

# A Guide to Hacking Rails Internals

by  
Pradeep Elankumaran

Intridea

# Our Products



# Good Code

Solves real-world problems

Efficient

Elegant

Clear

Non-repetitive

# Why hack the framework?

- Rails doesn't help you all the time.
- Convention over configuration is a template, not a way of life.
- It's fun (except when it's not).
- Craftsmen need to know their tools.

# If you understand Ruby, Rails Internals are easy to learn.

Don't be afraid to get your hands dirty.  
It's **totally** worth it.

# Metaprogramming

(thank you Matz)

# Your Metaprogramming Toolkit

## alias\_method\_chain

```
alias :save :save_without_photo  
alias :save_with_photo :save
```



```
alias_method_chain :save, :photo
```

```
#=> save_with_photo(*args) → you need to define this  
#=> save_without_photo(*args) → the old save method
```

# Your Metaprogramming Toolkit

## Accessors

`cattr_accessor`

```
class Post < ActiveRecord::Base
  cattr_accessor :class_level_method
  @@class_level_method = :hello
end
```

```
Post.class_level_method #=> :hello
```

# Your Metaprogramming Toolkit

## Accessors

`mattr_accessor`

```
module ActiveSupport::Dependencies
  mattr_accessor :loaded_plugins
  self.loaded_plugins = [:exception_logger]
end
```

```
ActiveSupport::Dependencies.loaded_plugins #=> [:exception_logger]
```

# Your Metaprogramming Toolkit

## Hook Methods (from Ruby)

```
module HasCollection
  module InstanceMethods
    def self.included(base)
      #do stuff
    end
  end
end

module ClassMethods
  def self.extended(base)
    #do stuff
  end
end
end
```

# Your Metaprogramming Toolkit

## Hook Methods (from Ruby)

```
class Post < ActiveRecord::Base
  def method_added(name)
    #do stuff
  end

  def singleton_method_added(name)
    #do stuff
  end
end
```

# Your Metaprogramming Toolkit

## Hook Methods (from Ruby)

```
module HasAvatar
  def method_missing(name)
    #do stuff
  end

  def const_missing(name)
    #do stuff
  end
end
```

# Your Metaprogramming Toolkit

## Other Useful Methods

```
returning(Post.new) do |post|  
  post.type = :blog_post  
end
```

```
#=> Post{type:blog_post, new_record:true}
```

```
with_options :only => [:new, :create] {|opt_parser|  
  opt_parser.before_filter :check_user_validity  
  opt_parser.around_filter :set_timezone  
}
```

# The Rails Class Loader

(know it, love it and then despise it deep down inside)

# Rails Class Loading

All class loading is done through the ActiveSupport::Dependencies module.

It's a dense piece of code.

# Rails Class Loading

*Dependencies.load\_paths*

This array contains a list of absolute paths that Rails looks for .rb files in

# Rails Class Loading

*Dependencies.load\_once\_paths*

The files in this array are only loaded once whenever they are first encountered, then never reloaded until the next stack startup.

Example: *RAILS\_ROOT/lib*

# Rails Class Loading

*Dependencies.autoloaded\_constants*

This array contains a list of constants found (as Strings) that are reloaded on every request in development mode.

# Rails Class Loading

Uses the **Object.const\_missing()** hook to load classes

**GOOD** - Allows Rails to always have the newest version of any class in development mode.

**BAD** - Rails doesn't load classes unless a reference to a class shows up while loading some other class. This means bad code doesn't crash until runtime every once in a while.

**BAD** - Rails uses the Inflector module to grab filenames from the missing constants. This causes lots of issues when files are not in the place that Rails expects them to be.

**UGLY** - Exceptions are suppressed! Most class loader issues are extremely difficult to track down.

# A Brief Tour

```
class User < ActiveRecord::Base
  has_many :posts
  has_many :comments
end
```

Dependencies.const\_missing(:Post)

resolve **Post** to the file  
'RAILS\_ROOT/app/models/post.rb'

Dependencies.require\_or\_load("RAILS\_ROOT/  
app/models/post.rb")

loads post.rb using load(), then  
returns to the next line of the  
User class

# It's Rough. So WHY?

- Because it's worth it.
- Reusable plugins that are slices of an app.
  - Engines, Desert
  - SocialSpring's custom classloader (tiny\_apps)
- A cleaner folder structure.
- Learning the class loader is an investment that pays off handsomely, since you will effectively have to learn a lot of Rails to make sense of it.
- Makes Rails less of a black box.

**Most Internals hacking  
involves hooking in the right  
code at the right places.**

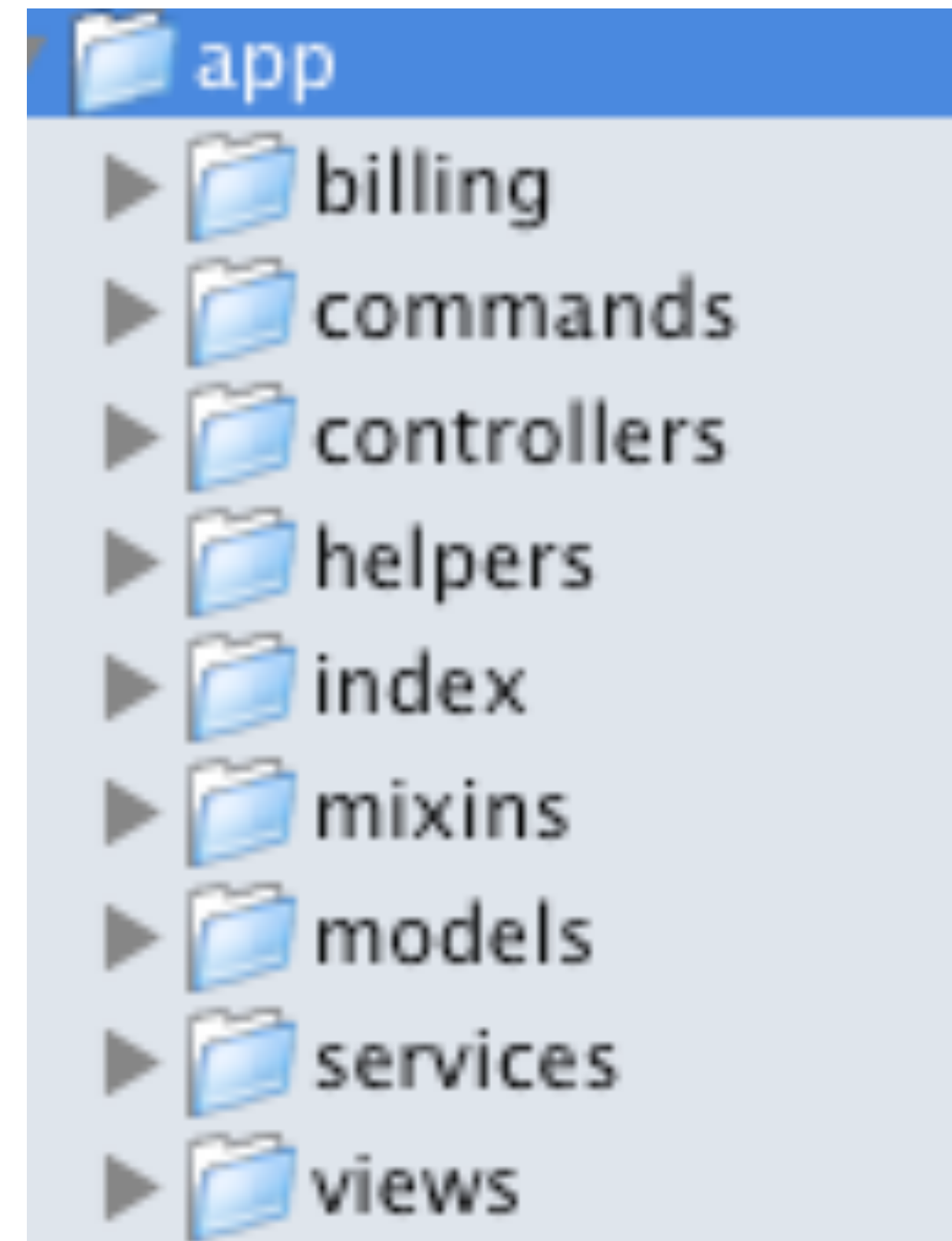
Keeping this in mind...

# Structure

(or How To OWN Your Codebase)

# Your Codebase Can Look Like This

In `environment.rb`,  
`config.load_paths += %W({RAILS_ROOT}/app/billing)`



Let's do a tiny exercise.

# The Requirements

- We want our ActiveRecord models to all have an `be_ruthless()` method.
- This method will somehow make our AR models **RUTHLESS!**
- This feature is app-specific, so no point in making it into a plugin.

# First, the Mixin

```
module BeRuthless
  attr_accessor :ruthless_classes
  self.ruthless_classes = []

  def self.included(base)
    base.send(:extend, ClassMethods)
  end

  module ClassMethods
    def be_ruthless(*args)
      # do stuff here
      include SingletonMethods
      include InstanceMethods
    end

    def is_ruthless?
      false
    end
  end

  module SingletonMethods
    def is_ruthless?
      true
    end
  end
end
```

```
module InstanceMethods
  def self.included(base)
    base.class_eval {
      attr_accessor :ruthlessness
      alias_method_chain :save,
                        :ruthless_pic
    }
    @ruthlessness = 10
  end

  def evil?
    @ruthlessness > 200
  end

  def save_with_ruthless_pic(pic, *args)
    self.ruthless_pic = pic
    save_without_ruthless_pic(*args)
  end
end

ActiveRecord::Base.send(:include, BeRuthless)
```

## Let's put `be_ruthless.rb` in `app/mixins`

In `environment.rb`,  
`config.load_paths += %W(#{RAILS_ROOT}/app/mixins)`

```
class User < ActiveRecord::Base
  be_ruthless
end
```

```
class UsersController < ApplicationController
  def index
    @user = User.find(:first)
    render :text => "hello"
  end
end
```

and then we restart our server, and hit /users

# Uh OH!

## NameError in UsersController#index

```
undefined local variable or method `be_ruthless' for User(id: integer, created_at: datetime, updated_at: datetime):Class
```

RAILS\_ROOT: /Users/pradeep/intridea/test\_app

[Application Trace](#) | [Framework Trace](#) | [Full Trace](#)

```
vendor/rails/activerecord/lib/active_record/base.rb:1699:in `method_missing'  
app/models/user.rb:2  
app/controllers/users_controller.rb:3:in `index'
```

## Request

### Parameters:

None

[Show session dump](#)

## Response

### Headers:

```
{"Content-Type"=>nil,  
 "cookie"=>[],  
 "Cache-Control"=>"no-cache"}
```

# The Solution

Hook into the plugin loader, and load the files in the mixins/ folder **before** any of the other files are loaded.

# The Rails Plugin Loader

- can be hacked easily :)
- loads each plugin in the order specified in `environment.rb`
- loads up each plugin's `init.rb` file right after it finds it

```
def load_plugins
  plugins.each do |plugin|
    plugin.load(initializer)
    register_plugin_as_loaded(plugin)
  end
  ensure_all_registered_plugins_are_loaded!
end
```

# The `mixin_loader` plugin

has just an `init.rb`

```
mixins_path = File.join(RAILS_ROOT, "app", "mixins")
Dependencies.load_paths << mixins_path
Dependencies.load_once_paths << mixins_path
FileList[File.join(mixins_path, "*")].each do |file|
  filename = file.gsub(/^#{mixins_path}\//, "")
  Dependencies.require_or_load(file)
end
```

**That should take care of it.**

# A Quick Sidenote

Here's how you load up the Rails stack from an external script

```
require "/path-to-rails/config/boot"
```

```
require "#{RAILS_ROOT}/config/environment"
```

After this, you will be able to access in your script everything you can in your Rails app.

# Some Tips

- `*args` is your best friend whenever you're doing something involving existing methods.
- While working with the Dependencies module, trace puts statements help a lot
- The majority of the class loader issues are due to classes being loaded in the wrong order.
- If you hit a wall, take a break and come back to it.

# Wrapping Up

- There is **ALWAYS** a way to do things the way you want using Rails.
- Being a Rails God requires you to be a Ruby God first.
- Up your game by learning your tools inside and out.

# Questions?