



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

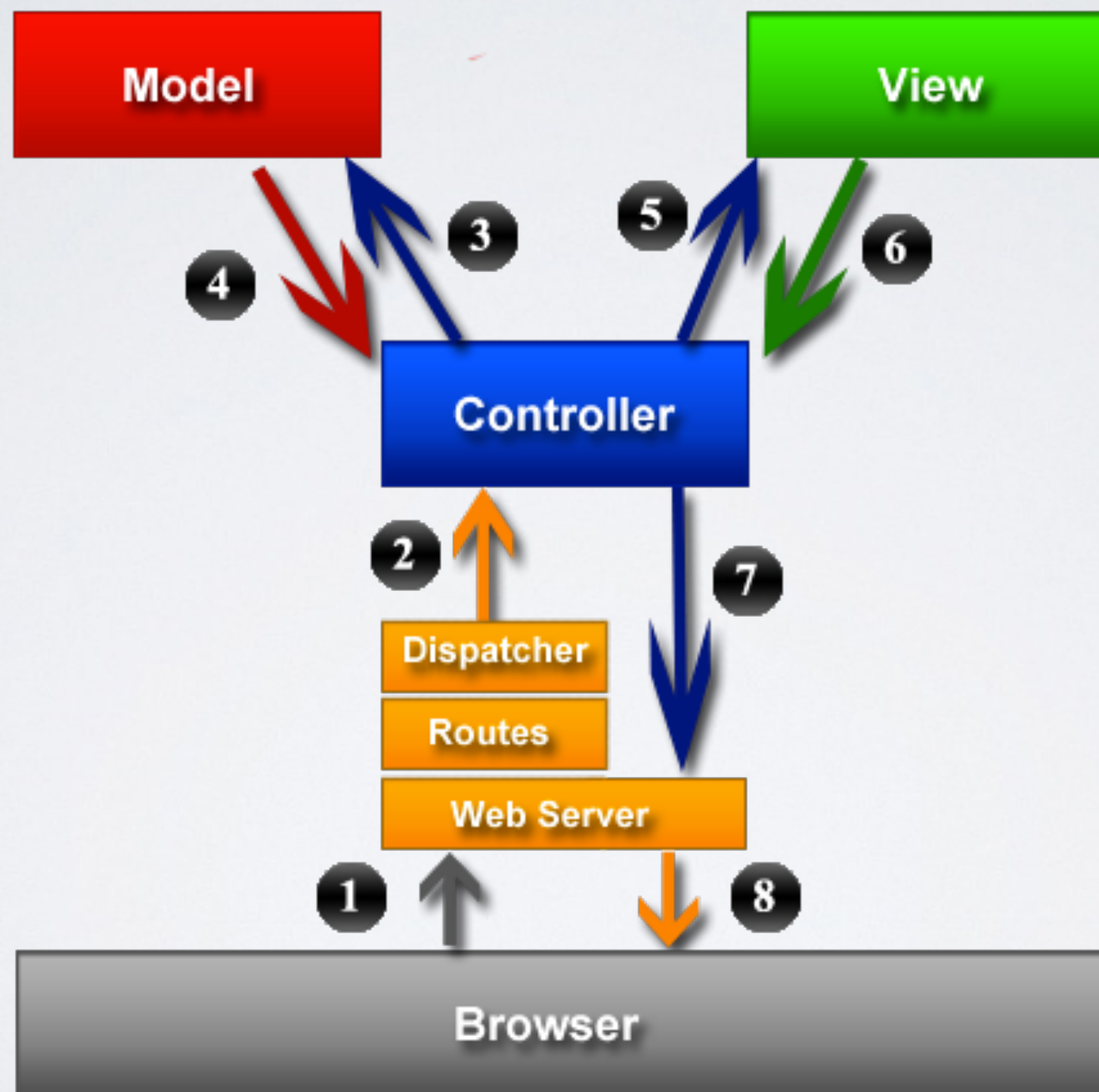
CARACTERÍSTICAS

- 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



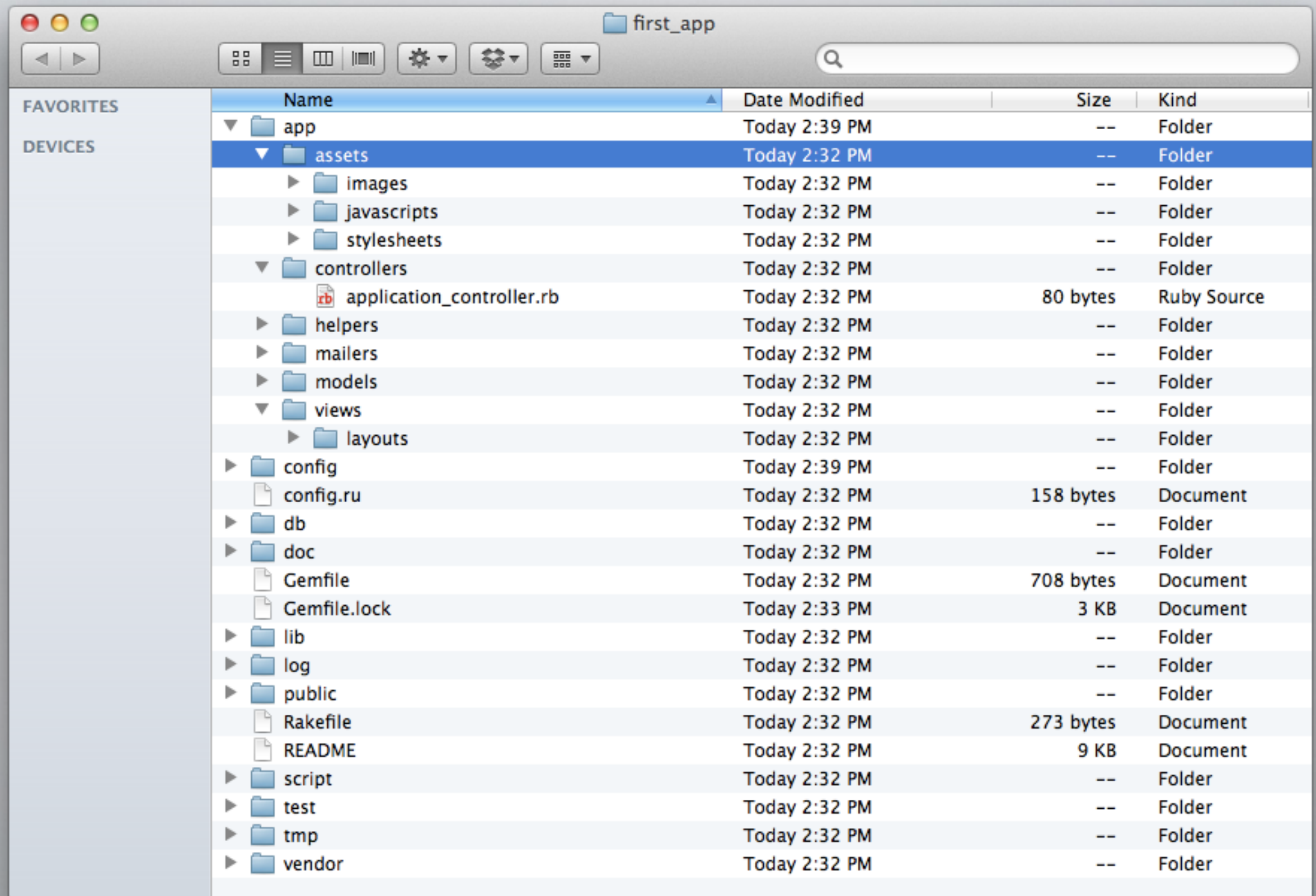
CREACIÓN DE UNA APP

```
# 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



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 CONVENCION

- 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
```

Límites 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_XXXXX* *afeter_XXXXX*

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

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: "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

- layout
 - son el contendor de más alto nivel
 - **yield** indica donde se va insertar el código de la plantilla
 - yield + content_for nos permite modificar partes de una plantilla

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 %>
```