Design Principles behind

I am José Valim

@josevalim

I work at < plataforma />

tecnologia e engenharia de software

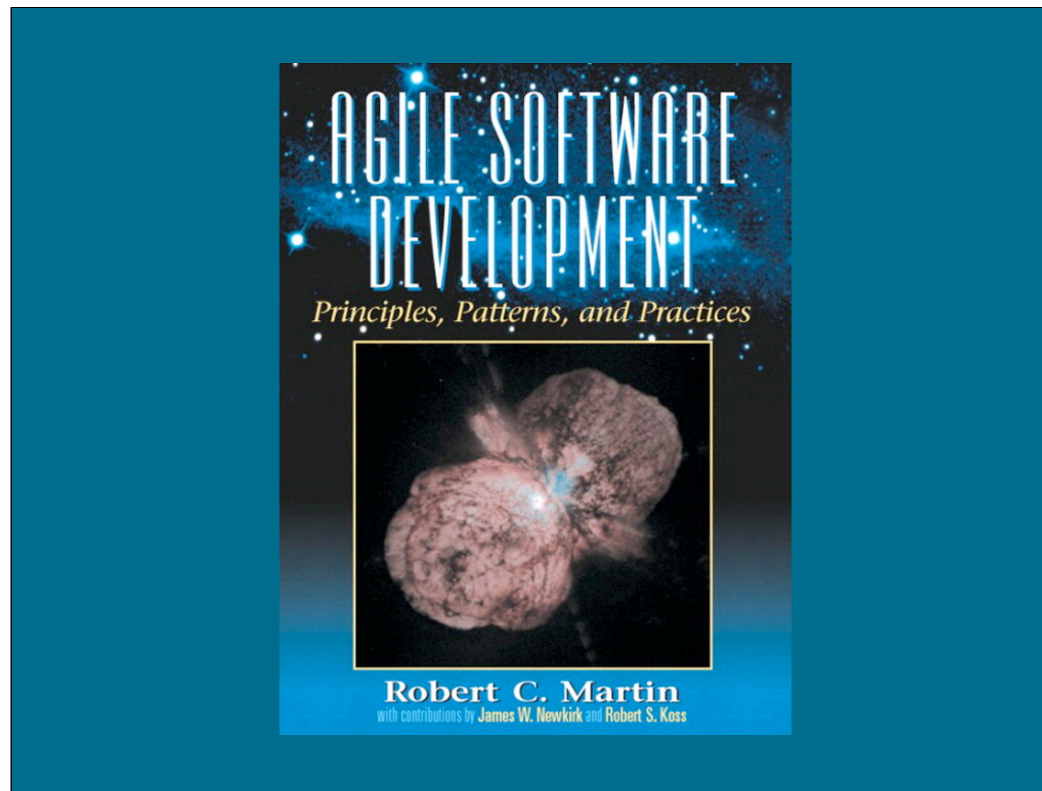
blog.plataformatec.co

Core Team Member

Rails 3

Give the context about this talk. Mention the first time we heard about SOLID Design Principles.

SOLID Design



1) Static -> Dynamic. Simpler. Easier to get it right.
2) You don’t need to know these principles. Mention Yehuda.
3) Explicitly show the benefits and what this allow us to do.

- **S**ingle Responsibility Principle
- **O**pen/closed Principle
- **L**iskov Substitution

Single Responsibility Principle

“A class should have one, and only one, reason to change”

- Uncle Bob

Example:

Views Refactoring

0.x

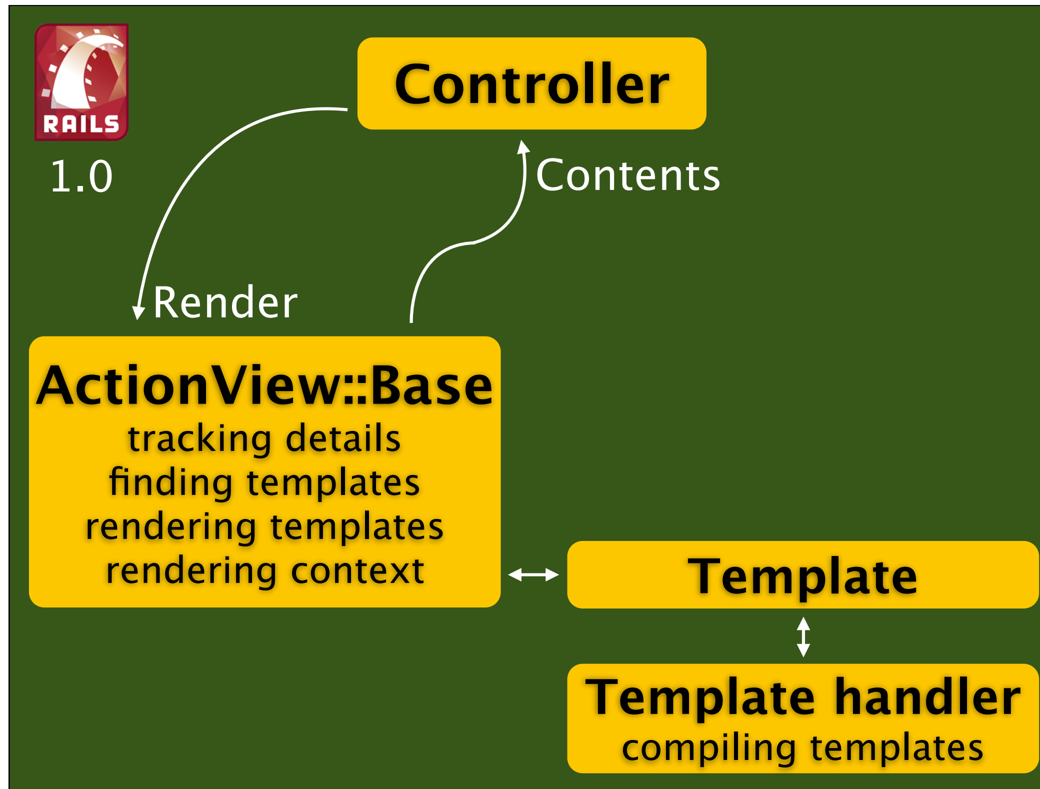
Controller

Render

Contents

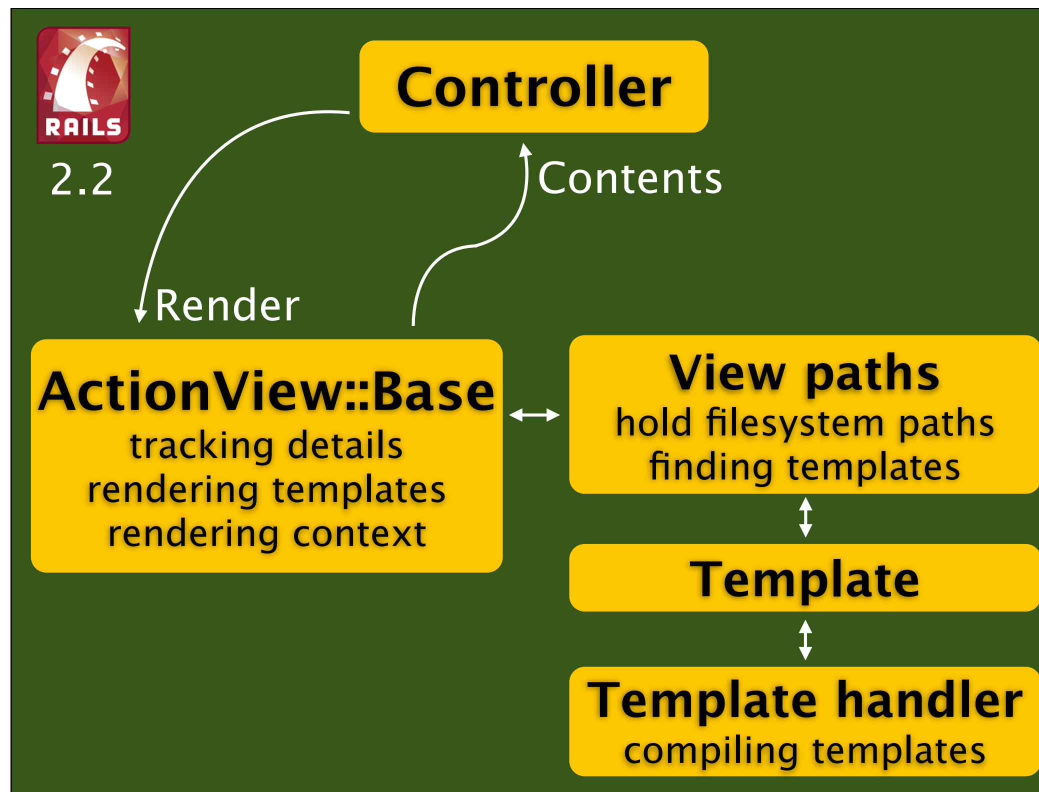
ActionView::Base

tracking details
finding templates
compiling templates
rendering templates
rendering context


1.0
Controller
Contents
Render
ActionView::Base
tracking details
finding templates
rendering templates
rendering context
Template
Template handler
compiling templates

RAILS

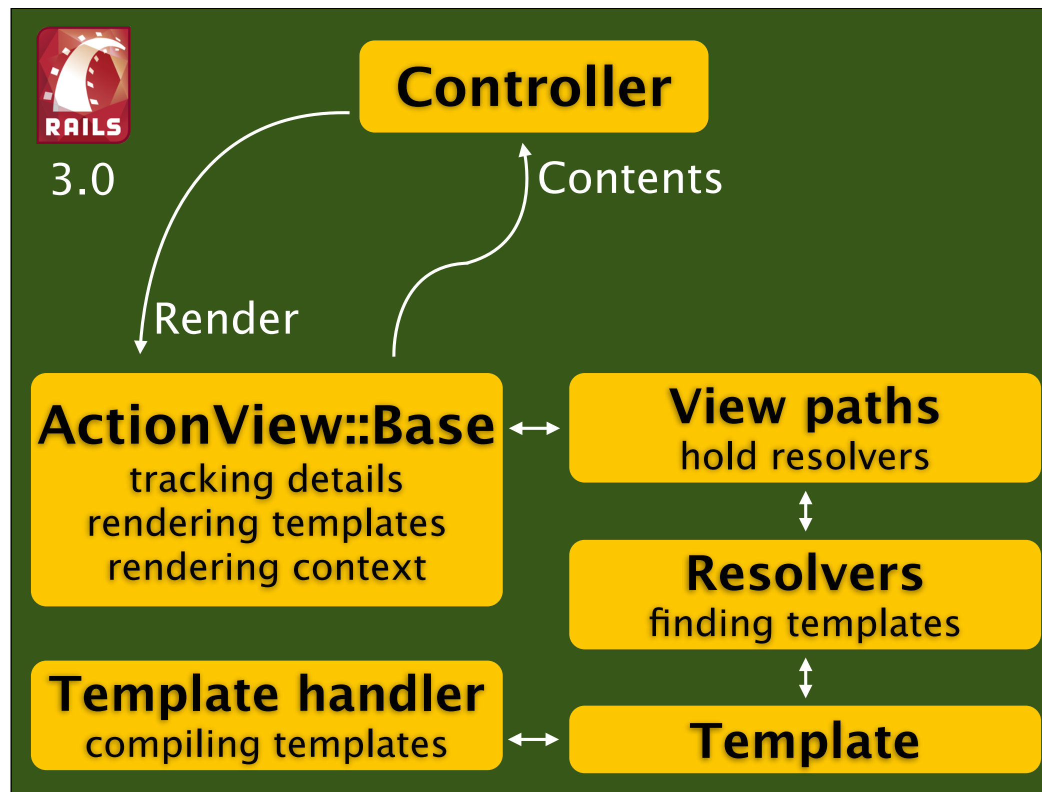
builder, rjs, haml...



Explain the workflow. Two responsibilities on view paths.

Rails Engines!

Explain what engines are.

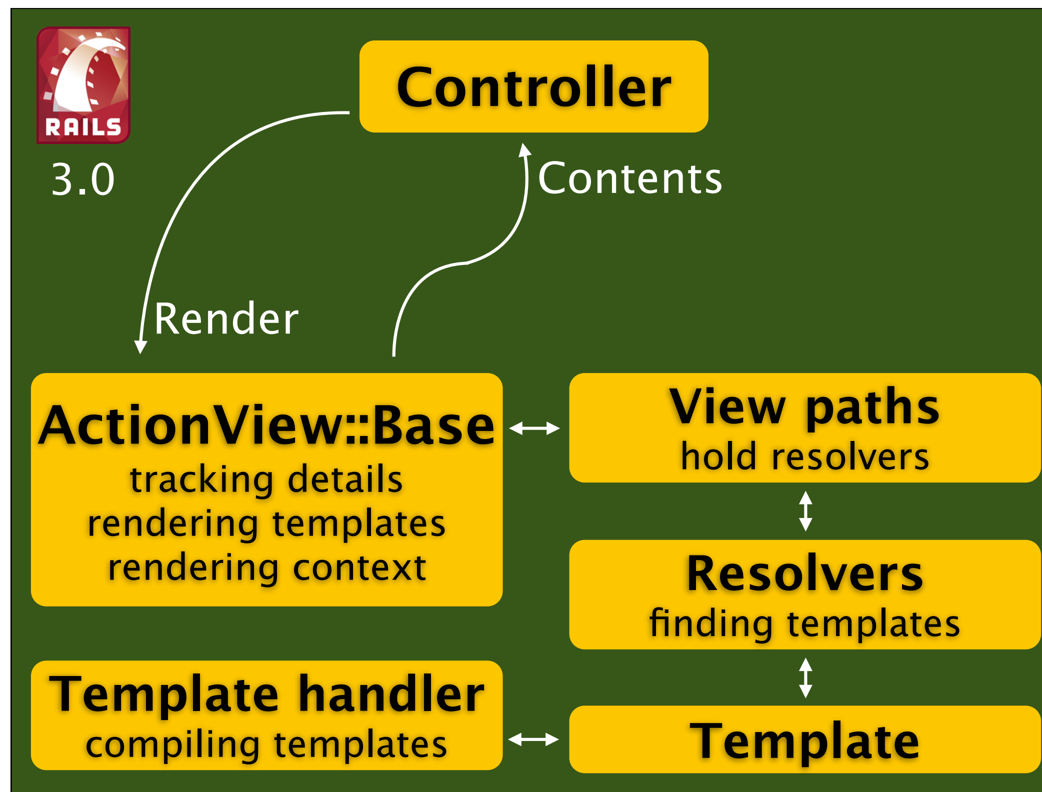


Explain the workflow.

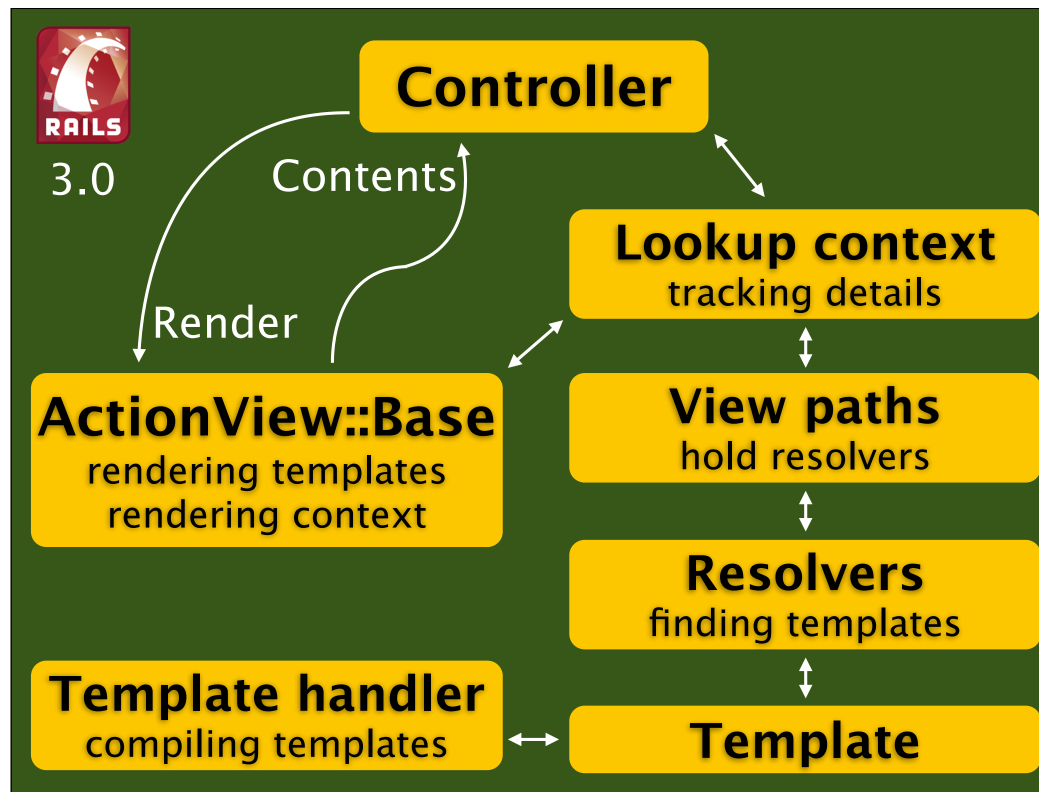
The resolver object no longer restricts templates to the filesystem

```ruby
class BasicController < ActionController::Base
  self.view_paths = [
    ActionView::FixtureResolver.new(
      "basic/hello_world.html.erb" => "Hello world!"
    )
  ]

  def hello_world
    render :action => "hello_world"
  end
end
```

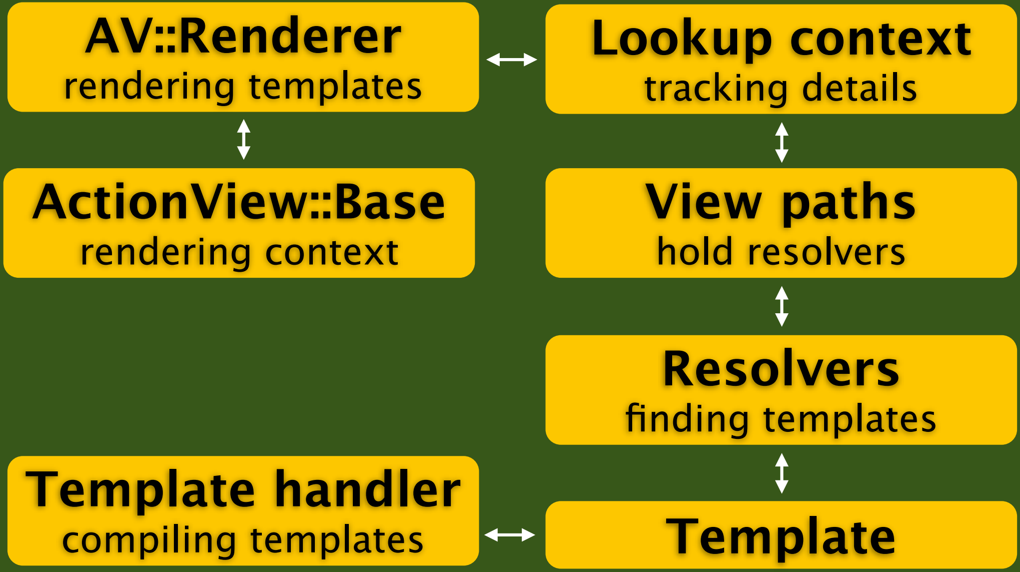
You can create your own abstractions, allowing you to read templates from the database!



Give an introduction on tracking details.



Interface segregation, just mention it. This is Rails 3.0, but we can see AV::Base has two responsibilities. Can we break these responsibilities apart? Which benefits would it bring?



3.1
Controller
AV::Renderer
rendering templates
Lookup context
tracking details
ActionView::Base
rendering context
View paths
hold resolvers
Resolvers
finding templates
Template handler
compiling templates
Template

```ruby
def lookup_context
  ActionView::LookupContext.new(self.class._view_paths)
end

def view_renderer
  ActionView::Renderer.new(lookup_context)
end

def _render_template(options)
  view_renderer.render(view_context, options)
end
```

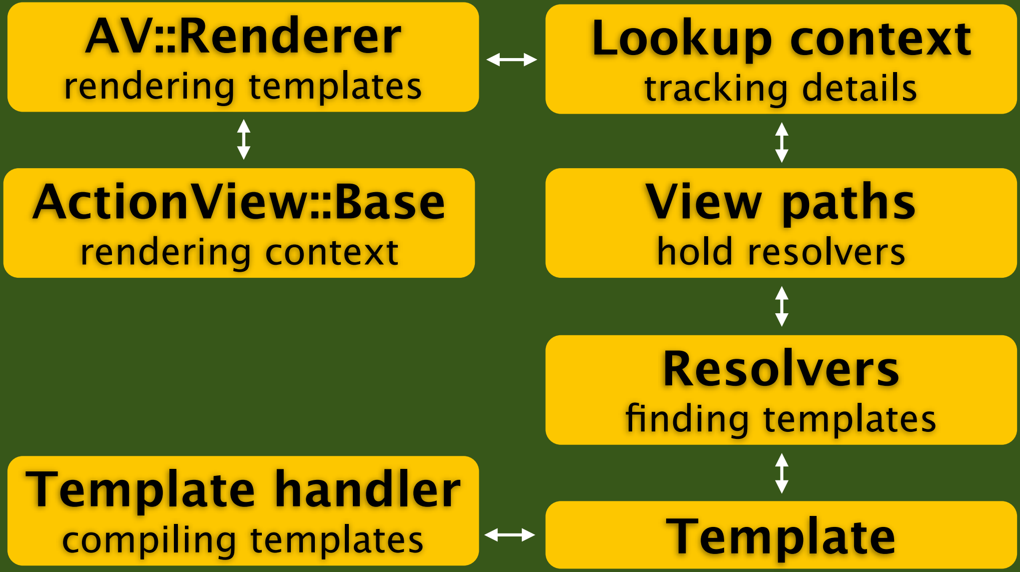
```
def view_context
  ActionView::Base.new(view_renderer, view_assigns, self)
end
```

But the whole deal about this is to make objects very simple and small so we can easily replace them. So how easily can we replace view_context?

Maybe templates could be rendered in the controller context?

```ruby
class BasicController < ActionController::Base
  include ActionView::Context     # STEP 1
  before_filter :_prepare_context # STEP 2

  def hello_world
    @value = "Hello"
  end

  protected

  def view_context # STEP 3
    self
  end

  def __controller_method__
    "controller context!"
  end
end
```

```
# app/views/basic/hello_world.html.erb
<%= @value %> from <%= __controller_method__ %>

# would return...
Hello from controller context!
```


3.1
Controller
AV::Renderer
rendering templates
Lookup context
tracking details
ActionView::Base
rendering context
View paths
hold resolvers
Resolvers
finding templates
Template handler
compiling templates
Template

RAILS

Top-down comprehension	Worse
Maintainability	Better
Focused comprehension	Much Better
Extensibility	POSSIBLE!

As Yehuda once put very well...

Open/closed Principle

“You should be able to extend a class behavior without modifying it”

- Uncle Bob

This is the open–closed principle, which in my opinion is one of the central innovations of object technology: the ability to use a software component as it is, while retaining the possibility of adding to it later through inheritance. Unlike the records or structures of other approaches, a class of object technology is both closed and open: closed because we can start using it for other components (its clients); open because we can at any time add new properties without invalidating its existing clients.

Open for extension,
closed for modification

```
class ApplicationController <
  ActionController::Base
end
```

Easy to follow because Ruby classes are all open for extensions...

... but easy to violate because they are not closed for modification.

```
# In your initializer ...
config.active_record.table_name_prefix = “foo”

# Then ...
ActiveRecord::Base.table_name_prefix #=> “foo”
```

```
class ApplicationModel <
  ActiveRecord::Base
end
```

```ruby
class MyApp::ApplicationModel <
  ActiveRecord::Base
end
```

Dependency Inversion

“Depend on abstractions, not on concretions”

- Uncle Bob

Duck typing. It doesn’t really matter the object as long as he implements method X.

def initialize(app:

def initialize(app)

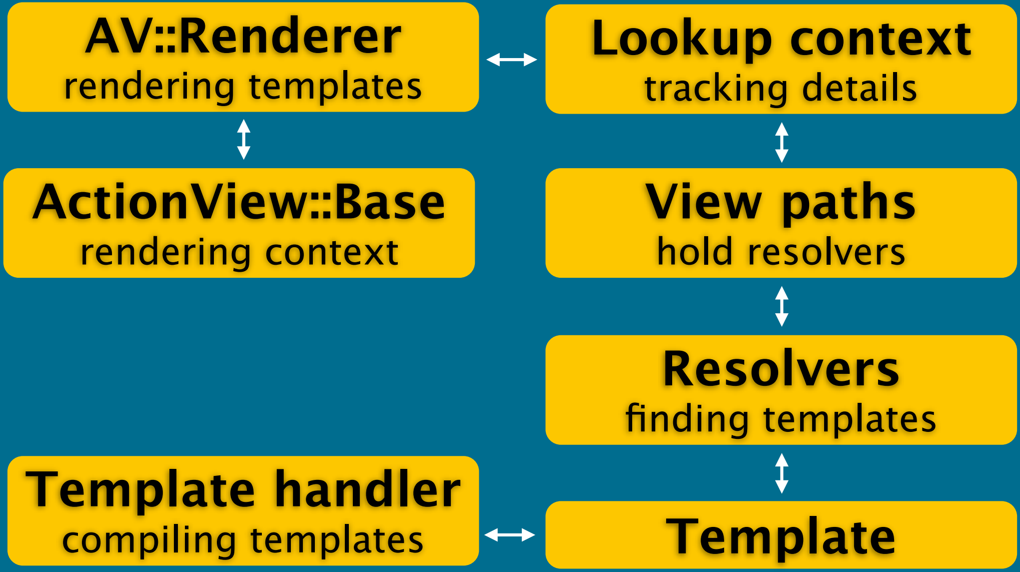
@rack_app.call(e

1) Define

match “/foo”, to:

match “/foo”, to:

match “/foo”,
  to:


Controller
3.1
AV::Renderer
rendering templates
Lookup context
tracking details
ActionView::Base
rendering context
View paths
hold resolvers
Resolvers
finding templates
Template handler
compiling templates
Template

RAILS

```
@resolver.find_all(
  name, prefix, partial, details,
key, locals
```

```
@haml_handler.call(templ
```

You don’t need to inherit from anything to be a template handler.

2) Remove hardcoded dependencies

```ruby
class PostsController < ApplicationController
  use ActiveRecord::IdentityMap::Middleware, :only => :index

  # ...
end

# And this builds the middleware stack ...
PostsController.action(:index)
```

```ruby
def self.action(name)
  middleware_stack.build(name.to_s) do |env|
    new.dispatch(name, ActionDispatch::Request.new(env))
  end
end
```

```
def self.action(name, klass = ActionDispatch::Request)
  middleware_stack.build(name.to_s) do |env|
    new.dispatch(name, klass.new(env))
  end
end
```

3) Define hooks

Not always is possible to have a protocol. For example, customize something in the rails initialization stack or the request stack.

1) Load console on sandbox
rails console --sandbox

2) Internally...
require “rails/console/sandbox”

```ruby
ActiveRecord::Base.connection.increment_open_transactions
ActiveRecord::Base.connection.begin_db_transaction

at_exit do
  ActiveRecord::Base.connection.rollback_db_transaction
  ActiveRecord::Base.connection.decrement_open_transactions
end
```

Instead provide a hook...

```ruby
class ActiveRecord::Railtie < Rails::Railtie
  console do |sandbox|
    if sandbox
      require "active_record/railties/console_sandbox"
    end
    ActiveRecord::Base.logger = Logger.new(STDERR)
  end
end
```

Liskov Substitution Principle

“Derived classes must be substitutable for their base classes”

- Uncle Bob

"Derived classes must be **substitutable** for their base classes"

- Uncle Bob

@rack_app.call(e

- Receives a hash
- Returns an array with status code, headers and an object that responds

Define substitutable but also don't violate the principle

Datamapper
x
Active Record

No defined protocol in Rails 2.3.

Active Model

User.model_name
user.persisted?
user.valid?
user.errors
user.to_key
user.to_param

It makes substitution possible.

How do we ensure substitutability?

ActiveModel::Lint::Tests

```ruby
class LintTest < ActiveSupport::TestCase
  include ActiveModel::Lint::Tests

  def setup
    @model = SomeDatamapperModel.new
  end
end
```

Interface Segregation Principle

“Make fine grained interfaces that are client specific”

- Uncle Bob

“Clients should depend on as narrow protocol as possible”

- Jim Weirich

Active Model

User.model_name
user.persisted?
user.valid?
user.errors
user.to_key
user.to_param

Define only what you need and nothing more.

How do we ensure
a narrow protocol?

```
class MyPost < ActiveRecord::Base
end

my_post = MyPost.new
assert_equal "my_posts/1", url_for(my_post)
```

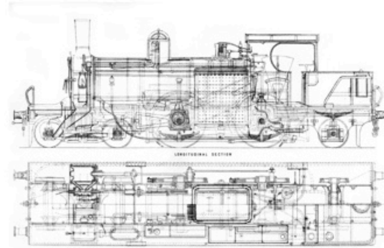
```
my_post = Mock.new
my_post.stubs(:model_name).returns("MyPost")
my_post.stubs(:to_param).returns(2)
assert_equal "my_posts/1", url_for(my_post)
```

Code to well-defined and narrow protocols

The Pragmatic Programmers

Crafting Rails Applications

Expert Practices for Everyday Rails Development



José Valim

edited by Brian P. Hogan

Questions?