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*

Simplicity	Expressive Power
Reliability, maintainability	Satisfying requirements, flexibility

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

rails/lib/webrick_server.rb
170 lines of code

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
```

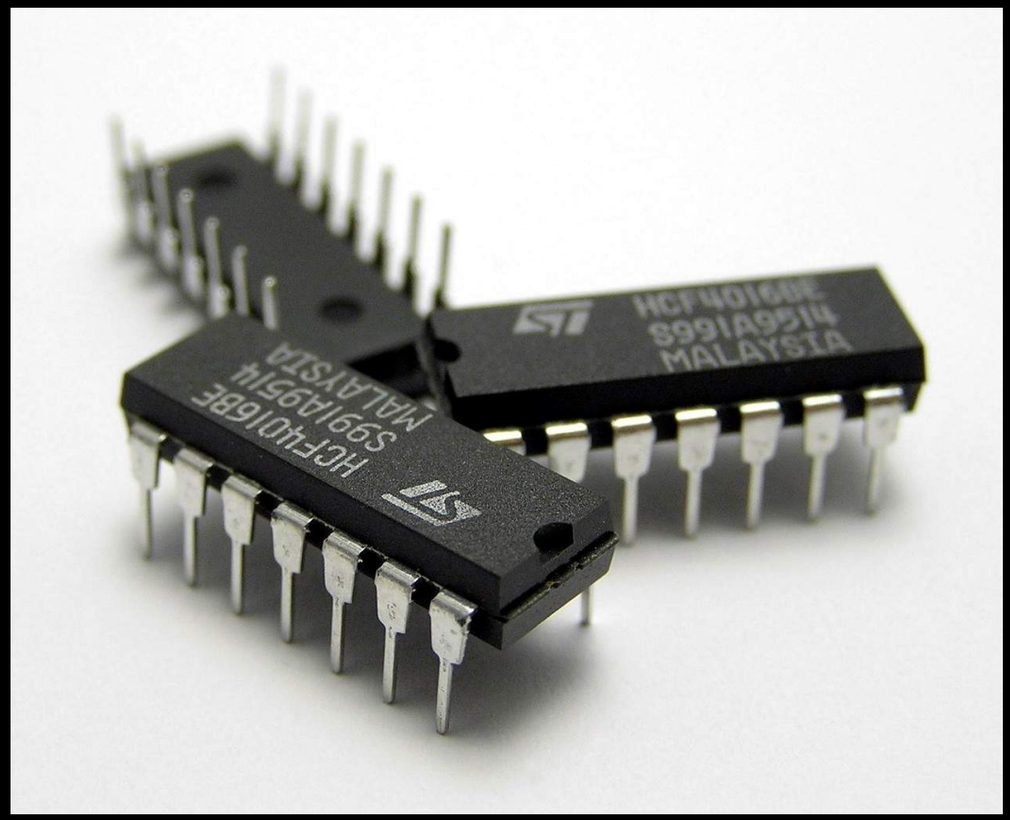
```
# rack/handler/apache.rb

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!’]] }
```

Modularity

(good fences make good neighbors)



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
```

Generally, we want to make systems that can easily be extended without modifying it source code.

- **Function Composition**
- **Higer-order functions**
- **Adapter Pattern**
- **Service Objects**
- **Lookup Tables**
- **Web API**

Extensibility

(don’t try to predict the future)

```
# config.ru

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

…but, what’s the cost?

Composition

(we're better together)

```
# config.ru

use Rack::CommonLogger
use Rack::Session::Cookie
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(App.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(App.new))
run App
```

```
# config.ru

app1 = App.new
app2 = Rack::Session::Cookie.new(app1)
app3 = Rack::CommonLogger.new(app2)
run app3
```

Composition

(we're better together)

```
# config.ru

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

Composable systems are extensible systems

Composition

(we're better together)

```
def square(x)
  x * x
end
```

```
square(4) # ⇒ 16
square(64) # ⇒ 4096
```

Composition

(we're better together)

```
def square(x)
  x * x
end
```

```
def sum_of_squares(x, y)
  square(x) + square(y)
end
```

Composition

(we're better together)

```
def square(x)
  x * x
end
```

```
def sum_of_squares(x, y)
  square(x) + square(y)
end


sum_of_squares(4, 64) # ⇒ 4112
```

Composition

(we're better together)

```
def square(x)
  x * x
end
```

```
def distance(x1, x2, y1, y2)
  Math.sqrt(square(x2 - x1) + square(y2 - y1))
end
```

Composition

(we're better together)

Composable systems…

- Are extensible
- Are very often easy to reason about
- Have a linguistic or lyrical quality (algebraic)

Composition

(we're better together)

- Can cost performance-wise (memory)

  (but most languages make it pretty cheap)

Composition

(we're better together)

- SQL and the relational model
- Object Systems (i.e. Ruby, JavaScript)
- Functions and Procedures
- Data Formats (i.e. JSON, HTML)

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?

Immutability

(respecting boundaries, keeping promises)

- Allows you to make assumptions
- Maintains the integrity of abstractions
- Simplicity
- Performance optimization & concurrency are simpler

Immutability

(respecting boundaries, keeping promises)

- At times can be difficult to pull off
- Can cost performance-wise

Immutability

(respecting boundaries, keeping promises)

- Most up-in-coming languages (i.e. Haskell, Clojure, Rust, Go, Swift, Scala, Kotlin…)

- Older languages adding (i.e. Java, C#, JavaScript)

- Frameworks (React.js, Vue.js)

- Ruby bang idiom (i.e. “!” at the end of method calls) (e.g. Hash#merge and Hash#merge!)

- Concurrent-ruby

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