this flexibility has major drawbacks when it comes to scaling your application, so you should avoid storing anything that cannot be fully serialized from the get-go to avoid problems down the road.

Ramaze provides multiple ways to persist your session beyond the boundaries of your application's memory. You may use the `Marshal`, `YAML`, `DRb`, `Memcached`, or `Sequel` caches that ship with Ramaze or you may even create your own Cache, which is covered in detail in the chapter about caches.

The default method of caching is in-memory, in a conventional Hash. To change this, you have to modify the options at `Ramaze::Cache.options`, also available at `Ramaze.options.cache`.

For example, to persist sessions to the different caches (useful if you want to share them between multiple instances), you can do following:

```
# Using Marshal serialization to the file-system
Ramaze.options.cache.session = Ramaze::Cache::Marshal

# Using YAML serialization to the file-system
Ramaze.options.cache.session = Ramaze::Cache::YAML

# Using DRb serialization to a shared object
Ramaze.options.cache.session = Ramaze::Cache::DRb

# Using Memcached serialization to a cloud of shared servers
Ramaze.options.cache.session = Ramaze::Cache::Memcached

# Using Sequel serialization to the file-system
Ramaze.options.cache.session = Ramaze::Cache::Sequel
```

Each of these cache back-ends has unique strengths and weaknesses, please read more about them in the chapter about caches.

7. Flash

The session flash has nothing to do with Adobe Flash ®.

It is best described as a special-purpose Hash that expires parts of it's contents after every request/response cycle.

The purpose of this functionality might be best explained as a way to display the results of some action a user invokes in the action coming after it.

So let's say an user tries to access a restricted area, and you simply want to redirect the user elsewhere, displaying a message that the access was denied.

```
class RestrictedArea < Ramaze::Controller
  def index
    do_some_important_stuff
  end

  before_all do
    unless user_is_privileged?
      session[:error_message] = "Access denied"
      redirect_referer
    end
  end
end

class NormalArea < Ramaze::Controller
  def index
    %q(
<?r if error_message = session.delete(:error_message) ?>
  <div class="error">#{h error_message}</div>
<?r end ?>
#{a('restricted area', RestrictedArea.r)}
    )
  end
end
```

Given that the user clicks the link in `NormalArea` and tries to access `RestrictedArea`, an error message will be displayed after the redirect, and the message is deleted from the session so it will not show up on the next request.

Flash basically automates the deletion, and acts as a short-term memory, so you never have to remember to delete the contents.

Using the default helper, `Helper::Flash`, our example would look like this:

```
class RestrictedArea < Ramaze::Controller
  def index
    do_some_important_stuff
  end

  before_all do
    unless user_is_privileged?
      flash[:error] = "Access denied"
      redirect_referer
    end
  end
end

class NormalArea < Ramaze::Controller
  def index
    %q(
```

```
#{flashbox}
#{a('restricted area', RestrictedArea.r)}
    )
  end
end
```

It might not be shorter in this case, but over a larger application this simplicity comes in handy.

Ramaze also ships with a `Helper::Flash#flashbox`, which automates the display of flash messages as they are used in this example.

The output of flashbox looks like this:

```
<div class="flash" id="flash_error">Access denied</div>
```

Also, this default representation can be changed easily, but more about this in the section about the flash helper.

7.1. Implementation

The implementation of flash functionality is quite simple, when you are accessing `Session#flash`, a Hash will be stored in `session[:FLASH]`. This is the location where all key/value pairs you assign to `Session#flash` will be stored.

After every request that used a session, `Session::Flash#rotate!` will be called and will move `session[:FLASH]` to `session[:FLASH_PREVIOUS]`. When you inspect the flash, it will show a merged Hash consisting of `:FLASH` and `:FLASH_PREVIOUS`, with `:FLASH` taking precedence.

This way, after two requests, values disappear forever unless reassigned.

Helpers

Helpers are modules for inclusion into, and extension of controllers or other classes.

The distinguishing feature of helper modules is that they are expected to be in the `Ramaze::Helper` name-space. There is also a naming-convention of helper modules and the files they reside in to allow automatic requiring. These constrains allow you to use any helper without prior configuration.

1. Usage

To tell Ramaze that you want to use a certain helper in your Controller, you can simply use the `helper` method. This method takes one or more arguments, every argument has to be the snake-cased name of a Helper module.

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

  def index
    number_format(rand)
  end
end
```

In this case we include the `Ramaze::Helper::Formatting` module into `MainController`.

Ramaze will first require a file named `ramaze/helper/formatting.rb`, then search the `Ramaze::Helper` name-space for the module called `Formatting`, and eventually include it.

Equivalent behaviour can be achieved with:

```
require 'ramaze/helper/formatting'

class MainController < Ramaze::Controller
  include Ramaze::Helper::Formatting

  def index
    number_format(rand)
  end
end
```

But once you start using multiple helpers this becomes very verbose. So using the `helper` method is recommended.

2. Making your own

Creating helpers is simple.

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

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

Just put this into a file in the root of your application, named *helper/smiley_helper.rb*, and you can use it in your Controller with `helper :smiley_helper`.

3. Naming

Helpers have to follow a specific naming convention for the respective file they are defined in. Ramaze will not require all helpers available on startup, since some of them may have external dependencies that are not relevant to your application, this also helps keeping the used memory lower.

In Ruby, a module is assigned to a constant, which should be named using camel-case. However, the common convention to name files after the class they contain, is to write the same name in snake-case. Ramaze follows these conventions, module-names are camel-case, while their filenames are snake-case.

Ramaze will search the `Ramaze::Helper` module for module-names in a case-insensitive manner and simply chooses the first it can find. This means that, if you have a helper called `AB` and another one named `Ab`, it's usually down to luck which one you will get. The reasoning behind this behaviour is

that helpers like `CGI` would be impossible to find. We simply rely on the fact that, if you ask for the `cgi` helper, only the first `cgi.rb` will be required, and so only one Helper matching `/^cgi$/i` exists.

4. Helper file lookup

Ramaze will search a few paths to find your helper. We well call this the Helper PATH. By default, there are three directories (and in this order) searched: *./helper/*, *lib/innate/helper/*, and *lib/ramaze/helper/*.

Note The paths to Innate and Ramaze helpers are in fact absolute, but since I don't know where they are on your machine, I simply show the known part.

You can modify the Helper PATH via the `paths` option on `Innate::HelpersHelper`. The final */helper* part of the path is appended on requiring, the PATH itself only specifies the directory the */helper* directory is in.

```
hh = Innate::HelpersHelper
hh.options.paths[0] # => "/home/manveru/c/ramaze-book"
hh.options.paths[1] # => "/home/manveru/c/innate/lib/innate"
hh.options.paths[2] # => "/home/manveru/c/ramaze/lib/ramaze"
```

5. Methods during action creation

There is a clear distinction between normal modules and helper modules in the way Ramaze treats helper modules during action creation.

Methods of helpers are not considered for actions unless they are public and the module was added to the `Ramaze::Helper::LOOKUP` Set.

By default, this Set is empty, and you will not have to care about this too much in your application, as the helpers that provide real actions add themselves as needed.

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.

Note Usually you will want to add a call to `redirect_referrer` in this method, but we keep it simple here, the actual functionality doesn't matter.

6. Default helpers

Ramaze comes with a small selection of default helpers deemed essential for any application, they are kept in the Innate source. The major difference to non-default helpers is that you will not have to explicitly state their usage, they will included in your Controller automatically.

The helpers available by default are:

- Aspect
- CGI
- Flash
- Link
- Redirect
- Render
- SendFile

We will cover each of them in more detail below.

At the time of writing, there are also around 30 non-default helpers in the Ramaze source. Let us take a look at the most popular ones.

Some of the helpers in Ramaze add functionality to the default helpers of Innate, I will simply cover the combined behaviour in these cases, as you are most likely using the ones in Ramaze.

7. Aspect helper

The Aspect helper provides you with functionality known from AOP (Aspect Oriented Programming). Sometimes you just want to execute some code around the execution of an Action without the Action being aware of it. Examples for this include authentication, logging, persistence, and routing.

The helper provides you with four methods, symmetrical in functionality.

7.1. Using before_all

So you have a Controller that requires a user to be administrator in order to be allowed to do anything.

This might look like:

```
require 'ramaze'

class AdminController < Ramaze::Controller
  def create_user(name)
    # ...
  end

  def delete_user(name)
    # ...
  end

  before_all{ redirect_referrer unless user_is_root? }

  private
```

```
  def user_is_root?
    session[:user_is_root]
  end
end

Ramaze.start
```

8. CGI helper

The CGI helper provides shortcuts to methods commonly used to deal with escaping and unescaping text for different contexts.

The name originated from the CGI module it utilized to provide it's functionality. Today there is only one method that isn't commonly used, left that calls the CGI module.

There are four symmetric methods to deal with escaping and unescaping HTML and URIs, and one method to escape text for usage in templates.

In common applications, the most prevalent method might be `h`, an alias for `html_and_code_escape`.

All methods of the CGI Helper may be accessed directly as module-functions as well.

8.1. Encoding text for use in URIs with url_encode

The set of allowed ASCII characters in an URI is very limited. When you want to create an URI, you have to take special care to avoid using characters that have special meaning and encode them properly.

For example, let's see how we can put *Innate & Ramaze* into a query parameter in the following IRB session:

```
name = Ramaze::Helper::CGI.url_encode('Innate & Ramaze')
name # => "Innate+%26+Ramaze"
uri = URI("http://google.com/search?q=#{name}")
uri # => #<URI::HTTP:0xf77f9980 URL:http://google.com/search?q=Innate+%26+Ramaze>
uri.to_s # => "http://google.com/search?q=Innate+%26+Ramaze"
```

8.2. Decoding text from URIs with url_decode

This is simply the reverse operation of `url_encode`, so we will use the output of the previous example as input.

```
encoded = Ramaze::Helper::CGI.url_encode('Innate & Ramaze')
encoded # => "Innate+%26+Ramaze"

decoded = Ramaze::Helper::CGI.url_decode(encoded)
decoded # => "Innate & Ramaze"
```

9. Flash helper

The naming of this helper might be confusing at first, it doesn't have anything to do with Adobe Flash ®.

It provides two methods, `flash` and `flashbox`.

The `flash` method is a shortcut to `Ramaze::Current.session.flash`, you can find more in-depth information in the chapter about `Ramaze::Session`.

10. Link helper

Linking and obtaining routes

11. Redirect helper

Redirect or respond immediately

12. Render helper

Render other actions

Testing

1. About testing

There is a large movement in the Ruby community that advocates testing applications and libraries. This chapter tries to cover most of the aspects of testing your Ramaze applications. We will see things like the history of testing with Ramaze, browser automation, XPATH assertion, and why you should care.

I need to clarify my terms for this chapter, with *tests* and *specs* I basically mean the same thing, asserting some kind of functionality or behaviour. The advocates for specs will usually try to make the distinction seem larger than it really is, all it comes down to is some changes in terminology and syntax.

The larger difference is between testing after the fact and so-called BDD, short for *Behaviour Driven Development*. For the prior one you simply write your code first and assert that it works later, in the latter case you formulate your expectations first, and then go on to write the least amount of code possible to make the expectation go green.

For the rest of this chapter, I will be using *specs* for an expectation in the BDD sense of writing a test before the code it tests is actually implemented, and *tests* for everything else.

My personal take on whether to use BDD is flexible, depending on the project. If you know the kind of output you can expect when your code is finished, e.g. if it's not something experimental, take the BDD approach, jot down the basics and then just make your specs pass. However, sometimes what you are doing will just be novel to you, and when you start out you will have no idea where you will end up. In this case it's probably better to start with writing small examples that are easy to change and don't have any fixed expectations. But make sure to actually write tests afterwards to avoid regressions.

I think the biggest value of testing is exactly this: avoiding regressions. It is very easy to go on a bug-hunt for a couple of hours, fix a bug and then skip the write-a-test part, and while this might work for smaller projects, you will be hating yourself once it grows or should be integrated into a larger system.

Take care of your future self, it doesn't have as many nerves as you, and it might have a time-machine.

2. Ramaze and testing

Since soon after Ramaze was created, most of it's functionality was tested regularly.

Initially, since I had limited experience with the whole concept of testing, it was not something I did for every feature and change, more like a guideline and help to avoid regression of functionality.

For the first year, I used the RSpec framework to write and run tests, mostly because it wasn't hard on my eyes like the Test::Unit library that ships with Ruby.

I found several problematic things with RSpec, it was very slow due to it's size, if you look at it today it has around five thousand lines of code, most of which will have to be parsed and executed for every single spec, in the early days it also had a very instable API, which made my specs break, even though my code didn't change at all.

Eventually I grew tired of fixing my code just to adapt it to their fancy new ways of doing things, back then the whole spec movement was really new and most of what they did had never been done before, so I understand their decision to keep moving forward, and today they still play a major role in the Ruby community. I just was not interested in features I was not going to use and minor changes in terminology and syntax. What I had worked and looked nice already.

At this point I would like to thank the people behind RSpec, without them Ramaze wouldn't be what it is today.

Late in 2007, Christian Neukirchen started work on a tiny RSpec clone called Bacon, which I found appealing immediately and which eventually replaced the role RSpec had within the Ramaze project.

We will cover more about Bacon later on.

3. Testing your application

This chapter should cover

- specs vs. tests

4. Bacon

Bacon is a small RSpec clone weighing less than 350 lines of code but nevertheless providing all essential features.

Truth will sooner come out from error than from confusion

— Francis Bacon

It had a small core of functionality, just enough to make specs look appealing and readable. The syntax was resembling RSpec closely, but it left out things that I didn't care about anyway. Thanks to that similarity it was a breeze to port my existing specs over to Bacon.

Since it was very small, and no release had been made at that point, I simply put it into our Ramaze repository and shipped it until around April of 2009.

Bacon has long since reached version 1.0 and remained as small as ever, Ramaze doesn't ship it anymore, but relies on it as a development-dependency.

4.1. Usage

Using Bacon on it's own is very simple, let's write a little spec for a method we want to implement.

```
require 'bacon'

Bacon.summary_on_exit

def sum(*args)
end

describe 'sum' do
  it 'sums arguments' do
    sum(1, 2, 3, 4).should == (1 + 2 + 3 + 4)
  end
end
```

```
require 'bacon'

Bacon.summary_on_exit

def sum(*args)
  args.inject(:+)
end

describe 'sum' do
  it 'sums arguments' do
    sum(1, 2, 3, 4).should == (1 + 2 + 3 + 4)
  end
end
```

```
require 'bacon'

Bacon.summary_on_exit

def sum(*args)
  args.inject(:+)
end

def average(*args)
end

describe 'sum' do
  it 'sums arguments' do
    sum(1, 2, 3, 4).should == (1 + 2 + 3 + 4)
  end
end

describe 'average' do
  it 'calculates average of arguments' do
    average(1, 2, 3, 4).should == (1 + 2 + 3 + 4) / 4.0
  end
end
```

```
require 'bacon'

Bacon.summary_on_exit
```

```ruby
def sum(*args)
  args.inject(:+)
end

def average(*args)
  args.inject(:+) / args.size.to_f
end

describe 'sum' do
  it 'sums arguments' do
    sum(1, 2, 3, 4).should == (1 + 2 + 3 + 4)
  end
end

describe 'average' do
  it 'calculates average of arguments' do
    average(1, 2, 3, 4).should == (1 + 2 + 3 + 4) / 4.0
  end
end
```

4.2. Shared contexts

…

5. Innate::Mock

…

6. Rack::Test

…

7. Hpricot

…

8. Webrat

…

9. Mechanize

…

10. Examples

…

10.1. GET

…

10.2. POST

…

10.3. Multipart POST

…

10.4. Working with models

…

Actions

In this chapter we will see what role Actions play within Ramaze, as usual we will see a bit of history first, then go on to look at the structure of the Action Struct, eventually creating our own Actions.

1. History

Actions were first introduced around Summer 2007, the intention behind this change was to bundle a template, a method, and the parameters to the method into one Object that could be easily passed around and modified.

Over time, this brought with it a change of the role of the Controller, which did everything from routing to rendering and grew quite large.

Eventually, the rendering part was ripped out of Controller, and put into the Action class, and further changes made the Action pass itself to the template rendering.

The role of the Controller became to build the Action depending on the state of the Controller. The Controller would still do most of the routing, but how that changed is probably better explained in the chapter about controllers.

Today, Action still resembles the original Action from 2007, but has been expanded as there were additions like aspects and layouts.

2. Structure

The Action class is an instance of the Struct class, with following members:

engine	The templating engine called during rendering
instance	Lazily created instance of the node
layout	Array consisting of the type and name of the layout

method	Name of the instance-method called on instance during rendering
node	Class that the instance will be instantiated from
options	Hash with options and meta-information for the engine
params	Array of parameters for the instance-method used for the call
method_value	The return value of the instance-method, set during rendering
variables	Hash of instance-variables usable in the method and templates
view	String with the full path to a file used as view-template
view_value	Contents of the view-template file
wish	Indicates the requested content-representation

Most of these members are not of interest for the average user, but they become increasingly interesting if you try doing partial rendering of actions.

3. Creating actions

If you try to learn more about the internals of Ramaze, you will need to understand Actions, if you are not interested in this, feel free to skip this section.

3.1. The hard way

The theoretical knowledge from above is of course not very valuable if you don't have had any previous experience with Ramaze, so I'll give you a quick rundown to creating a simple Action for a small example.

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

  def sum(*numbers)
    numbers.inject(0.0){|sum, num| sum + num.to_f }.to_s
  end
end

index = Ramaze::Action.create(
          :node => MainController,
          :method => :index,
          :engine => lambda{|action, value| value })
index.call # => "Hello, World!"
```

Similarly we can create an Action for the `sum` Action:

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

  def sum(*numbers)
    numbers.inject(0.0){|sum, num| sum + num.to_f }.to_s
  end
end

sum = Ramaze::Action.create(
          :node => MainController,
          :method => :sum,
          :params => ['32', '8', '2'],
          :engine => lambda{|action, value| value })
sum.call # => "42.0"
```

I will not go into much detail about these examples, as I think they can speak for themselves. You will soon see more information about the methods involved in the rendering of an Action. If you want to see the Action automatically created during normal runtime of Ramaze you can use the `Current.action` or simply `action` methods in your Controller and inspect the returned value. It is also possible to change parts of the Action at this stage, allowing you to influence details of the rendering process.

3.2. The easy way

It's quite tedious to do all this work of creating actions manually when Ramaze can simply do it for you. It's just good to gain an understanding, so let's create the same actions as in the examples above, this time using the `Node#resolve` method which is also available in controllers.

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

  def index
    "Hello, World!"
  end

  def sum(*numbers)
    numbers.inject(0.0){|sum, num| sum + num.to_f }.to_s
  end
end

index = MainController.resolve('index')
index.call # => "Hello, World!"
```

And the same can be done for the `sum` Action.

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

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

```
  def sum(*numbers)
    numbers.inject(0.0){|sum, num| sum + num.to_f }.to_s
  end
end

sum = MainController.resolve('sum')
sum.params = %w[32 8 2]
sum.call # => "42.0"
```

Views

Views in Ramaze are a special category of templates. Every Controller has various paths and rules used to find templates for views. Lookup of the View is done when assembling an Action, but usually it is not necessary to actually obtain an Action, alternatively an Action method can provide a string template that is used for the rendering of the View.

1. History

As a MVC framework, Ramaze used views from the very start, and the concept has changed very little since.

Originally, Ramaze didn't have a concept of layouts, and after layouts were added, things got messy. More about this should be said in the chapter about layouts, but in essence layouts are now confined in their own lookup paths and are not considered as file templates for views anymore.

At first, view file templates were located below a directory called *template* in the application's root directory.

This was changed over time to a *view* directory in a move to streamline terminology around the templating engines.

2. Structure

By default, view file templates are located in a directory called *view* in the application root. Nested inside this directory can be further directories, named after Controller mappings, which contain further templates for that Controller. The Controller mapped to */* owns the files at the top of *view/*, but using special invocations, it may reach inside other folders below that and use templates located there.

The structure for a medium-sized application may look like this:

```
view
|-- contact.xhtml
|-- financial_report
|   `-- index.xhtml
|-- index.xhtml
|-- new.xhtml
|-- news
|   `-- index.xhtml
|-- portfolio
|   |-- edit.xhtml
|   |-- index.xhtml
|   `-- new.xhtml
`-- view.xhtml
```

This gives us information about the internal structure of the application already, there may be controllers mapped to /, */financial_report*, */news*, and */portfolio*, and all of these files represent a possible Action.

2.1. The index template

Ramaze has a convention about the naming of the / Action for every Controller, which is instead called *index* as it is quite hard to create a file called */.xhtml* (which would instead turn out to be a hidden file called *.xhtml*).

So, every time you want to have a view file template that is used in requests to, for example, */foobar*, you create a file at *view/foobar/index.xhtml*.

2.2. Faking depth

If you want to make an Action for */foobar/foo/bar/baz* but want to have the view file template provided by the Controller mapped to /, you can create a file at *view/foobar__foo__bar__baz.xhtml*, every double-underscore is translated into slashes. The same applies for names of methods of that Controller.

2.3. Inline templates

You will find that Ramaze will try to utilize the return value of the controller method as template if no file template was found. It does this by calling `#to_s` on the object returned and running the resulting String through the specified templating engine.

This kind of template is commonly called *inline template*, and it's usually what you start out with when you write a small application.

It's very easy to sketch out ideas this way, and makes it easy to prototype functionality in a single file.

I am sure that there will be some that don't hesitate writing even large applications with one of the builder templating engines and they will be content with writing their templates directly in the controller methods.

However, as your application grows, I highly recommended to separate controllers and views to avoid a maintenance nightmare.

As a rule of thumb, I start splitting if I need more than one Controller or use more than a handful of templates. Don't take this as a rule set in stone, it's simply my view of things, and everyone may use Ramaze as they see fit.

3. Path lookup

There are a number of factors that play into the lookup of views, I will analyze and explain them.

Ramaze provides powerful ways to change just about anything in your application, so it is only natural that the way it looks for view templates is just a convention by pre-configuration. After reading this section you should have enough knowledge to make Ramaze serve your purposes.

The path to any template consists of three four parts, which I will call *application mapping*, *application view mapping*, *controller view mapping*, and *view file template*.

The defaults result in a path as follows.

```
application mapping: './'
```

```
application view mapping: 'view'
controller view mapping: '/controller'
view file template: 'foo.xhtml'

./view/controller/foo.xhtml
```

The directory portions of the path may actually be each an Array of paths, providing alternative lookup paths. To illustrate this, take following path definition and the simple brute-force order in which it is searched.

```
application mapping: ['app1', 'app2']
application view mapping: ['view1', 'view2']
controller view mapping: ['controller1', 'controller2']
view file template: 'foo.xhtml'

app1/view1/controller1/foo.xhtml
app1/view1/controller2/foo.xhtml

app1/view2/controller1/foo.xhtml
app1/view2/controller2/foo.xhtml

app2/view1/controller1/foo.xhtml
app2/view1/controller2/foo.xhtml

app2/view2/controller1/foo.xhtml
app2/view2/controller2/foo.xhtml
```

Although this allows for very smart sharing of templates between controllers or even whole applications, it is also a way to shoot yourself in the foot. You should be fully aware of what you are doing and consider your choice carefully.

Ramaze ships with a tool called AppGraph, which helps you by visualizing all possible Actions in your application. This can be very valuable if you are dealing with complex lookup patterns, especially if you are dealing with an application you didn't write yourself or when aggregating different applications.

4. Configuration

The paths for view file template lookup can be easily modified. Every Ramaze App allows you to configure it's own paths as well, without affecting other applications in your site.

The following example should give you a quick overview of the path looked up for a specific Controller.

```
class Example < Ramaze::Controller
end

Ramaze.start(:started => true)

Ramaze.options.roots # => ["."]
Ramaze.options.views # => ["view"]
Example.mapping # => "/example"
Example.view_mappings # => [["view"], ["/example"]]

Example.possible_paths_for(Example.view_mappings) # => ["./view/example/"]
```

4.1. Global configuration

The default values for the *application mapping* and *application view mapping* are:

```
Ramaze.options.roots # => ["."]
Ramaze.options.views # => ["view"]
```

They will be inherited into every new Application.

If you change these defaults, it will also affect any application that didn't specify their own paths, so watch out for that.

4.2. Controller configuration

There is one method to manipulate the *controller view mapping* called `map_views`.

Please do not use absolute paths, all paths in the *controller view mapping* are relative to the *application view mapping*, which in turn is relative to the *application mapping*.

```
class Example < Ramaze::Controller
  map_views 'foo', 'bar'
end

Ramaze.start(:started => true)

Ramaze.options.roots # => ["."]
Ramaze.options.views # => ["view"]
Example.mapping # => "/example"
Example.view_mappings # => [["view"], ["foo", "bar"]]

Example.possible_paths_for(Example.view_mappings) # => ["./view/foo/",
"./view/bar/"]
```

5. Content representations

As if the above wasn't enough already, Ramaze also offers you ways to present your content in different ways for different requests, in Ramaze this functionality is called *provides*.

Relevant to the lookup of provides is the filename extension of your view file template. We will deal with this in more depth in the chapter about content representations, for now you don't have to care about it.

Layouts

Layouts in Ramaze are a special category of templates. Every Controller has paths and rules used to find templates for layouts. Lookup of the template is done when assembling an Action, but is in no way a necessary element of an Action.

1. History

Layouts were first introduced in mid-2007. At first they were simply a small addition to wrap a rendered Action into another Action. This allowed for easier use of alternative templating engines, as at

the time only the Ezamar templating engine provided equivalent functionality through Elements.

As it became evident that Elements were not as easy to work with, layouts were used increasingly. However, the functionality was still rather hard to control and not well integrated into the framework.

This was changed when Ramaze was starting to utilize Innate, and they are now well integrated and powerful citizens of Ramaze.

Since this time, layouts also have their own directory to live in.

2. Usage

This is a collection of small examples that try to give you a taste of what's possible with layouts.

Usually, there will be only one layout per controller, but you can dispatch to different layouts depending on the request. The `layout` method takes a block, which, depending on the argument to the method, behaves differently. In the normal case, you will only supply one argument and no block.

```
layout('default')
```

This will apply the same *default* layout to all actions.

```
layout('default'){|path, wish| !request.xhr? }
```

This will apply the *default* layout only to actions that were not requested via AJAX (which sets the *HTTP_X_REQUESTED_WITH* header to *XMLHttpRequest*). This can be very useful if you want to include parts of a page dynamically from JavaScript without using a separate controller.

```
layout('default'){|path, wish| wish == 'html' }
```

Now we apply a layout only if the client wished for html (through using no, or the *html* extension in the URI) So a request to */foo* or */foo.html* will have the *default* layout, but a request to */foo.json* won't. Please see the section about content representations about more information about how Ramaze handles these requests.

```
layout{|path, wish| ['red', 'blue', 'green'].choice }
```

Now we're getting even more dynamic, choosing a random layout for every request. Usually you'll want to put some reasonable logic inside the block, but the point of this example is that, if you don't pass an argument, but a block, to the `layout` method, it will use the return value of the block as the name for the layout. This can be useful if you want to let users choose a theme, or have a different layout for users that are logged in.

2.1. Layout from template

Layouts are usually read from a file template that resides in the */layout* directory. Following example will search for a file called */layout/default.xhtml* and, if found, wrap the content of the action inside it.

```
class Box < Ramaze::Controller
  layout :default

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

Additionally to this, we will need the file for the layout template to reside in, we'll use an identical layout as further above.

```
$ cat layout/default.xhtml
{ #{@content} }
```

That's just as easy, but enables you to work easier on larger layouts.

2.2. Layout from method

Following example should illustrate a simple use-case for a layout that uses the return value of a method:

```
class Box < Ramaze::Controller
  layout :default

  def index
    'Hello, World!'
  end

  def default
    '{ #{@content} }'
  end
end
```

This will wrap every request to */index* in curly brackets. Not very useful, but it should illustrate the basic working principle.

Note: If you have both a file template and a method for the layout, Ramaze will use the file template and will not call the method. In future versions this behavior might change to call the method as well.

2.3. Layout directories per controller

Additionally to the */layout* directory, sub-directories thereof can be searched for templates. This lookup is specific to each Controller. To control the behavior you can use the `map_layouts` method, which takes multiple arguments.

3. Structure

By default, layout file templates are located in a directory called *layout* in the application root. Nested inside this directory can be further directories, if you want a controller to look for layout file templates in a deeper directory, you can use the `map_layouts` method. By default all layout file templates are searched at the top.

It is not very common for applications to utilize more than a few layouts, you will find many applications with only a single layout, and even a few without any layouts at all.

The layouts of a medium-sized application may organized like this.

```
layout
|-- default.xhtml
|-- feed
|   |-- default.rss.xhtml
|   `-- default.atom.xhtml
```

```
`-- mobile.xhtml
```

It is a common pattern to name the layout used for most controllers *default*, but there is nothing enforcing it. In fact, to use a layout, you will have to explicitly tell Ramaze which to use.

4. Path lookup

To provide flexibility for large projects (and for use-cases I haven't imagined yet), I decided to reuse the view file template lookup already in use for layouts. This ensures that the behavior of both is identical and available through a consistent API.

4.1. Inline templates

If no layout file template can be found for a specified layout, a method may be used instead.

5. Configuration

To get a feeling for the places where Ramaze will search for layouts, you can inspect like in the following example.

```
class Example < Ramaze::Controller
end

Ramaze.start(:started => true)

Ramaze.options.roots # => ["."]
Ramaze.options.layouts # => ["layout"]
Example.mapping # => "/example"
Example.layout_mappings # => [["layout"], ["/"]]

Example.possible_paths_for(Example.layout_mappings) # => ["./layout/"]
```

There are two options on `Ramaze.options` that influence the lookup, one is the `roots`, which is set depending on your application, the other the `layouts` which defaults to `['/layout']`.

```
Ramaze.options.roots # => ["."]
Ramaze.options.layouts # => ["layout"]
```

6. Content representations

Layouts can also play a role in content representations, and you can add layouts for specific provides. As we saw above, we were using files like *feed/default.rss.xhtml* and *feed/default.atom.xhtml*, they apply to the `rss` and `atom` provides respectively.

Let's show this with another little example so you can learn how to use this functionality effectively.

```
class Smiley < Ramaze::Controller
  map '/'
  layout :default
  provide :frown, :engine => :Etanni
  provide :smile, :engine => :Etanni

  def index
```

```
    'emotions ftw!'
  end
end
```

So we add two provides named `frown` and `smile`, and there is one default provide for `html`. To take advantage of this, we can add three layout files in our *layout* directory.

```
$ cat layout/default.xhtml
:| #{@content} |:

$ cat layout/default.frown.xhtml
:( #{@content} ):

$ cat layout/default.smile.xhtml
:) #{@content} (:
```

Following responses will be served on requests:

/index	*:| emoticons ftw! |:*
/index.frown	*:(emoticons ftw!):*
/index.smile	*:) emoticons ftw! (:*

7. Implementation

Sometimes the Ramaze default behavior won't be what you need, so knowing how layouts are implemented can help you building custom actions with layouts tailored to your needs.

I will start with an arbitrary existing Action, then explain how it was created and how it will be rendered, finally we'll cover some more possibilities.

On every request, the layout will be determined and set in the Action, which will clone itself.

Rack spec

This specification aims to formalize the Rack protocol. You can (and should) use Rack::Lint to enforce it. When you develop middleware, be sure to add a Lint before and after to catch all mistakes.

1. Rack applications

A Rack application is an Ruby object (not a class) that responds to `#call`. It takes exactly one argument, the *environment* and returns an Array of exactly three values: The *status*, the *headers*, and the *body*.

1.1. The Environment

The environment must be an true instance of Hash (no subclassing allowed) that includes CGI-like headers.

The application is free to modify the environment.

The environment is required to include these variables (adopted from PEP333), except when they'd be empty, but see below.

REQUEST_METHOD

> The HTTP request method, such as "GET" or "POST". This cannot ever be an empty string, and so is always required.

SCRIPT_NAME

> The initial portion of the request URL's "path" that corresponds to the application object, so that the application knows its virtual "location". This may be an empty string, if the application corresponds to the "root" of the server.

PATH_INFO

> The remainder of the request URL's "path", designating the virtual "location" of the request's target within the application. This may be an empty string, if the request URL targets the application root and does not have a trailing slash. This value may be percent-encoded when I originating from a URL.

QUERY_STRING

> The portion of the request URL that follows the `?`, if any. May be empty, but is always required!

SERVER_NAME, *SERVER_PORT*

> When combined with *SCRIPT_NAME* and *PATH_INFO*, these variables can be used to complete the URL. Note, however, that HTTP_HOST, if present, should be used in preference to *SERVER_NAME* for reconstructing the request URL. *SERVER_NAME* and *SERVER_PORT* can never be empty strings, and so are always required.

*HTTP_* Variables

> Variables corresponding to the client-supplied HTTP request headers (i.e., variables whose names begin with *HTTP_*). The presence or absence of these variables should correspond with the presence or absence of the appropriate HTTP header in the request.

In addition to this, the Rack environment must include these Rack-specific variables:

rack.version

> The Array `[1,0]`, representing this version of Rack.

rack.url_scheme

> *http* or *https*, depending on the request URL.

rack.input

> See below, the input stream.

rack.errors

> See below, the error stream.

rack.multithread

> `true` if the application object may be simultaneously invoked by another thread in the same process, `false` otherwise.

rack.multiprocess

> `true` if an equivalent application object may be simultaneously invoked by another process, `false` otherwise.

rack.run_once

> `true` if the server expects (but does not guarantee!) that the application will only be invoked this one time during the life of its containing process. Normally, this will only be true for a server based on CGI (or something similar).

Additional environment specifications have approved to standardized middleware APIs. None of these are required to be implemented by the server.

rack.session

> A hash like interface for storing request session data. The store must implement: `#store(key, value)` (aliased as `[]=`); `#fetch(key, default = nil)` (aliased as `[]`); `#delete(key)`; `#clear`;

The server or the application can store their own data in the environment, too. The keys must contain at least one dot, and should be prefixed uniquely. The prefix *rack.* is reserved for use with the Rack core distribution and other accepted specifications and must not be used otherwise.

The environment must not contain the keys *HTTP_CONTENT_TYPE* or *HTTP_CONTENT_LENGTH* (use the versions without *HTTP_*).

The CGI keys (named without a period) must have String values. There are the following restrictions:

- *rack.version* must be an array of Integers.
- *rack.url_scheme* must either be *http* or *https*.
- There must be a valid input stream in *rack.input*.
- There must be a valid error stream in *rack.errors*.
- The *REQUEST_METHOD* must be a valid token.
- The *SCRIPT_NAME*, if non-empty, must start with /

- The *PATH_INFO*, if non-empty, must start with /
- The *CONTENT_LENGTH*, if given, must consist of digits only.
- One of *SCRIPT_NAME* or *PATH_INFO* must be set. *PATH_INFO* should be / if *SCRIPT_NAME* is empty. *SCRIPT_NAME* never should be /, but instead be empty.

1.1.1. The Input Stream

The input stream is an IO-like object which contains the raw HTTP POST data. If it is a file then it must be opened in binary mode. The input stream must respond to `#gets`, `#each`, `#read`, and `#rewind`.

`#gets`

Must be called without arguments and return a string, or `nil` on EOF.

`#read`

Behaves like IO#read. Its signature is `#read([length, [buffer]])`. If given, `#length` must be an non-negative Integer (>= 0) or `nil`, and `#buffer` must be a String and may not be nil. If `#length` is given and not nil, then this method reads at most `#length` bytes from the input stream. If `#length` is not given or nil, then this method reads all data until EOF. When EOF is reached, this method returns nil if `#length` is given and not nil, or "" if `#length` is not given or is nil. If `#buffer` is given, then the read data will be placed into `#buffer` instead of a newly created String object.

`#each`

Must be called without arguments and only yield Strings.

`#rewind`

Must be called without arguments. It rewinds the input stream back to the beginning. It must not raise `Errno::ESPIPE`: that is, it may not be a pipe or a socket. Therefore, handler developers must buffer the input data into some rewindable object if the underlying input stream is not rewindable.

`#close`

Must never be called on the input stream.

1.1.2. The Error Stream

The error stream must respond to `#puts`, `#write`, and `#flush`.

`#puts`

Must be called with a single argument that responds to `#to_s`.

`#write`

Must be called with a single argument that is a String.

`#flush`

Must be called without arguments and must be called in order to make the error appear for sure.

`#close`

Must never be called on the error stream.

1.2. The Response

1.2.1. The Status

This is an HTTP status. When parsed as integer (`#to_i`), it must be greater than or equal to 100.

1.2.2. The Headers

The header must respond to `#each`, and yield values of key and value. The header keys must be Strings. The header must not contain a *Status* key, contain keys with *:* or newlines in their name, contain keys names that end in - or *, but only contain keys that consist of letters, digits,* or - and start with a letter. The values of the header must be Strings, consisting of lines (for multiple header values, e.g. multiple *Set-Cookie* values) seperated by *\n*. The lines must not contain characters below 037.

1.2.3. The Content-Type

There must be a *Content-Type*, except when the *Status* is 1xx, 204 or 304, in which case there must be none given.

1.2.4. The Content-Length

There must not be a *Content-Length* header when the *Status* is 1xx, 204 or 304.

1.2.5. The Body

The Body must respond to `#each` and must only yield String values. The Body itself should not be an instance of String, as this will break in Ruby 1.9. If the Body responds to `#close`, it will be called after iteration. If the Body responds to `#to_path`, it must return a String identifying the location of a file whose contents are identical to that produced by calling `#each`; this may be used by the server as an alternative, possibly more efficient way to transport the response. The Body commonly is an Array of Strings, the application instance itself, or a File-like object.

1.3. Thanks

Some parts of this specification are adopted from PEP333: Python Web Server Gateway Interface v1.0 (http://www.python.org/dev/peps/pep-0333/). I'd like to thank everyone involved in that effort.

Glossary

Innate

: Provides the core functionality for Ramaze.

Rack

: A modular Ruby webserver interface that specifies the way Ramaze communicates with web servers.

Ruby

: Interpreted object-oriented programming language, created by Yukihiro Matsumoto.

MVC

: Short for: Model View Controller. A concept made popular by Smalltalk GUIs and since adopted for many other purposes. Ramaze is a web application framework that advocates the MVC paradigm.

View

: yada ydaa

Model

: yada ydaa

Controller

: yada ydaa

provides

: Term used within Ramaze for various means of content representation of resources.