

Designing by Principle

A Case Study: Rack

Principles

Everything should be made as simple as possible, but not simpler.

Principles

Everything should be made as simple as possible, but not simpler.

Definition

“a basic truth, law, or assumption”

Principles

Everything should be made as simple as possible, but not simpler.

Definition

“a basic truth, law, or assumption”

Rule

“don’t use Comic Sans”

Principles

Everything should be made as simple as possible, but not simpler.

Definition

“a basic truth, law, or assumption”

Rule

“don’t use Comic Sans”

Principle

“use a font that works well for your design”

Principles

Everything should be made as simple as possible, but not simpler.

Definition

“a basic truth, law, or assumption”

Rule

“don’t use Comic Sans”

Principle

“use a font that works well for your design”

Where rules are specific principles are general, and therefore have tremendous utility.

Learning to apply them is hard.

Principles

Everything should be made as simple as possible, but not simpler.

Definition

“a basic truth, law, or assumption”

Scientific Method

Keeping clear in mind the outcome we’re trying to achieve, and then considering what principles will help us get there.

Tradeoffs

(everything has a cost)

System design is about balancing *simplicity* with
expressive power



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

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

  # Start the WEBrick server with the given options, mounting the
  # DispatchServlet at /dispatch/v1/tbs
  def self.dispatch(options = {})
    Socket.do_not_reverse_lookup = true # patch for OS X

    params = {
      :port => options[:port].to_i,
      :server_type => options[:server_type],
      :bind_address => options[:ip]
    }
    params[:mime_types] = options[:mime_types] if options[:mime_types]

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

    trap("INT") { server.shutdown }

    require File.join(Rails.root, "...", "config", "environment") unless defined?(RAILS_ROOT)
    require "dispatcher"
  end

  def initialize(server, options) #nodoc:
    super
  end
end
```

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

...after Rack

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                                # rack/handler/apache.rb
class Rack::Handler::Tomcat                         class Rack::Handler::Apache
  def self.run(app, options = {})                   def self.run(app, options = {})
    # talk to Tomcat                                # talk to Apache
  end                                              end
end                                            end

# rack/handler/nginx.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 its source code.

- **Modularity (via functions, or objects)**
 - **Function Composition**
 - **Higer-order functions**
 - **Adapter Pattern**
 - **Service Objects**
- **Web API**

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
```

Why does this work?

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

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=IKXe3HUG2I4>