Designing by Principle

A Case Study: Rack

Rack

```
Dockerfile
Gemfile
Gemfile.lock
README.md
Rakefile
app/
bin/
config/
config.ru
db/
docker-stack.yml
docs/
extra/
lib/
log/
node_modules/
package.json
public/
spec/
surveyor.iml
tmp/
vendor/
yarn.lock
```

Rack

```
# This file is used by Rack-based servers to start the application.

require_relative 'config/environment'

run Rails.application
```

Rack...Why?

Web Servers / Application Containers

Thin Puma Mongrel
WEBrick Unicorn

Server-side Web Programming Interfaces

CGI FastCGI

SCGI

Web Frameworks

Your web application...

Rack...Why?

Rails 1.0.0...before Rack

rails/lib/fcgi_handler.rb

188 lines of code

```ruby
require 'fcgi'
require 'logger'
require 'dispatcher'
require 'rbconfig'

class RailsFCGIHandler
  SIGNALS = {
    'HUP'  => :reload,
    'TERM' => :exit_now,
    'USR1' => :exit,
    'USR2' => :restart
  }

  attr_reader :when_ready

  attr_accessor :log_file_path
  attr_accessor :gc_request_period


  # Initialize and run the FastCGI instance, passing arguments through to new.
  def self.process!(*args, &block)
    new(*args, &block).process!
  end

  # Initialize the FastCGI instance with the path to a crash log
  # detailing unhandled exceptions (default RAILS_ROOT/log/fastcgi.crash.log)
  # and the number of requests to process between garbage collection runs
  # (default nil for normal GC behavior.)  Optionally, pass a block which
  # takes this instance as an argument for further configuration.
  def initialize(log_file_path = nil, gc_request_period = nil)
    self.log_file_path = log_file_path || "#{RAILS_ROOT}/log/fastcgi.crash.log"
    self.gc_request_period = gc_request_period

    # Yield for additional configuration.
    yield self if block_given?

    # Safely install signal handlers.
    install_signal_handlers

    # Start error timestamp at 11 seconds ago.
    @last_error_on = Time.now - 11

    dispatcher_log :info, "starting"
  end

  def process!(provider = FCGI)
    # Make a note of $" so we can safely reload this instance.
    mark!

    run_gc! if gc_request_period

    provider.each_cgi do |cgi|
      process_request(cgi)

      case when_ready
        when :reload
          reload!
        when :restart
          close_connection(cgi)
          restart!
        when :exit
          close_connection(cgi)
          break
      end

      gc_countdown
    end

    GC.enable
    dispatcher_log :info, "terminated gracefully"

  rescue SystemExit => exit_error
    dispatcher_log :info, "terminated by explicit exit"

  rescue Object => fcgi_error
    # retry on errors that would otherwise have terminated the FCGI process,
    # but only if they occur more than 10 seconds apart.
    if !(SignalException === fcgi_error) && Time.now - @last_error_on > 10
      @last_error_on = Time.now
      dispatcher_error(fcgi_error, "almost killed by this error")
      retry
    else
      dispatcher_error(fcgi_error, "killed by this error")
    end
  end


  private
    def logger
      @logger ||= Logger.new(@log_file_path)
    end

    def dispatcher_log(level, msg)
      time_str = Time.now.strftime("%d/%b/%Y:%H:%M:%S")
      logger.send(level, "[#{time_str} :: #{$$}] #{msg}")
    rescue Object => log_error
      STDERR << "Couldn't write to #{@log_file_path.inspect}: #{msg}\n"
      STDERR << "  #{log_error.class}: #{log_error.message}\n"
    end

    def dispatcher_error(e, msg = "")
      error_message =
        "Dispatcher failed to catch: #{e} (#{e.class})\n" +
        "  #{e.backtrace.join("\n  ")}\n#{msg}"
      dispatcher_log(:error, error_message)
    end

    def install_signal_handlers
      SIGNALS.each do |signal, handler_name|
        install_signal_handler(signal, method("#{handler_name}_handler").to_proc)
      end
    end

    def install_signal_handler(signal, handler)
      trap(signal, handler)
    rescue ArgumentError
      dispatcher_log :warn, "Ignoring unsupported signal #{signal}."
    end

    def exit_now_handler(signal)
      dispatcher_log :info, "asked to terminate immediately"
      exit
    end

    def exit_handler(signal)
      dispatcher_log :info, "asked to terminate ASAP"
      @when_ready = :exit
    end

    def reload_handler(signal)
      dispatcher_log :info, "asked to reload ASAP"
      @when_ready = :reload
    end

    def restart_handler(signal)
      dispatcher_log :info, "asked to restart ASAP"
      @when_ready = :restart
    end

    def process_request(cgi)
      Dispatcher.dispatch(cgi)
    rescue Object => e
      raise if SignalException === e
      dispatcher_error(e)
    end

    def restart!
      config       = ::Config::CONFIG
      ruby         = File::join(config['bindir'], config['ruby_install_name']) + config['EXEEXT']
      command_line = [ruby, $0, ARGV].flatten.join(' ')

      dispatcher_log :info, "restarted"

      exec(command_line)
    end

    def reload!
      run_gc! if gc_request_period
      restore!
      @when_ready = nil
      dispatcher_log :info, "reloaded"
    end

    def mark!
      @features = $".clone
    end

    def restore!
      $".replace @features
      Dispatcher.reset_application!
      ActionController::Routing::Routes.reload
    end

    def run_gc!
      @gc_request_countdown = gc_request_period
      GC.enable; GC.start; GC.disable
    end

    def gc_countdown
      if gc_request_period
        @gc_request_countdown -= 1
        run_gc! if @gc_request_countdown <= 0
      end
    end

    def close_connection(cgi)
      cgi.instance_variable_get("@request").finish
    end
end
```

rails/lib/webrick_server.rb

170 lines of code

```ruby
class DispatchServlet < WEBrick::HTTPServlet::AbstractServlet
  REQUEST_MUTEX = Mutex.new

  # Start the WEBrick server with the given options, mounting the
  # DispatchServlet at <tt>/</tt>.
  def self.dispatch(options = {})
    Socket.do_not_reverse_lookup = true # patch for OS X

    params = { :Port        => options[:port].to_i,
               :ServerType  => options[:server_type],
               :BindAddress => options[:ip] }
    params[:MimeTypes] = options[:mime_types] if options[:mime_types]

    server = WEBrick::HTTPServer.new(params)
    server.mount('/', DispatchServlet, options)

    trap("INT") { server.shutdown }

    require File.join(@server_options[:server_root], "..", "config", "environment") unless defined?(RAILS_ROOT)
    require "dispatcher"

    server.start
  end

  def initialize(server, options) #:nodoc:
    @server_options = options
    @file_handler = WEBrick::HTTPServlet::FileHandler.new(server, options[:server_root])
    Dir.chdir(ABSOLUTE_RAILS_ROOT)
    super
  end

  def service(req, res) #:nodoc:
    begin
      unless handle_file(req, res)
        REQUEST_MUTEX.lock unless ActionController::Base.allow_concurrency
        unless handle_dispatch(req, res)
          raise WEBrick::HTTPStatus::NotFound, "`#{req.path}' not found."
        end
      end
    ensure
      unless ActionController::Base.allow_concurrency
        REQUEST_MUTEX.unlock if REQUEST_MUTEX.locked?
      end
    end
  end

  def handle_file(req, res) #:nodoc:
    begin
      req = req.dup
      path = req.path.dup

      # Add .html if the last path piece has no . in it
      path << '.html' if path != '/' && (%r{(^|/)[^./]+$} =~ path)
      path.gsub!('+', ' ') # Unescape + since FileHandler doesn't do so.

      req.instance_variable_set(:@path_info, path) # Set the modified path...

      @file_handler.send(:service, req, res)
      return true
    rescue HTTPStatus::PartialContent, HTTPStatus::NotModified => err
      res.set_error(err)
      return true
    rescue => err
      return false
    end
  end

  def handle_dispatch(req, res, origin = nil) #:nodoc:
    data = StringIO.new
    Dispatcher.dispatch(
      CGI.new("query", create_env_table(req, origin), StringIO.new(req.body || "")),
      ActionController::CgiRequest::DEFAULT_SESSION_OPTIONS,
      data
    )

    header, body = extract_header_and_body(data)

    set_charset(header)
    assign_status(res, header)
    res.cookies.concat(header.delete('set-cookie') || [])
    header.each { |key, val| res[key] = val.join(", ") }

    res.body = body
    return true
  rescue => err
    p err, err.backtrace
    return false
  end

  private
    def create_env_table(req, origin)
      env = req.meta_vars.clone
      env.delete "SCRIPT_NAME"
      env["QUERY_STRING"] = req.request_uri.query
      env["REQUEST_URI"]  = origin if origin
      return env
    end

    def extract_header_and_body(data)
      data.rewind
      data = data.read

      raw_header, body = *data.split(/^[\xd\xa]+/on, 2)
      header = WEBrick::HTTPUtils::parse_header(raw_header)

      return header, body
    end

    def set_charset(header)
      ct = header["content-type"]
      if ct.any? { |x| x =~ /^text\// } && ! ct.any? { |x| x =~ /charset=/ }
        ch = @server_options[:charset] || "UTF-8"
        ct.find { |x| x =~ /^text\// } << ("; charset=" + ch)
      end
    end

    def assign_status(res, header)
      if /^(\d+)/ =~ header['status'][0]
        res.status = $1.to_i
        header.delete('status')
      end
    end
end
```

Almost 400 lines of code just to use 2 different handlers!

...after Rack

```ruby
# This file is used by Rack-based servers to start the application.

require_relative 'config/environment'

run Rails.application
```

Rack…Why?

Principles

- Well-defined interfaces
  (good fences make good neighbors)
- Extensibility *(don’t try to predict the future)*
- Composition *(we’re better together)*
- Immutability *(respecting boundaries, keeping promises)*

Well-Defined Interface

(good fences make good neighbors)

```
# rack/handler/tomcat.rb

class Rack::Handler::Tomcat
  def self.run(app, options = {})
    # talk to Tomcat
  end
end
```

```
# rack/handler/apache.rb

class Rack::Handler::Apache
  def self.run(app, options = {})
    # talk to Apache
  end
end
```

```
# rack/handler/nginix.rb

class Rack::Handler::Nginix
  def self.run(app, options = {})
    # talk to NGINIX
  end
end
```

```
# config.ru

run Proc.new { |env| ['200', {'Content-Type' => 'text/html'}, ['Hello neighbor!']] }
```

Extensibility

(don’t try to predict the future)

```
# config.ru

use Rack::CommonLogger
use Rack::Session::Cookie
run App
```

Extensibility

(don't try to predict the future)

```
# config.ru

use Rack::CommonLogger
use Rack::Session::Cookie
run App
```

We want to make systems that can easily be extended without modifying it source code.

- **Web API**
- **Service Objects**
- **Adapter Pattern**
- **Blocks / Procs / Closures**

Composition

(we’re better together)

```
# config.ru

use Rack::CommonLogger
use Rack::Session::Cookie
run App
```

```
# config.ru

App = Rack::CommonLogger.new(
        Rack::Session::Cookie.new(MyApp.new))

run App
```

Composition

(we're better together)

```
# config.ru

use Rack::CommonLogger
use Rack::Session::Cookie
run App
```

```
# config.ru

App = Rack::CommonLogger.new(
        Rack::Session::Cookie.new(MyApp.new))

run App
```

Composable systems are extensible systems

Immutability

(respecting boundaries, keeping promises)

```
class MyApp
  def call(env)
    [‘200’, {‘Content-type’ ⇒ ‘text/html’}, [“This is true”]]
  end
end
```

What’s missing?

Immutability

(respecting boundaries, keeping promises)

```
class MyApp
  def call(env)
    ['200', {'Content-type' => 'text/html'}, ["This is true"]]
  end
end

class Logger
  def initialize(app)
    @app = app
  end

  def call(env)
    log(env)
    app.call(env)
  end
end
```

Why does this work?

Tradeoffs

(everything has a cost)

References

- Inventing on Principle - Bret Victor
  https://vimeo.com/36579366
- Simplicity Matters - Rich Hickey
  https://www.youtube.com/watch?v=rI8tNMsozo0
- The Mess We're In - Joe Armstrong
  https://www.youtube.com/watch?v=lKXe3HUG2l4