



# RUBY ON RAILS

“Web development that doesn't hurts”

# PARTE II: ROR

- Contenido
  - Introducción al framework
  - Componentes clave
  - Testing
  - JavaScript // CoffeeScript
  - LiveController
  - RailsAvanzado
  - APIs y Servicios

# INTRODUCCIÓN A RAILS

- Creado por David Hanson (HDD)
- 2004 se publica la primera beta
- La empresa de HDD, 37Signals, es la encargada de gestionar el core
- Sale a partir del proyecto Basecamp de 37S
- La combinación con Ruby hace un framework muy potente y flexible. Reduce el time 2 market.

# MITOS SOBRE RAILS

- Rails es un framework demasiado nuevo
- Rails es difícil de desplegar en sistemas
- Rails no tiene comunidad
- No escala
- No es multi-thread (comparémoslo con el siguiente...)
- No es concurrente



# USAN RAILS

**EA**

**MicroSites**

**YellowPages**

**Twitter**

algo le queda

**NewYork  
Times**

**Groupon**

**GitHub**

**Ask.fm**

**Basecamp**

**Hulu**

**ThemeForrest**

**SlideShare**

**Scribid**

**VK**

**Shopify**

**change.org**

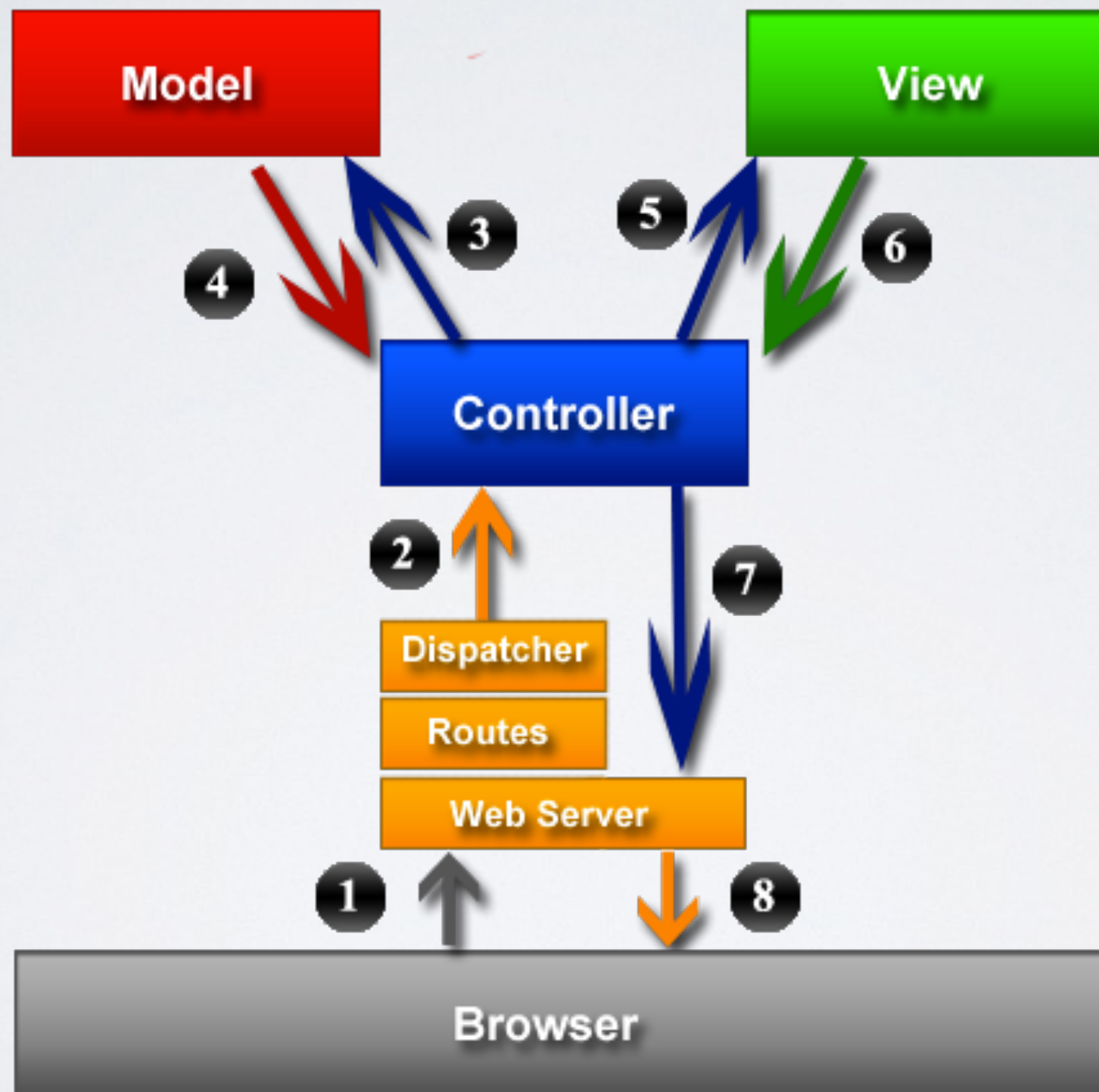
# CARACTERÍSTICAS

- Arquitectura MCV
- ORM (ActiveRecord)
- Convención sobre configuración
- Principios DRY
- Embedded Ruby para las vistas (\*.erb)
- jQuery como framework de JavaScript por defecto

# CARACTERÍSTICAS

- Testing completo incluido
- Conexiones permanentes con LiveController
- Gestión de la caché básica o avanzada (Russian doll caching)
- Turbolinks
- Sistema de comandos muy completo

# MVC





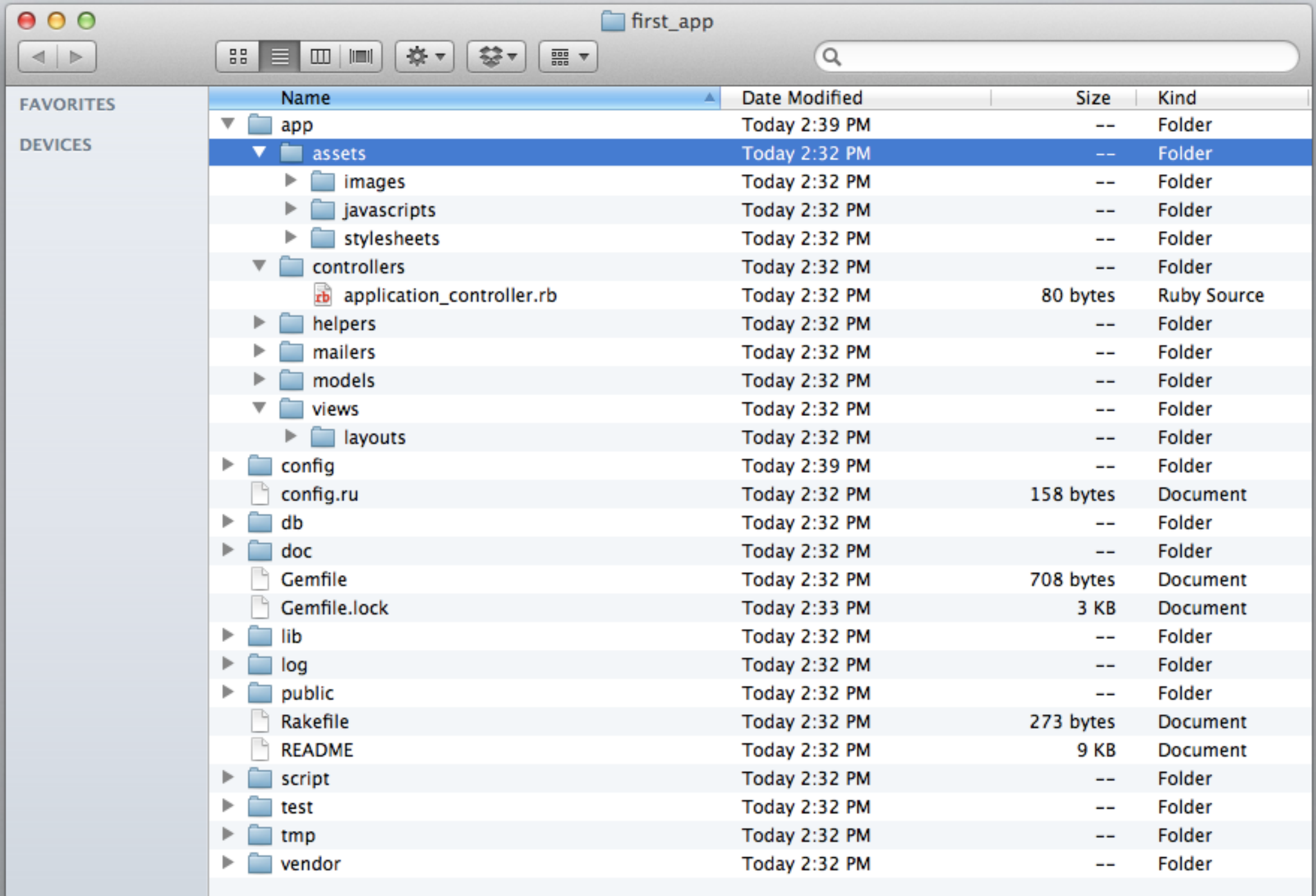
# CREACIÓN DE UNA APP

```
# Creación de una app  
$ rails new nombre_app
```

.....

```
# Variaciones  
-m "plantilla.rb"      # usa un template  
-d "base_de_datos"     # modifica el adapter  
-edge                  # usa vers. edge
```

# FICHEROS



Name	Date Modified	Size	Kind
app	Today 2:39 PM	--	Folder
assets	Today 2:32 PM	--	Folder
images	Today 2:32 PM	--	Folder
javascripts	Today 2:32 PM	--	Folder
stylesheets	Today 2:32 PM	--	Folder
controllers	Today 2:32 PM	--	Folder
application_controller.rb	Today 2:32 PM	80 bytes	Ruby Source
helpers	Today 2:32 PM	--	Folder
mailers	Today 2:32 PM	--	Folder
models	Today 2:32 PM	--	Folder
views	Today 2:32 PM	--	Folder
layouts	Today 2:32 PM	--	Folder
config	Today 2:39 PM	--	Folder
config.ru	Today 2:32 PM	158 bytes	Document
db	Today 2:32 PM	--	Folder
doc	Today 2:32 PM	--	Folder
Gemfile	Today 2:32 PM	708 bytes	Document
Gemfile.lock	Today 2:33 PM	3 KB	Document
lib	Today 2:32 PM	--	Folder
log	Today 2:32 PM	--	Folder
public	Today 2:32 PM	--	Folder
Rakefile	Today 2:32 PM	273 bytes	Document
README	Today 2:32 PM	9 KB	Document
script	Today 2:32 PM	--	Folder
test	Today 2:32 PM	--	Folder
tmp	Today 2:32 PM	--	Folder
vendor	Today 2:32 PM	--	Folder

# ENTORNOS & CONF

- En la carpeta environments encontramos diferentes configuraciones. Tres por defecto.
  - Development
  - Test
  - Production
- database.yml guarda la configuración de la bbdd

# SCAFFOLDING

- Técnica de andamiaje
  - generación rápida de los principales componentes de un objeto
  - se indica el objeto, sus atributos y el tipo de dato
  - fichero de “migración” , modelo, controlador, vistas, tests, helpers, javascript y css



# SCAFFOLDING

# Creación de un scaffold

\$ rails g scaffold *modelo* [*atributos...*]

# otros generadores

\$ rails g layout *nombre*

\$ rails g model *nombre atributos*

\$ rails g controller *nombre métodos*

# FICHEROS DE MIGRACIÓN

- Contienen las instrucciones para trabajar sobre las tablas de las base de datos
- Son independientes del SGBD
- Gestiona la marcha atrás en caso de error con el rollback

# FICHEROS DE MIGRACIÓN

```
class CreatePosts < ActiveRecord::Migration
  def change
    create_table :posts do |t|
      t.string :titulo
      t.text :contenido

      t.timestamps
    end
  end
end
```

# FICHEROS DE MIGRACIÓN

- Las migraciones además gestionan:
  - carga de datos programadas en el `seed.rb`
  - la versión de la bbdd actual y controla que migración debe ejecutar
  - `create`, `drop`, `truncate` de la base de datos



# FICHEROS DE MIGRACIÓN

# Podemos usar los siguientes comandos rake

\$ rake db:create	# crea la bbdd
\$ rake db:migrate	# carga migraciones
\$ rake db:seed	# relleno de bbdd
\$ rake db:rollback	# marcha atrás

# BLOG EN 20 MINUTOS

```
# Vamos a crear nuestra primera app rails
```

```
$ rails new my_blog
```

```
$ cd my_blog
```

```
# configuramos la base de datos
```

```
$ rails g scaffold Post title:string  
content:text
```

```
$ rails g scaffold Comment content:text  
post:references
```

```
$ rake db:create
```

```
$ rake db:migrate
```

```
$ rails server
```

# COMPONENTES DEL FRAMEWORK

# PRINCIPALES

- ActiveRecord
  - Validaciones, asociaciones, callback, interfaz de consulta
- ActionView
  - Layouts, renders, parciales y helpers
- ActionController
- ActiveSupport



# ACTIVE RECORD (AR)

- Representa la M del MVC. Nos proporciona una capa de acceso a los datos, lógica y persistencia.
- Baso en un patrón diseñado por Martin Fowler
- Sigue los principios de convención sobre configuración.

# AR CONVENCION

- Nombre de tablas: plural con guión bajo para separar las palabras.
- Modelos: Singular y con las palabras juntas y capitalizadas.
- Claves primarias: campo **id**
- Claves foráneas: nombre de la tabla en singular + “\_id”

# AR CONVENCION

- created\_at / updated\_at para la gestión de “versiones”
- lock\_version: provoca que se use la estrategia de bloqueo optimista en un modelo
- type: usado para hacer STI (single table inheritance)
- asociacion\_**type**: almacena relaciones polimórficas.

# AR CONVENCIÓN

- Los modelos que extiendan de ActiveRecord::Base se entiende que siguen la convención.
- Podemos modificar la convención:
  - `self.table_name` indica el nombre de la tabla
  - `self.primary_key` indica el nombre de la clave primaria



# AR BÁSICO

- Operaciones CRUD
  - Create
  - Read
  - Update
  - Delete

# AR BÁSICO

# CRUD: Create

```
Post.create(title: "...", content: "...")
```

```
user = User.new
```

```
user.name = "Pablo"
```

```
user.email = "pablo@pabloformoso.com"
```

```
user.save
```

```
user = User.new do |u|
```

```
  ...
```

```
end
```

# AR BÁSICO

# CRUD: Create

```
Post.create(title: "...", content: "...")
```

```
user = User.new
```

```
user.name = "Pablo"
```

```
user.email = "pablo@pabloformoso.com"
```

```
user.save
```

```
user = User.new do |u|
```

```
  ...
```

```
end
```

# AR BÁSICO

# CRUD: Read

`Post.all`

`post_uno = Post.first`

`ultimo_post = Post.last`

`post = Post.find_by(title: "...")`

`post = Post.find_by(id: 2)`

`Post.where(id: 1).order("created_at DESC")`



# AR BÁSICO

```
# CRUD: Update
```

```
post = Post.find_by(id: 2)
```

```
post.update(title: "nuevo")  
post.update_attribute(:title, "nuevo")  
post.update_attributes(title: "nuevo",  
content: "...")
```

```
Post.update_all "title = 'general'"
```

# AR BÁSICO

```
# CRUD: DELETE
```

```
post = Post.find_by(id: 2)  
post.destroy
```

# AR QUERIES

- La interfaz para consultas nos permite lanzar sentencias sencillas y complejas sin mucho esfuerzo
- Haciendo uso del eager loading estas consultas están optimizadas en su gran mayoría.
- No implica que un join o includes puedan generar una query lenta.

# AR QUERIES

- Entre los principales métodos nos encontramos:
  - where, uniq, distinct
  - order, reverse\_order
  - group, from, offset, limit
  - joins, includes



# AR QUERIES

# Recuperación de un solo registro

`Post.take`

`Post.first`

`Post.last`

`Post.find 1`

`Post.find_by title: “...”`

# AR QUERIES

# Recuperación de varios registros

Post.find([1,10])                   # 2 valores  
Post.find([1..10])

Post.take(2)

Post.first(2)  
Post.last(3)

# AR QUERIES

# Recuperación de registros por batch

```
Post.find_each do |p|  
  enviar_por_email(p)  
end
```

```
Post.find_each(batch_size: 2000) do |p|  
  enviar_por_email(p)  
end
```

# AR QUERIES

# Recuperación condicionada

`Post.where(published: true)`

`Post.where("published == 1")`

`Post.joins(:comments).where(  
 comments: {published: true})`

`Post.where(created_at:  
 (Time.now.midnight - 1.day)..Time.now.midnight)`

`Post.where(comments_count: [2,6])`

`Post.where.not(published: true)`



# AR QUERIES

# Recuperación de campos

```
Post.select("title")
```

```
Post.select(:title).distinct
```

# Limites y offsets

```
Post.limit(10)
```

```
Post.limit(2).offset(10)
```

# Agrupaciones

```
Post.where(published: true)  
    .group("comments_count")
```

# AR QUERIES

# Joins

`Post.joins(:category, :comments)`

```
SELECT posts.* FROM posts  
  INNER JOIN categories ON posts.category_id = categories.id  
  INNER JOIN comments ON comments.post_id = posts.id
```

# AR ASSOCIATIONS

- Nos ayudan a establecer diferentes asociaciones entre modelos.
- Simplifica operaciones de dependencia a la hora de crear, leer o destruir un recurso.
- Hace de interfaz a la hora de recuperar una colección de datos.

# AR ASSOCIATIONS

```
# Supongamos estos dos modelos  
class Customer < AR::Base  
end
```

```
class Order < AR::Base  
end
```

```
# Normalmente para eliminar un usuario  
# borraríamos todos los pedidos y luego  
# el usuario
```



# AR ASSOCIATIONS

# Supongamos estos dos modelos

```
class Customer < AR::Base
  has_many :orders, dependent: :destroy
end
```

```
class Order < AR::Base
  belongs_to :customer
end
```

# A través de las dependencias al borrar el  
# comprador se borrarán todos los pedidos  
# :)

# AR ASSOCIATIONS

# Tipos de asociaciones

belongs\_to

has\_one

has\_many

has\_one :through

has\_many :through

has\_and\_belongs\_to\_many

# AR ASSOCIATIONS

```
class Customer < AR::Base
  has_many :orders, dependent: :destroy
end
```

```
class Order < AR::Base
  belongs_to :customer
end
```

El **belongs\_to** nos indica el modelo en el que se espera la clave foránea en la relación, para este caso siguiendo la convención **customer\_id**

# AR ASSOCIATIONS

El `through` nos permite establecer una relación a través de un modelo. En el ejemplo médico, `cita`, `paciente`, se ilustra la vinculación.

```
class Doctor < AR::Base
  has_many :citas
  has_many :pacientes, through: :citas
end
```

```
class Cita < AR::Base ... end
class Paciente < AR::Base ... end
```



# AR ASSOCIATIONS

```
class Doctor < AR::Base
  has_many :citas
  has_many :pacientes, through: :citas
end
```

```
class Cita < AR::Base
  belongs_to :doctor
  belongs_to :paciente
end
```

```
class Paciente < AR::Base
  has_many :citas
  has_many :doctores, through: :citas
end
```

# AR ASSOCIATIONS

- Las relaciones polimórficas esconden una potencia, flexibilidad y tolerancia a cambios. Entraremos en detalle con las nociones avanzadas sobre Rails.

# AR VALIDATIONS

- Por medio de directivas de AR nos permite comprobar si los datos de un objeto cumplen con una especificación.
- Se asegura que los datos son correctos en operaciones de guardado o actualización.

# AR VALIDATIONS

- Métodos que ejecutan validación

create  
create!  
save  
save!  
update  
update!



# AR VALIDATIONS

- Métodos que **NO** ejecutan validación

`toggle!`

`touch`

`update_all`

`update_attribute`

`update_column`

`update_columns`

`update_counters`

`save(validate: false)`

**OJO CON EL  
USO DE ESTOS**

# AR VALIDATIONS

- Comprobaciones sobre objetos

# Objeto guardado

```
p = Post.new(title: "", content: "")
```

```
p.new_record?      => true
```

```
p.save
```

```
p.new_record?      => false
```

# Objeto válido

```
p.valid?
```

```
p.invalid?
```

```
p.errors            => {title:["cant.."]}
```

# AR VALIDATIONS

- Excepciones y retornos

```
# No genera excepción
```

```
p = Post.new(title: "", content: "")
```

```
p.save
```

```
=> false
```

```
# Raise exception
```

```
p.save!
```

```
=> ActiveRecord::RecordInvalid...
```

```
Post.create!
```

```
=> ActiveRecord::RecordInvalid...
```

# AR VALIDATIONS

- Helpers

# Para usar en el modelo

`validates :attr, acceptance: true`

`validates :attr, confirmation: true`

`validates :attr, presence: true`

`validates :attr, length: { minimum: 2 }`

`validates :attr, numericality: true`  
`{ only_integer: true }`

`validates_associated :assoc`



# AR VALIDATIONS

- Helpers

# Para usar en el modelo

`validates :attr, uniqueness: true`

`validates :size, inclusion: { in: %w(small medium large),  
 message: "%{value} is not a valid size"  
}`

`validates :legacy_code, format: { with: /\A[a-zA-Z]+\z/, message: "Only letters allowed" }`

`validates :subdomain, exclusion: { in: %w(www us ca jp),  
 message: "%{value} is reserved." }`

# AR VALIDATIONS

- Personalización validate\_with

```
class GoodnessValidator < ActiveRecord::Validator
  def validate(record)
    if record.first_name == "Evil"
      record.errors[:base] << "This person is evil"
    end
  end
end
```

```
class Person < ActiveRecord::Base
  validates_with GoodnessValidator
end
```

# AR VALIDATIONS

- Personalización validates\_each

```
validates_each :name, :surname do |record, attr, value|  
  record.errors.add(attr, 'must start with upper case')  
  if value =~ /\A[a-z]/  
  end  
end
```

# AR VALIDATIONS

- Validaciones condicionales

```
class Order < ActiveRecord::Base
  validates :card_number, presence: true, if: :paid_with_card?

  def paid_with_card?
    payment_type == "card"
  end
end
```



# AR VALIDATIONS

- Identificación y manejo de errores

# Cada modelo de ActiveRecord tiene un *errors[]*

*p.errors* => Diccionario con attr y mensajes de error

*p.errors[:attr]* => acceso al array de mensajes de error

*p.errors.add :attr, "mensaje de error"*

*p.errors.clear*

*p.errors.size*

# AR VALIDATIONS

- Errores en las vistas

```
<% if @post.errors.any? %>
  <div id="error_explanation">
    <h2><%= pluralize(@post.errors.count, "error") %>
prohibited this post from being saved:</h2>

    <ul>
      <% @post.errors.full_messages.each do |msg| %>
        <li><%= msg %></li>
      <% end %>
    </ul>
  </div>
<% end %>
```

# AR CALLBACKS

- Ciclo de vida de un objeto
  - Create
  - Update
  - Destroy
- Se registran usando *before\_XXXX afeter\_XXXX*

# AR CALLBACKS

- Callbacks de creación

`before_validation`

`after_validation`

`before_save`

`around_save`

`before_create`

`around_create`

`after_create`

`after_save`



# AR CALLBACKS

- Callbacks en updates

`before_validation`

`after_validation`

`before_save`

`around_save`

`before_update`

`around_update`

`after_update`

`after_save`

# AR CALLBACKS

- Callbacks al borrar

before\_destroy  
around\_destroy  
after\_destroy

# AR CALLBACKS

- Callbacks al instanciar un objeto

`after_initialize`  
`after_find`

# AR CALLBACKS

- Métodos que desencadenan callbacks

`create`

`destroy`

`destroy_all`

`save`

`save(validate: false)`

`update_attribute`

`update`

`valid?`

# Y sus variantes con !



# AR CALLBACKS

- Métodos que desencadenan callbacks

```
all
first
find
find_by
find_by_*
find_by_*!
find_by_sql
last
```

# AR CALLBACKS

- Métodos que desencadenan callbacks

```
class Post < ActiveRecord::Base
  after_destroy :log_destroy_action

  def log_destroy_action
    puts 'Post destroyed'
  end
end
```

ROUTING

# RAILS ROUTER

- Se encarga de realizar un matching entre urls y recursos (controladores y acciones)
- Crea un conjunto de métodos REST para separar cada una de las acciones CRUD



# RAILS ROUTER

```
# resource :posts
```

GET	/posts	=>	index
GET	/posts/new	=>	new
POST	/posts	=>	create
GET	/post/:id	=>	show
GET	/post/:id/edit	=>	edit
PUT	/post/:id	=>	update
PATCH	/post/:id	=>	update
DELETE	/post/:id	=>	destroy

# RAILS ROUTER

```
# routes.rb  
# rutas simples
```

```
root to: "welcome#index" # /
```

```
get "home", to: "welcome#index" # /home
```

ROUTING

# RAILS ROUTER

```
# routes.rb
# resources simples y anidados

resources :posts do
  resources :comments
    # comments_post_path(@p) ...
  member do
    get "preview" # preview_post_path(@p)
  end
  collection do
    get "search" # search_posts_path
  end
end
```



# RAILS ROUTER

```
# routes.rb
# Espacios de nombre

namespace :admin
  resources :posts do
    get "statistics" on: :collection
  end
end

admin_statistics_posts_path
# => /admin/posts/statistics
```

# RAILS ROUTER

```
# routes.rb  
# Matching
```

```
match "/landing", to: "welcome#index",  
                  as: :landing
```

```
=> landing_path
```

# ACTION CONTROLLER

# ACTION CONTROLLER

- Representa la C dentro del MVC
- Justo después del matching de rutas se lanza el controlador y acción
- De forma transparente se genera una request y parámetros de la petición recibida



# ACTION CONTROLLER

- Las principales funciones de AC:
  - Primera entrada para sacar el render
  - Gestionar la request
  - Crear los parámetros
  - Mensajes flash
  - Gestión de la respuesta

# ACTION CONTROLLER

- Siempre se ejecutará antes el *application\_controller.rb*
- Es el mejor punto para incluir acciones comunes a todos los controladores o métodos para comprobar parámetros de una request.

# ACTION CONTROLLER

- Layouts y rendering
  - el método layout nos permite modificar la plantilla para un controlador o un conjunto de acciones
  - render dentro de una acción nos permite modificar la vista a cargar en relación a la asignada por convención

# ACTION CONTROLLER

- *params[]*
  - almacena los parámetros extraídos de la URL
  - diccionario en forma de clave valor
  - `get '/clients/:status' => 'clients#index', foo: 'bar'`
    - `params[:status]` `params[:foo]`
    - `params[:action]` `params[:controller]`



# ACTION CONTROLLER

- mensajes **flash**
  - son mensajes volatiles que llevan información temporal como resultado de una operación
  - *redirect\_to xxx\_path, notice: “soy un mensaje”*  
*render ‘edit’, error: “no se ha podido enviar”*  
*render ‘new’, flash: {code: “2 | 23 | 23 |” }*
  - *flash[:notice]* en las vistas nos devuelve el mensaje

# ACTION CONTROLLER

- cookies y session
  - `cookies[:product_id] = @p.id`
  - la sesión se usa para “traquear” un cliente, se puede almacenar en:
    - Cookies
    - Cache
    - ActiveRecord
    - Memcached

# ACTION VIEW

- gestiona la V del MVC
- además es el encargado de gestionar la “response”
- tres elementos principales
  - parciales
  - plantillas de acciones (templates)
  - plantillas globales (layouts)

# ACTION VIEW

- templates
  - usa el taggeado ERB con HTML
  - existen más formatos como XML, JSon o xHr
  - Builder es el “ERB” para XML



# ACTION VIEW

- parciales
- la forma más práctica de reutilizar vistas
- permiten el paso de objetos o variables locales
- existen pequeñas convenciones para facilitar la escritura

# ACTION VIEW

```
<%= render partial: "shared/footer" %>
```

```
<%= render partial: "user",  
      locals: {user: @user} %>
```

```
<%= render "user", object: @user %>
```

# con colecciones

```
<% for post in @posts %>
```

```
  <%= render "post", locals: {post: post} %>
```

```
<% end %>
```

```
<%= render "post", collection: @posts %>
```

```
<%= render @posts %>
```

# ACTION VIEW

```
# para simplificar el código en diseños muy  
# complejos podemos usar una plantilla para  
# separar objetos
```

```
<%= render partial: @products,  
spacer_template: "product_ruler" %>
```

# ACTION VIEW

```
# para simplificar el código en diseños muy  
# complejos podemos usar una plantilla para  
# separar objetos
```

```
<%= render partial: @products,  
spacer_template: "product_ruler" %>
```



# ACTION VIEW

```
# application.html.erb
<html>...
<div><%= yield %></div>
<footer><%= yield :footer %></footer>
```

```
# index.html.erb
<% content_for :footer %>
  Este texto se mete en el pie
<% end %>
```

# METAPROGRAMACIÓN CON RUBY ON/ & RAILS

# METAPROGRAMMING

- “Escribir que software que genera nuevo software”
- En la vida real es algo más complejo. Con Ruby se vuelve algo divertido y natural
  - Monkey patching
  - Dynamic Methods
  - ...
  - Method missing

# MP RUBY/RAILS

# Algunas intrusiones importantes

> send(method, \*attr) # Invoca métodos

> define\_method # Crea métodos

> eval(expr) # Evalúa instrucciones

- class\_eval(expr)

- instance\_eval(expr)

> metthod\_missing # =)



# MP RUBY/RAILS

```
# Monkey Patching  
# Técnica de agregar código a una clase ya  
# definida; clases abiertas.
```

```
class String  
  def hola  
    "hola #{self}"  
  end  
end
```

```
> "Pablo".hola
```

# MP RUBY/RAILS

```
# Dynamic Methods
# Técnica para generar métodos de forma
# dinámica en tiempo de ejecución
class Bicicleta
  MARCHAS = ["primera", ..., "sexta"]

  def va_en_primera?
    marcha == "primera"
  end

  ...

  def va_en_sexta?
    marcha == "primera"
  end
end
```

# MP RUBY/RAILS

```
# Dynamic Methods  
# Si cambiamos de bici solo tenemos que  
# agregar más marchas
```

```
class Bicicleta  
  MARCHAS = ["primera", ..., "sexta"]  
  MARCHAS.each do |m|  
    define_method "va_en_#{m}?" do  
      marcha == m  
    end  
  end  
end
```

```
# Like a sir!
```



# MP RUBY/RAILS

```
# Evals y cadenas de caracteres  
# Todos los eval nos permiten trabajar con  
# expresiones en cadenas y ejecutarlas
```

```
String.class_eval “def self.hola; ‘hola’;  
end”
```

```
nombre = “pablo”  
nombre.instance_eval “def hola; ‘hola’; end”
```

```
# class_eval e instance_eval hacen lo mismo  
# solo cambia el contexto
```



# MP RUBY/RAILS

# Method Missing

# es un método que se ejecuta cada vez que

# no se encuentra el invocado, se combina

# con monkey patching

```
class Producto
```

```
  def method_missing(method, *args, &block)
```

```
    if method =~ /ruby/
```

```
      puts "Tenemos libros de Ruby!"
```

```
    else
```

```
      super
```

```
    end
```

```
end
```

```
producto.ruby
```

```
# => "Tenemos libros..."
```

# METAPROGRAMMING

- Hay que tener cuidado
  - Podemos sobre escribir métodos del core
  - Con strings podemos redefinir sin querer métodos ya existentes
  - Ojo con la inyecciones de código usando evals

# METAPROGRAMMING

- Hay que tener cuidado
  - Tener en cuenta los *métodos fantasma*
  - Method\_missing nos puede jugar malas pasadas si no tenemos acotado su uso
  - Estos métodos fantasma no se comprueban con un respond\_to?

# MP RAILS

```
# Meta helpers  
# Muy usados dentro de rails para crear  
# HTML sin ifs ni lógica por el medio
```

```
def cat_link(cat = "todas")  
  content_tag :div, class: "cat" do  
    link_to "Ver #{cat}",  
      categories_path(cat)  
    class: "category_link"  
  end  
end
```



# AJAX ON RAILS

# AJAX ON RAILS

- Rails nos permite trabajar con peticiones AJAX de forma muy sencillas
  - `link_to remote: true`
  - `link_to_function`
  - `form_for remote: true`

# AJAX ON RAILS

- Para manipular las vistas y las respuesta podemos usar renders normales o ficheros \*.js.erb
- Dentro de los js.erb podemos intercalar erg y javascript con jQuery incluido

# TESTING ON RAILS



# TESTING ON RAILS

- Rails ya está preparado para soportar testing. La carpeta test tiene todo lo necesario para realizar test unitarios, funcionales y de integración.
- Tenemos diferentes variantes. Las más conocidas son:
  - TDD estricto
  - BDD con RSpec
  - BDD a través de historias de usuario

# TESTING ON RAILS

- La filosofía del testing:
  - Llevar a raja Rojo - Verde - Amarillo
  - Hace los test antes que el código
  - Ayuda al debug
  - Ayuda a la documentación

# TESTING ON RAILS

- La clase principal es `Test::Unit`
- Cada fichero extenderá de `Test::Unit::TestCase`

# TESTING ON RAILS

# Ejemplo de fichero de test en Ruby

```
require 'test/unit'
class BooleanTest < Test::Unit::TestCase
  def test_true_is_true
    assert true, "verdad verdadera"
  end
end
```



# TESTING ON RAILS

# Ejemplo de fichero de test en Ruby

# Comprobar si un String es un número con  
# un método is\_number?

#

# - Creamos los tests

# - Probamos, falla

# - Escribimos el código

# - Volvemos a probar

# TESTING ON RAILS

# Diferentes asserts que podemos usar

```
assert condicion  
assert_respond_to obj, method  
assert_nil  
assert_not_nil val  
assert_equal val1, val2  
assert_not_equal  
assert_match regExp, “string”  
assert_not_match  
assert_raise(Exp) { codigo }  
assert_kind_of(Klass, obj)
```

# TESTING ON RAILS

- En Rails los tests extienden de `ActiveSupport::TestCase`
- Los asserts y las vinculaciones de con los modelos las hace ActiveSupport y los `test_helpers`
- La base de datos de testing se limpia tras cada test.

# TESTING ON RAILS

```
# Test de un modelo
```

```
require 'test_helper'
```

```
class ZombieTest < ActiveSupport::TestCase
  def test_zombie_is_valid
    z = Zombie.new
    assert !z.valid?
  end
end
```



# TESTING ON RAILS

- Fixtures son ficheros de yml que nos permiten definir valores para los atributos de un modelo y usarlos en los test
- Accedemos a ellos con el nombre del modelo y con un símbolo al fixture que hemos nombrado

# TESTING ON RAILS

```
# Fixture
```

```
pablo:
```

```
  name: Pablo
```

```
  lastname: Formoso
```

```
# Código
```

```
z = zombies(:pablo)
```

# TESTING ON RAILS

# Relaciones entre Fixtures

pablo:

id: 1

name: Pablo

lastname: Formoso

tweet:

status: "Brainsssss...!"

zombie\_id: 1

# Código

zombies(:pablo).tweets

# TESTING ON RAILS

```
# Comprobar relaciones
```

```
test “debe contener tweets del zombie” do  
  z = zombies(:pablo)  
  assert z.tweets.all? {|t| t.zombie == z}  
end
```



# TESTING ON RAILS

- Para no duplicar fixtures podemos ejecutar un método setup que va a correr antes de cada test
- Importante recordar que no hay persistencia, para cada test se cargaran los fixtures en la base de datos de test, tras cada iteración esta se trunca.

# TESTING ON RAILS

- Podemos crear nuestros propios helpers para los asserts de forma sencilla
- Para que estén disponibles dentro de todos los test los agregamos a los test\_helpers

# TESTING ON RAILS

# assert para validaciones de campos vacíos

```
def assert_presence(model, field)
  model.valid?
  assert_match /can't be blank/,
    model.errors[field].join,
    "#{field} está en blanco"
end
```

# TESTING ON RAILS

- Los test de integración completan el stack completo
- Para ellos hacemos test de caja negra
- Se basan en `ActionDispatch::IntegrationTest`



# TESTING ON RAILS

# Métodos para la “navegación”

get root\_path

post root\_path, user: {name: “Pablo”}

put root\_path, user: {name: “Pablo”}

delete root\_path

follow\_redirect!

# TESTING ON RAILS

```
# Métodos para los test
assert_response :success
assert_response 200
assert_redirected_to root_url
assert_tag "a", attributes: {href: root_url}
assert_no_tag
assert_select "h1", "Book Shop"
```

# TESTING ON RAILS

```
# Métodos para los test
assert_response :success
assert_response 200
assert_redirected_to root_url
assert_tag "a", attributes: {href: root_url}
assert_no_tag
assert_select "h1", "Book Shop"
```

# TESTING ON RAILS

```
require 'test_helper'
```

```
class ZombieTest < ActionDispatch::IntegrationTest
  test "la cabecera es BookShop" do
    get root_path
    assert_response :success
    assert_select "h1", "BookShop"
  end
end
```



# TESTING ON RAILS RSPEC

# TESTING ON RAILS

- Es uno de los frameworks de testing más populares fuera de los que viene por defecto
- Se aproxima más al lenguaje natural
- La salida es más clara
- Se basa en describir el comportamiento

# TESTING ON RAILS

```
require 'spec_helper'
```

```
describe "A Zombie" do  
  it "se llama Pablo"  
end
```

```
# Esto nos genera un pending
```

# TESTING ON RAILS

```
require 'spec_helper'

describe Zombie do
  it "se llama Pablo" do
    zombie = Zombie.new
    zombie.name.should == "Pablo"
  end
end

# El should es el nuevo assert
```



APIS ON RAILS

# API ON RAILS

- Rails tiene una capacidad muy alta de responder a diferentes formato
- Eso sumado a la capacidad de Rails para gestionar recursos, rutas, restricciones etc. hace Rails un framework muy potente para crear APIs

# API ON RAILS

- La interacción de con una API sigue tambien el MVC, donde muchas veces la V será un render de un objeto a JSon

# TESTING ON RAILS

```
# Las rutas permiten que las personalizemos  
# hasta el mínimo detalle
```

```
resources :products, except: :destroy  
# crea todas la rutas menos la de destroy
```

```
resources :products, only: :index
```

```
with_options only: :index do |list|  
  list.resources :products  
  list.resources :categories  
end
```



# API ON RAILS

- Constraints son restricciones que aplicamos a unas determinadas rutas.
- Las constraints se pueden aplicar a subdominios, expresiones con las rutas y elementos de la cabecera.

# TESTING ON RAILS

```
# routes.rb
```

```
resources :products, constraints:  
{ subdomain: "api" }
```

```
constraints subdomain: "api" do  
  resources :products  
  resources :categories  
end
```

```
# TIP editar el /etc/hosts para probarlos
```

# TESTING ON RAILS

```
# routes.rb
```

```
resources :products, constraints:  
{ subdomain: "api" }
```

```
constraints subdomain: "api" do  
  resources :products  
  resources :categories  
end
```

```
# TIP editar el /etc/hosts para probarlos
```

# API ON RAILS

- Es muy importante mantener la api ordenada y separar los controladores de las diferentes partes de la aplicación “admin”, “front”, “api”



# TESTING ON RAILS

```
# routes.rb
```

```
constraints subdomain: "api" do
  namespace :api do
    resources :products
  end
end
```

```
# El controlador
module Api
  class ProductsController... end
end
```

# API ON RAILS

- Otra buena práctica es mantener diferentes versiones de la API para poder mantener la compatibilidad con versiones antiguas de las apps.

# TESTING ON RAILS

```
# routes.rb
```

```
constraints subdomain: "api" do
  namespace :api, path: '/' do
    resources :products
  end
end
```

```
scope module: :v2,
  constraints: ApiConstraints.new(
    version: 2,
    default: :false) do
  ...
end
```

# API ON RAILS

- En el ejemplo anterior vemos el uso de scopes
- Estos hacen referencia a **modules**
- Se les pueden aplicar constraints



# API ON RAILS

- ApiConstraints es un objeto que creamos para atender la versión y separa hacia uno u otro **scope**
- Atención con el default, el scope que lo tenga puede ser accedido sin indicar el número de versión

# TESTING ON RAILS

```
class ApiConstraints
  def initialize(options)
    @version = options[:version]
    @default = options[:default]
  end

  def matches?(req)
    @default ||
    req.headers['Accept'].include?(
      "application/vnd.feiron.v#{@version}")
  end
end
```

# API ON RAILS

- Dentro del namespace podemos tratar las cabeceras de todas las llamadas.
- Esto nos permite hacer una autenticación por token y restringir el acceso.

# TESTING ON RAILS

```
include ActionController::HttpAuthentication::Token::ControllerMethods

respond_to :json
before_filter :restrict_access

private

def restrict_access
  authenticate_or_request_with_http_token do |token, options|
    @api_session = ApiSession.find_by_token(token)
    if @api_session and
      (@api_session.expiration_date >= DateTime.current) and
      (@api_session.digest ==
Digest::SHA512.hex.digest("#{token}:@{api_session.random}"))
      return true
    else
      head :unauthorized
      return false
    end
  end
end
end
```



DEV OPS

# API ON RAILS

- Las mejores combinaciones son:
  - Apache + mod\_rails (Passenger)
  - NGiNX + Unicorn
- Para hacer os despliegues la mejor opción es Capistrano.