# Ruby On Rails

# Profesor

Alexandre Freire García

@xandrefreire

http://alexandrefreire.com

# Instalar Ruby

## en macOS

# Instalar ruby

>> Pasos:

1. Aceptar las Condiciones de Uso de Xcode
2. Instalar Homebrew
3. Instalar RVM
4. Escoger/instalar ruby 2.3.1

# Instalar ruby

- Aceptar las Condiciones de Uso de Xcode
    - Instalar Xcode y aceptar las condiciones de uso.

# Instalar ruby

- Instalar homebrew
  - `/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"`

# Instalar ruby

- Instalar RVM
  - `\curl -sSL https://get.rvm.io | bash -s stable --ruby`

# Instalar ruby

- Instalar/escoger ruby 2.3.1
  - `rvm install 2.3.1`

# Instalar Ruby

## en Windows

# Instalar Ruby en Windows

>> Pasos:

- Pensárselo una (dos, o tres veces) antes de elegir Windows como SO para desarrollar.
- Leer la guía de supervivencia para usar Rails y Windows
- Volver a pensar si es la decisión correcta

# Instalar Ruby en Windows

>> Pasos:

- Descargar Railsinstaller
- Actualizar Ruby
- Actualizar Rails

# Desarrollo web en backend

# Desarrollo Web en Backend

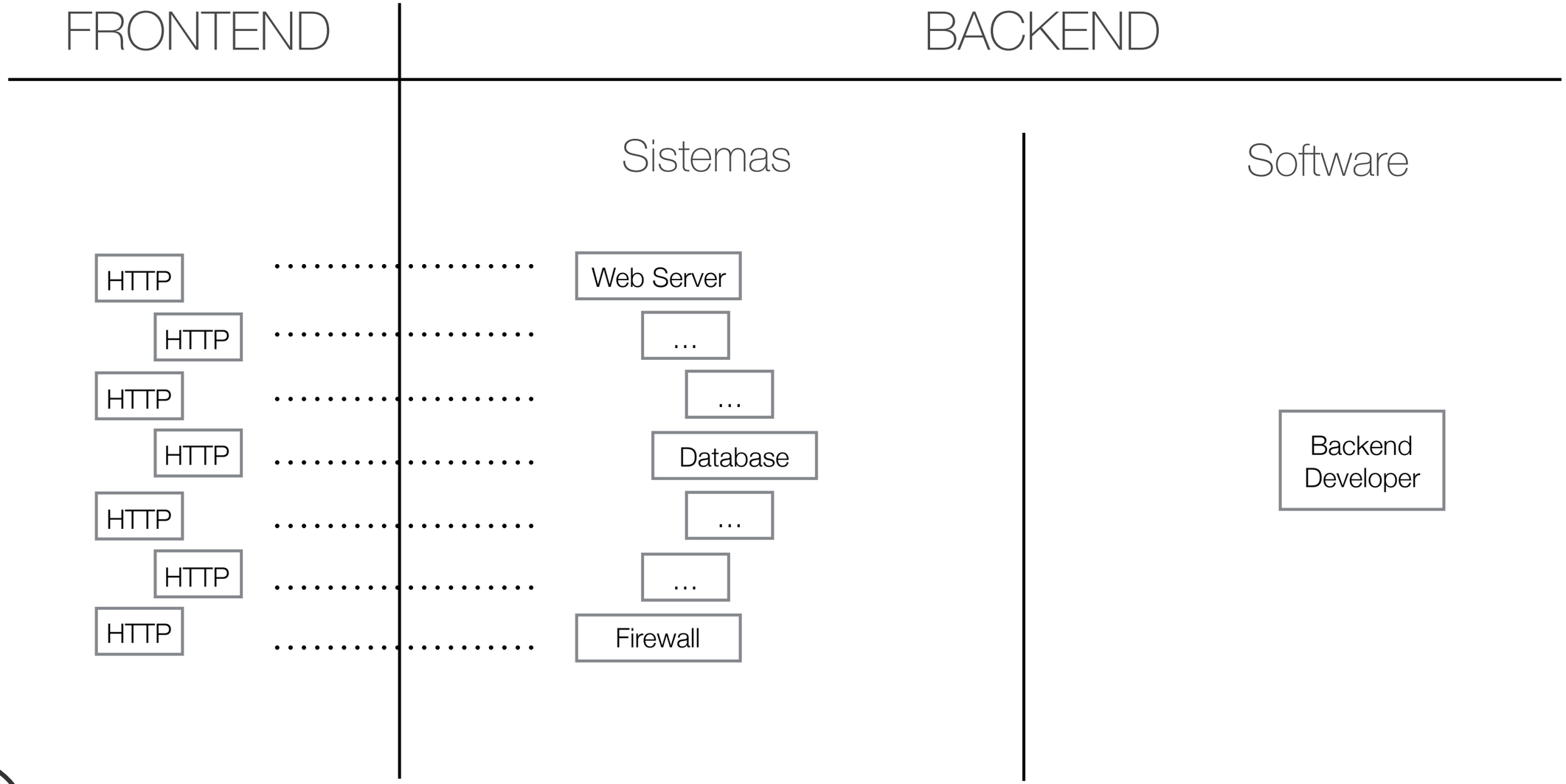
- Corresponde a la parte no visible para el usuario de un sistema
- Utiliza tecnologías que funcionan en el lado de servidor (*server-side*) para responder a peticiones HTTP realizadas desde los clientes (*browsers*, *apps* u otros)

# Desarrollo Web en Backend

>> Sistemas
- Web server
- Database
- Firewall
- …

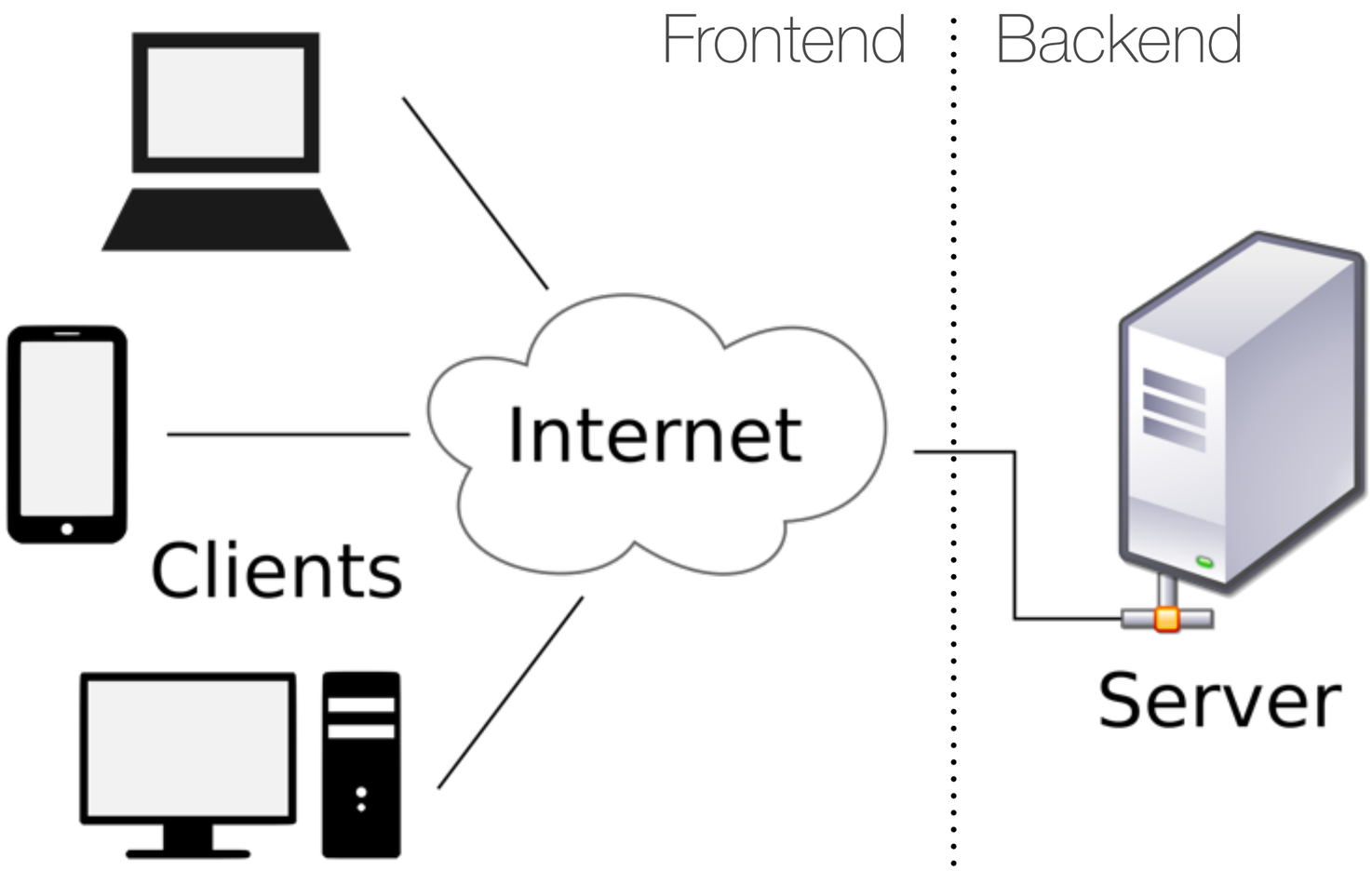
>> Software
- **Backend developer**

# Desarrollo Web en Backend



# Arquitectura Cliente-Servidor

# Arquitectura cliente-servidor



# Arquitectura cliente-servidor

Backend developer

+

Frontend developer

=

**Full Stack Developer**

# Arquitectura cliente-servidor

**RockStar Developer**

**=**

Backend developer

+

Frontend developer

+

App Developer

# Tecnologías de desarrollo backend

# Tecnólogas de desarrollo backend

| Lenguaje | Frameworks |
|---|---|
| Java | Spring, Struts |
| Php | Lavarel, Symfony |
| Ruby | Ruby on Rails |
| Python | Django |
| Swift | Kitura, Perfect, Vapor |
| Otros | node.js, Grails |

# HTTP

## Hypertext Transfer Protocol

# ¿Qué es HTTP?

- Es el protocolo de comunicación más extendido en Internet
- Permite las transferencias de información en la web
- La versión más extendida es la 1.1
- Utiliza varios métodos de petición de datos
- Las respuestas vienen identificadas por un código
- Un browser es un cliente HTTP porque envía peticiones a un servidor web, y éste devuelve las respuestas al cliente.
- HTTP se usa para transmitir recursos

# ¿Qué es un recurso?

>> Un recurso es un trozo de información que puede ser identificado por una URL

- Un recurso puede ser un fichero, un documento, una imagen…

# ¿Qué es una transacción HTTP?

- Un cliente HTTP (un browser) abre una conexión y envía una *request* al servidor HTTP.
- El servidor devuelve una *response*, que contiene el recurso que fue solicitado.
- Después de enviar la respuesta, el servidor HTTP cierra la conexión.

# ¿Qué es una transacción HTTP?

>> El formato de una *request* y de una *response* es similar:

- Linea inicial, diferente si se trata de una request o una response
- Cero o más lineas de cabecera
- Una linea en blanco
- El cuerpo del mensaje
  - Por ejemplo, un fichero.

# Linea inicial de HTTP

>> En una *request*
- 3 partes:
  - El nombre del método (GET, POST, PUT, DELETE, …)
  - El *path* local (users/index.html)
  - Versión de HTTP (HTTP/x.x)

>> En una *response*:
- Versión de HTTP (HTTP/x.x)
- Código de respuesta (200)
- Descripción de la respuesta (OK)

# Códigos de respuesta HTTP

>> Tienen tres dígitos

- 1xx: Información
- 2xx: Éxito
- 3xx: Redireccionamiento a otra URL
- 4xx: Error en la parte de cliente
- 5xx: Error en la parte de servidor

# Cabeceras de HTTP (Headers)

- Da información sobre la *request* o *response*, o sobre el objeto enviado en el cuerpo (*body*) del mensaje
- Formato:
  - “Header name : value”

# Cuerpo del mensaje (*Body*)

- Va después de las lineas de cabecera.
- En una *response*, es donde va el recurso solicitado al cliente, o información de error.
- En una *request*, es donde va la información que es enviada al servidor
- Si hay *request* o *response* con *body*, es común que vaya acompañado de unas lineas de cabecera que describen el cuerpo:
  - Content-Type: text/html, image/gif, …
  - Content-Length: número de bytes en el body

# Ejemplo HTTP

>> *Request*: para obtener el fichero de la URL `http://www.somehost.com/path/file.html`

```
GET /path/file.html HTTP/1.1
Host: www.somehost.com:80
[blank line here]
```

>> El 80 no es obligatorio. Es el puerto por defecto en HTTP/1.1

# Ejemplo HTTP

>> *Response*:

```
HTTP/1.1 200 OK
Date: Fri, 31 Dec 1999 23:59:59 GMT
Content-Type: text/html
Content-Length: 1354
some-footer: some-value
another-footer: another-value

<html>
<body>
<h1>Bienvenido Keepcoder!</h1>
   .
   .
   .
</body>
</html>
```

# Principales métodos HTTP

>> **GET**: se utiliza para obtener información

>> **POST**: se utiliza para enviar o crear información

>> **PUT**: se utiliza para actualizar información

>> **DELETE**: se utiliza para eliminar información

# Sublime Text

## Nuestro editor de texto

# Sublime Text

>> Versión gratuita
>> Potente y versátil
>> Descarga: https://www.sublimetext.com/
>> Package Manager: https://packagecontrol.io/installation

# Otros IDEs

>> Aptana Studio
>> Net Beans
>> RubyMine
- https://www.jetbrains.com/student/

# Introducción

## Ruby

# Introducción a Ruby

>> Sintaxis simple, clara y sencilla

>> Lenguaje interpretado o de script

>> Tipado dinámico

>> Multiplataforma

>> Orientado a objetos

- Todo es un objeto

# Tipos de datos

- >> Fixnum
- >> Bignum
- >> Float
- >> String
- >> Hash
- >> Array
- >> Symbol
- >> Range
- >> NilClass

# Números

>> Numeric
- Fixnum
- Bignum
- Float

>> Operaciones
- +
- -
- *
- /
- **
- %

# Strings

```
name = “Alexandre”
name[0] #=> “A”
name[-1] #=> “e”
puts name #=> “Alexandre”
name.length #=> 9
name.downcase #=> “alexandre”
name.upcase #=> “ALEXANDRE”
```

# Array

```
array = [0, 1, 2, 3, 4, 5]
array = [“Hola”, 0, 8.5, user]

array[2]
#=> 0

array << “Alexandre”
#=> [“Hola”, 0, 8.5, user, “Alexandre”]
```

# Diccionarios o Hashes

```
dict = { key: value}

dict = {name: “Alexandre”}

dict[:name]
#=> “Alexandre”

dict.has_key?(:name)
#=> true
```

# Symbol

>> Es una manera eficiente de representar strings o nombre
>> Son inmutables

```
“alexandre”.object_id #=> 70293204989700
“alexandre”.object_id #=> 70293204947140

:alexandre.object_id #=> 1148188
:alexandre.object_id #=> 1148188
:alexandre.object_id #=> 1148188
```

# Range

```
(1..4)
#=> Todos los valores entre 1 y 4, ambos inclusive

(1…4)
#=> Igual que el anterior, pero no se incluye el
último número
```

# Variables

>> Existen 4 tipos de variables

- **Locales**: empiezan por minúscula o _
- **De instancia**: empiezan por @. Se acceden a ellas a través de los métodos.
- **De clase**: empiezan por @@. Pertenecen y son características de una clase
- **Globales**: empiezan por el símbolo $

# Ruby

- Operadores: `==, <, >, >=, <=, ?valor1:valor2`
- Condicionales: `if, if..else, if..elsif..else, case, unless`
- Bucles: `for, while, until`
- Métodos
- Clases

# Introducción

## Ruby On Rails

# Introducción

>> Ruby On Rails (RoR) es un framework centrado en el desarrollo web escrito en lenguaje Ruby
>> DRY (Don’t Repeat Yourself)
>> Convention over Configuration
>> Rails 5 y Ruby 2.2.2+

# Introducción

>> Utilizado por empresas como:

- Airbnb
- Github
- Groupon
- Zendex
- Themeforest

# Cosas que lo hacen tan maravilloso

# Cosas que lo hacen tan maravilloso

>> Utiliza patrón MVC

>> Dispone de ORM (Object Relational Mapping)

>> Mecanismos del patrón Publisher-Subscriber

>> Autogeneración de código

>> Muy buena documentación

>> Gran soporte por parte de la comunidad

# Cosas que lo hacen tan maravilloso

>> Infinidad de plugins/gemas que nos solucionan muchas tareas

>> Soporte de sesiones

>> Internacionalización

>> Sistema de URL's amigables

>> Incluye un servidor web para desarrollar

# Iniciar el proyecto

# Iniciar el proyecto

$ rails new [nombre del proyecto]

# Estructura del proyecto

>> app/

- assets/
- channels/
- controllers/
- helpers/
- jobs/
- mailers/
- models/
- views/

# Estructura del proyecto

- bin/  contiene el script de rails para iniciar, actualizar o deploy tu app
- config/  configuración de las rutas, base de datos, …
- db/  contiene el esquema actual y las migraciones de la db
- lib/  módulos externos de tu app
- log/  archivos de los logs
- public/  archivos estáticos y assets compilados
- test/  unit test
- tmp/  archivos temporales
- vendor/  código de terceros
- Gemfile  archivo de configuración de gemas

# Arrancando el servidor de desarrollo

# Arrancando el servidor de desarrollo

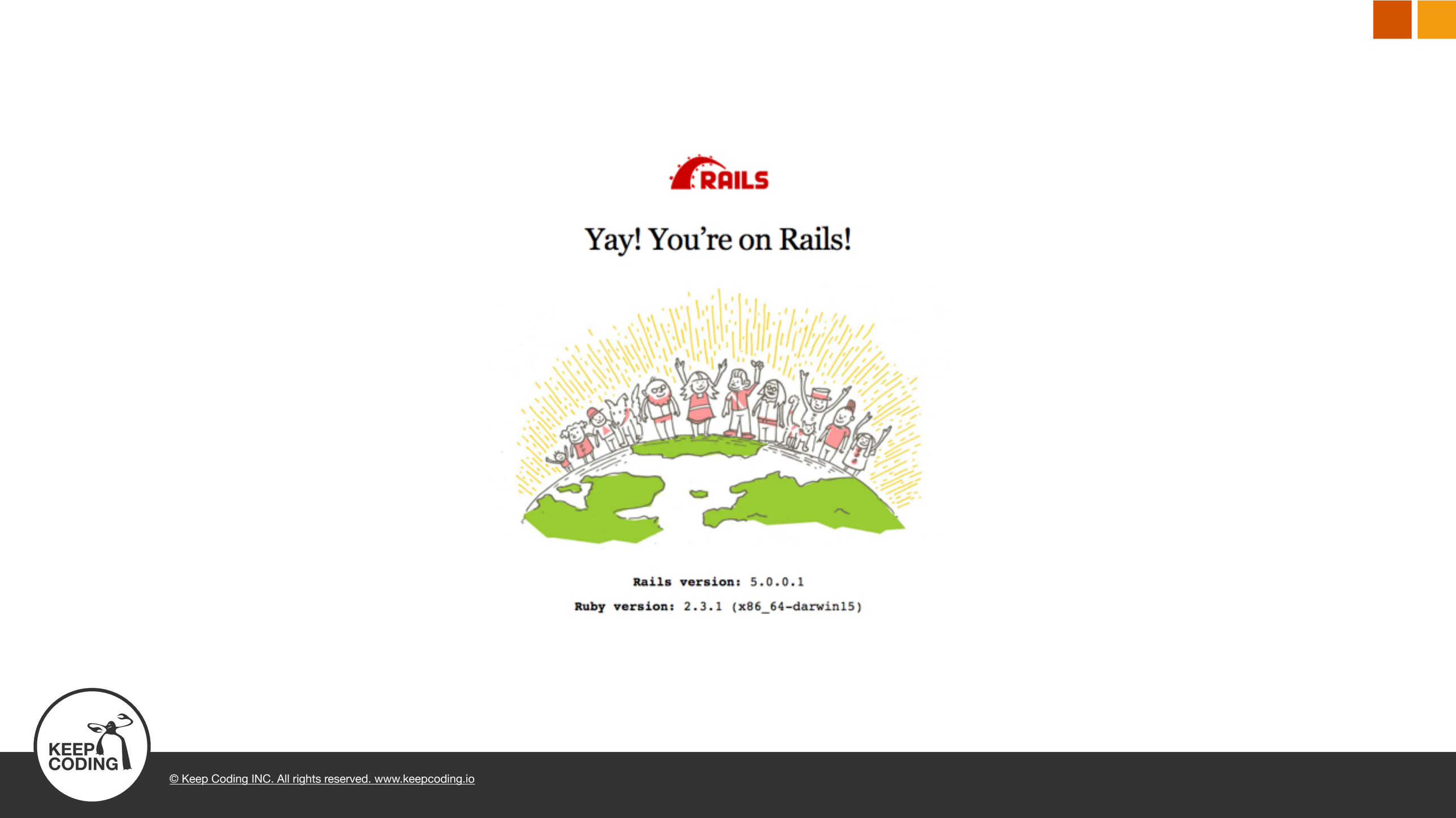
```
$ rails server
```

ó

```
$ rails s
```

# Iniciando el servidor de desarrollo

```
=> Booting Puma
=> Rails 5.0.0.1 application starting in development on http://localhost:3000
=> Run `rails server -h` for more startup options
Puma starting in single mode...
* Version 3.6.0 (ruby 2.3.1-p112), codename: Sleepy Sunday Serenity
* Min threads: 5, max threads: 5
* Environment: development
* Listening on tcp://localhost:3000
Use Ctrl-C to stop
```



# Configurando nuestra app

En la carpeta *config/*

# Configurando nuestra app

>> config/

- environments/
- initializers/
- locales/
- database.yml
- Otros ficheros

# Configurando nuestra app

## >> database.yml

```
default: &default
  adapter: postgresql
  encoding: unicode
  pool: <%=
ENV.fetch("RAILS_MAX_THREADS") { 5 } %>

development:
  <<: *default
  database: frikr_development

test:
  <<: *default
  database: frikr_test

production:
  <<: *default
  database: frikr_production
  username: frikr
  password: <%=
ENV['FRIKR_DATABASE_PASSWORD'] %>
```

# MVC

## El patrón Modelo-Vista-Controlador

# MVC

>> Es un patrón de diseño muy común en tecnologías web

>> También se utiliza en el desarrollo móvil

>> Separa la lógica de negocio de la interfaz de usuario

>> 3 actores

- Modelo
- Vista
- Controlador

# Modelo

>> Representa las entidades de nuestra aplicación

- Usuarios
- Mensajes entre usuarios
- Post de un blog
- Fotos
- etc

# Vista

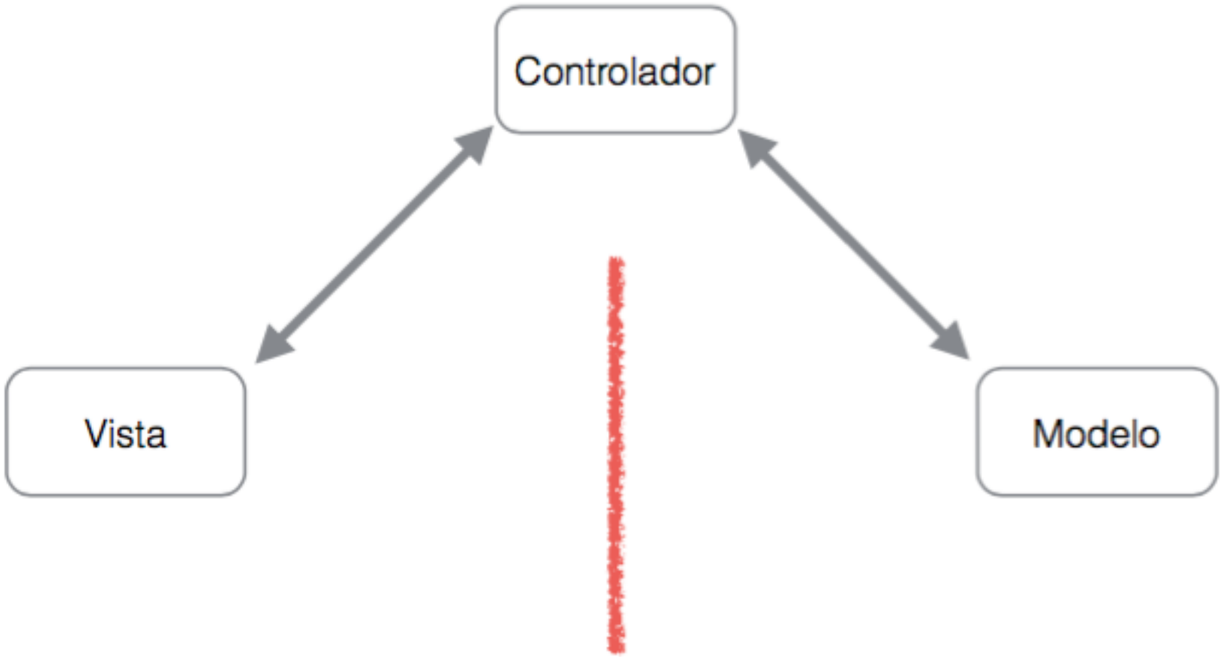
>> La vista se encarga de presentar los datos que el controlador le indica en el formato que se le especifica:

- HTML
- JSON
- XML

# Controlador

- Se encarga de gestionar los modelos y cómo serán éstos presentados (las vistas)
- Es el nexo de unión del patrón
- El modelo y la vista nunca “se tocan”
- Los controladores manejan las URLs que solicita el cliente y se encarga de las gestiones necesarias con los modelos para presentarlos en la vista en el formato que se solicita.

# Controlador



# Los controladores

- Los controladores están formados por funciones o métodos
- Hereda de `ApplicationController`
- A través de las URLs, indicamos qué controlador maneja cada URL
- Convención de nombres: en plural
  - `ClientsController` preferible a `ClientController`
- Los parámetros se reciben en el hash `params`
  - No se hace distinción entre parámetros con `GET` o `POST`

# Controlador

$ rails g controller [nombre controlador] [metodos]

# Los modelos

>> ActiveRecord como ORM

- Representan los modelos y su información
- Representa las relaciones entre modelos
- Herencia
- Valida los modelos antes de persistirlos
- No hace falta crear un campo id, Rails siempre genera uno por defecto

# Los modelos

>> ActiveRecord como ORM

- Al heredar de ActiveRecord, se implementan automáticamente los métodos save, update y destroy que crean/actualiza/eliminan objetos de la base de datos.

>> Un modelo se crea así:

```
$ rails g model [Nombre del modelo] atributo:tipo atributo2:tipo2 …
```

# Peticiones

- Recuperar todos los objetos:
  - `Model.all`
- Recuperación filtrada:
  - `Model.where(atributo: valor)`
- Recuperación de un sólo elemento
  - `Model.find(2) #recupera el modelo con id = 2`
    - Lanza excepción si no encuentra ningún registro con ese id.
  - `Model.find_by(<atributo>:<valor>)`
    - Ej: `Model.find_by(name: 'Alexandre')`

# Validación de modelos

>> Rails proporciona un método de validación de modelos que se ejecuta siempre antes del guardado de un modelo en la base de datos.

```
class Person < ApplicationRecord
  validates :name, presence: true
end

Person.create(name: “Alexandre Freire").valid? # => true
Person.create(name: nil).valid? # => false
```

# Validación de modelos

>> errors[ ]: Cuando un objeto no es válido, podemos preguntar el porqué.

- `errors.details[<atributo>]`
  `# => [{error: :blank}]`

# Relaciones entre modelos

>> http://guides.rubyonrails.org/association_basics.html

>> belongs-to

>> has_one

>> has_many

>> has_and_belongs_to_many

>> belongs-to_and_has_one

>> …

# Convención de nombres

>> **Tabla de la base de datos**: plural con palabras separadas con guiones bajos
>> **Clases**: singular con la primera en mayúscula

| Model / Class | Table / Schema |
|---|---|
| `Article` | `articles` |
| `LineItem` | `line_items` |
| `Deer` | `deers` |
| `Mouse` | `mice` |
| `Person` | `people` |

# Convención de nombres

>> Rails pluraliza automáticamente los nombres. Si una palabra tiene su plural irregular, se puede especificar en el fichero *inflections.rb* dentro de *config/initializers*

```
ActiveSupport::Inflector.inflections(:en) do |inflect|
  inflect.plural /^(ox)$/i, '\1en'
  inflect.singular /^(ox)en/i, '\1'
  inflect.irregular 'person', 'people'
  inflect.uncountable %w( fish sheep )
end
```

# Creando la base de datos

## Con la estructura de nuestros modelos

# Creando la base de datos

- El lenguaje SQL
- Claves únicas, claves foráneas
- CREATE
- SELECT
- INSERT
- UPDATE
- DELETE

# Olvídate del SQL

## con Ruby On Rails

# Creando la base de datos

>> Para crear la base de datos:

- $ rake db:create

# El admin de Rails

## Con active-admin

# El admin de Rails

- Utilizaremos la gema ‘active-admin’
- Nos proporciona una interfaz web de administración basado en los modelos de nuestra app
- Con poco código, podemos hacer que nuestras apps sean fácilmente gestionables desde una web que Active-Admin crea por nosotros.

# El admin de Rails

- Debemos añadir ‘active-admin’, ’inherited_resources’ (Rails 5) y ‘devise’ a nuestro fichero Gemfile
- En el terminal, ejecutar
  - $ bundle install
- Instalar e inicializar Active Admin
  - $ rails g active_admin:install
    - Esto nos generará una serie de archivos de configuración, así como un modelo AdminUser para autenticarse en la página de administración.
- Registrar nuestros modelos en Active Admin:
  - $ rails g active_admin:resource [Modelo]

# Las vistas

## El aspecto de nuestra app

# Las vistas

- Las vistas en Rails es el código HTML (ó XML, JSON, …) que devolvemos desde un controlador
- Definen cómo se presenta la información
- En el caso de HTML, Rails nos proporciona un potente sistema de plantillas (plantillas erb)

# Las vistas

- Se almacenan en nuestra carpeta *app/views/*
- Cada controlador tendrá su propia subcarpeta dentro de *app/views*
  - Ej: el controlador de usuarios
    - *app/views/**users/index.html.erb***
- El sistema de plantillas permite la reutilización (header, footer, etc) y utiliza una sintaxis muy sencilla.

# Las vistas

- El trabajo sucio ha de hacerlo en controlador. La vista, cuanto menos lógica contenga, mejor.
- Las vistas reciben datos en forma de variables que crea su controlador.

# Cómo cargamos una vista desde el controlador

>> `render :index`
  - Cargará la vista llamada index

>> Si no se especifica la vista en concreto que se quiere renderizar, se cogerá aquella vista que se llame exactamente igual que el método.

>> `redirect_to`

# Sintaxis de plantillas

# Sintaxis de plantillas

>> Rails utiliza un sistema de plantillas para el que no se requiere conocimientos de ruby (esa es la teoría)
>> La idea es que los maquetadores de frontend puedan maquetar sin saber backend

# Impresiona de datos

`<%= @user %>`

Pinta el valor de la variable `@user`

`<%= @user.name %>`

Pinta el atributo “`name`” del objeto `@user`

# Sintaxis para las instrucciones

```
<% INSTRUCCIONES %>
```

Las instrucciones van entre <% %>

# Instrucciones IF

```
<% if user_signed_in? %>
 <div> Name: <%= @user.name %> </div>
<% else %>
 <div> Usuario no autenticado </div>
<% end %>
```

# Instrucciones FOR

```
<% @users.find_each do |user| %>
<div> Name: <%= user.name %> </div>
<% end %>
```

# Reutilización de vistas

- Rails permite extender una plantilla base y reescribir diferentes partes
- Muy útiles para webs donde tenemos que mantener un mismo header, footer, o que tengan diferentes partes en común, por ejemplo, un menú.

# Reutilización de vistas

>> Con la palabra reservada yield, se identifica una sección donde el contenido de otra vista debe ser insertado.

>> Se pueden crear múltiples “regiones” yield.

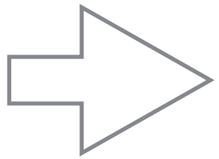
```
<html>
  <head>
  <%= yield :head %>
  </head>
  <body>
  <%= yield %>
  </body>
</html>
```

# Reutilización de vistas

>> Con `content_for` podemos rellenar las diferentes regiones:

```
<% content_for :head do %>
  <title>A simple page</title>
<% end %>

<p>Hello, Rails!</p>
```



```
<html>
  <head>
  <title>A simple page</title>
  </head>
  <body>
  <p>Hello, Rails!</p>
  </body>
</html>
```

# Incluyendo otras vistas

>> Se pueden incluir fragmentos de código comunes con vistas parciales

>> Estas vistas deben comenzar por un guión bajo

- ej: *_user_form.html.erb*

>> Para incluirlos en las vistas:

```
<%= render partial: “user_form” %>
```

>> Si el partial utiliza objetos:

```
<%= render partial: “user_form”, object: @user %>
```

# Migraciones

# Como crear una migración

$ rails g migration [Nombre descriptivo de la migración]

Ejemplo:
$ rails g migration AddPartNumberToProducts

```
class AddPartNumberToProducts < ActiveRecord::Migration[5.0]
  def change
  end
end
```

# Cómo crear una migración

>> Si la migración la creas con los nombres AddXXXToYYY ó RemoveXXXToYYY y se lista una serie de nombres de las columnas, la migración contendrá automáticamente los campos `add_column` ó `remove_column`

# Cómo crear una migración

Ejemplo:

$ rails g migration AddPartNumberToProducts part_number:string

```
class AddPartNumberToProducts < ActiveRecord::Migration[5.0]
  def change
    add_column :products, :part_number, :string
  end
end
```

# Cómo crear una migración

>> Permite volver hacia atrás de manera muy sencilla

- $ rake db:rollback
- $ rake db:rollback STEP=3

# Routes

Para configurar y enlazar las URLs con los controladores

# Las rutas en Rails

Verbo CRUD ‘la url’, to: ‘controlador#método’

```
get '/photos/:id', to: 'photos#show'
```

>> Cuando se reciba, por ejemplo, la request `GET /photos/17`, la aplicación ejecutará el método `show` del controlador `photos_controller`

# Las rutas en Rails

>> Se pueden nombrar las rutas de la siguiente manera

```
get '/photos/:id', to: 'photos#show', as: 'photo'
```

>> Y se creará el path correspondiente, de manera que podamos acceder a este endpoint de manera muy sencilla

```
<%= link_to 'Photo Record', photo_path(@photo) %>
```

>> De manera que el router generará el path `/photo/17`

# Rutas en Rails

>> Rails nos proporciona un mecanismo para crear y nombrar las rutas más típicas de cada objeto
>> En lugar de declarar una ruta para `index`, `show`, `new`, `edit`, `create`, `update`, y `destroy`, Rails nos proporciona un atajo.

```
resources :photos
```

# Las Rutas en Rails

>> Cuando utilizamos resources :photos, Rails crea siete rutas en tu app:

| HTTP Verb | Path | Controller#Action | Used for |
|---|---|---|---|
| GET | /photos | photos#index | display a list of all photos |
| GET | /photos/new | photos#new | return an HTML form for creating a new photo |
| POST | /photos | photos#create | create a new photo |
| GET | /photos/:id | photos#show | display a specific photo |
| GET | /photos/:id/edit | photos#edit | return an HTML form for editing a photo |
| PATCH/PUT | /photos/:id | photos#update | update a specific photo |
| DELETE | /photos/:id | photos#destroy | delete a specific photo |

# Las Rutas en Rails

- Rails resuelve las rutas en el orden que son especificadas.
- Esto es, si tienes `resources :photos` declarado en tu archivo de rutas, y se recibe `GET /photos/poll`, la acción `show` sería la que se ejecutaría.
- Para resolver esto, simplemente declara esta ruta específica antes de `resources :photos`

# Devise

Tu mejor solución para la autenticación de tus usuarios

# Devise

>> Está compuesto por 10 módulos:

- Database Authenticatable
- Omniauthable
- Confirmable
- Recoverable
- Registerable
- Rememberable
- Trackable
- Timeoutable
- Validatable
- Lockable

# Devise

>> Añadir ‘devise’ a tu fichero Gemfile

>> Ejecutar:

- $ rails g devise:install

>> En *config/environments/development.rb*

```
config.action_mailer.default_url_options = { host: 'localhost', port: 3000 }
```

>> En el terminal:

- $ rails g devise MODEL

>> Disponibles muchos helpers útiles: `user_signed_in?`, `current_user`, `user_session`

# Devise

>> Para editar las vistas, mailers, etc…

- $ rails g devise:views

# Formularios en Rails

# Formularios

>> En la web, el envío de formularios es algo básico e imprescindible
>> Rails nos proporciona unos helpers para trabajar con ellos de una manera muy sencilla

# Formularios

```
<%= form_tag("/search", method: "get") do %>
  <%= label_tag(:q, "Search for:") %>
  <%= text_field_tag(:q) %>
  <%= submit_tag("Search") %>
<% end %>
```

```
<form accept-charset="UTF-8" action="/search" method="get">
  <input name="utf8" type="hidden" value="&#x2713;" />
  <label for="q">Search for:</label>
  <input id="q" name="q" type="text" />
  <input name="commit" type="submit" value="Search" />
</form>
```

# Formularios

>> El form_tag helper acepta dos argumentos

- El path de la acción
- Un hash con opciones

```
form_tag({controller: "people", action: "search"}, method: "get", class: "nifty_form")
# => '<form accept-charset="UTF-8" action="/people/search" method="get" class="nifty_form">'
```

# Formularios

>> Una tarea muy común de los formularios es la de editar o crear un modelo.

>> Para ahorrarse código, Rails proporciona una serie de helpers donde el primer argumento es el objeto que se esta creando/editando.

```
<%= text_field(:person, :name) %>
<input id="person_name" name="person[name]" type="text" value="Henry"/>
```

# Formularios

>> Si el objeto tiene muchos atributos, puede ser repetitivo.

>> Para ello, Rails nos proporciona otra forma de crear el formulario

```
<%= form_for @article, url: {action: "create"}, html: {class: "nifty_form"} do |f| %>
  <%= f.text_field :title %>
  <%= f.text_area :body, size: "60x12" %>
  <%= f.submit "Create" %>
<% end %>

## El HTML resultante
<form accept-charset="UTF-8" action="/articles" method="post" class="nifty_form">
  <input id="article_title" name="article[title]" type="text" />
  <textarea id="article_body" name="article[body]" cols="60" rows="12"></textarea>
  <input name="commit" type="submit" value="Create" />
</form>
```

# Formularios

>> Declarando en el archivo *routes.rb* el modelo como **`resource`**

```
resources :articles
```

>> Podemos declarar el formulario de la siguiente manera

```
## Creating a new article
# long-style:
form_for(@article, url: articles_path)
# same thing, short-style (record identification gets used):
form_for(@article)

## Editing an existing article
# long-style:
form_for(@article, url: article_path(@article), html: {method: "patch"})
# short-style:
form_for(@article)
```

# Formularios

- Se puede comprobar que el short-style para crear y para editar es el mismo
- Rails conoce automágicamente si es un nuevo objeto o si se está editando uno existente y selecciona el path correspondiente
  - `object.new_record?`

# Formularios

>> Strong parameters

- Es necesario añadir la whitelist de los parámetros permitidos, por seguridad

```
def create
  @person = Person.new(person_params)
  # ...
end

private
def person_params
  params.require(:person).permit(:name, address)
end
```

# Archivos estáticos

# Bootstrap

>> Añadir ‘bootstrap-sass’ y ‘autoprefixer-rails’ a nuestro fichero Gemfile

>> Renombre el fichero application.css situado en app/assets/stylesheets/ por application.css.sass

>> Añadir al final del fichero:

- @import “bootstrap-sprockets”
- @import “bootstrap”

>> Ejecutar:

- $ bundle install

# Rack Attack

## Middleware para defenderse de ataques

# Rack Attack

>> Ayuda a defender tu aplicación de:
- Ataques DDoS
- Ataques por fuerza bruta
- Hammering

>> También nos permite monetizar la aplicación tras superarse el límite de uso gratuito

# Rack Attack

>> Nos permite configurar:

- **Whitelist**: uso normal si se cumplen unas condiciones
- **Blacklist**: envío de un mensaje de acceso denegado para determinadas *request*
- **Throttle**: comprobar si el usuario está dentro de sus límites de uso permitido
- **Track**: para tener los logs de las *request*

# Rack Attack

>> Añadimos gem ‘rack-attack’
>> Bundle install
>> Actualizamos config/application.rb

```
class Application < Rails::Application

    # ...

    config.middleware.use Rack::Attack

  end
```

# Rack Attack

## >> Creamos un initializer: rack_attack.rb

```
class Rack::Attack

  # `Rack::Attack` está configurado para usar la 'Rails.cache' por defecto,
  #  pero podemos cambiar esta propiedad cambiando el  valor
  #  de `Rack::Attack.cache.store`
  Rack::Attack.cache.store = ActiveSupport::Cache::MemoryStore.new

  # Permitimos todo el tráfico local
  whitelist('allow-localhost') do |req|
    '127.0.0.1' == req.ip || '::1' == req.ip
  end

  # Permitimos a una misma dirección IP hacer 5 request cada 5 segudnos
  throttle('req/ip', limit: 5, period: 5) do |req|
    req.ip
  end

  # Mensaje para los usuarios que exceden el límite de uso
  self.throttled_response = ->(env) {
    retry_after = (env['rack.attack.match_data'] || {})[:period]
    [
      429,
      {'Content-Type' => 'application/json', 'Retry-After' => retry_after.to_s},
      [{error: "Throttle limit reached. Retry later."}.to_json]
    ]
  }
end
```

# Internationalization

tu app en varios idiomas

# Internationalization

>> Rails posee una potente y sencilla herramienta para que puedas dar soporte a varios idiomas en tu app
>> Utiliza i18n
>> Ofrece varias maneras de implementación
- A través de la url
- A través de headers
- A través de las preferencias de usuarios
- …

# A través de la url

>> Dentro de esta opción, también podemos escoger entre:
  - Diferenciar entre .com, .es, .pt, …
  - añadir un parámetro a la url:
    - www.example.com/resources?locale=es
  - Mediante un subdominio
    - www.example.com/es/resources

>> Implementaremos las dos últimas

# A través de la url

>> En el archivo *config/application.rb* añadimos:

```
# ...
module Frikr
  class Application < Rails::Application
    # Settings in config/environments/* take precedence over those specified here.
    # Application configuration should go into files in config/initializers
    # -- all .rb files in that directory are automatically loaded.
    config.i18n.default_locale = :es
  end
end
```

>> Para añadir el idioma por defecto de nuestra app

- Si no se especifica, por defecto es en inglés

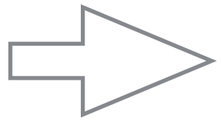
# A través de la url

>> Dentro de la carpeta *config/locales/* creamos un nuevo archivo llamado *es.yml* y copiamos el contenido de *en.yml*

>> Traducimos las claves al idioma correspondiente.

```
en:
  hello: "Hello world"

  photos:
    list:
      title: 'Photo list'
```



```
es:
  hello: "Hello world"

  photos:
    list:
      title: 'Lista de fotos'
```

>> Muy importante, identar con espacios y no con tabs.

- Se puede configurar Sublime para que el tab inserte espacios.

# A través de la url

>> Modificamos el archivo *application_controller.rb*

```
class ApplicationController < ActionController::Base

  # …

  before_action :set_locale


  # Asigna o a lee el parámetro locale que viene en el url antes de
  # cualquier acción
  def set_locale
    I18n.locale = params[:locale] || I18n.default_locale
  end

  # Agrega, por defecto, el parámetro locale a nuestras urls
  def default_url_options(options={})
    { locale: I18n.locale }
  end

  # …

end
```

# A través de la url

>> En las vistas, por ejemplo en *list.html.erb* cambiamos el string “hardcodeado” por la clave de nuestro archivo de locales.

>> Cambiamos esto:

```
<% content_for :title do %> Un listado de fotos <% end %>
<h1>Listado de fotos</h1>
<%= render partial: "photos_loop", object: @photos%>
```

>> Por esto:

```
<% content_for :title do %> Un listado de fotos <% end %>
<h1><%= t '.title' %></h1>
<%= render partial: "photos_loop", object: @photos%>
```

# A través de la url

>> Ahora debemos elegir el estilo que queremos que tengan nuestras rutas:
  - localhost:3000/photos?locale=es
  - localhost:300/es/photos

>> Si nos importa la legibilidad de la URL, escogeremos la segunda.
  - Para ello, es necesario cambiar nuestro fichero *routes.rb*

# A través de la url

>> En *routes.rb* añadimos

```
scope "(:locale)" do
  get 'photos',           to: 'photos#list' ,       as: 'photos_list'
  get 'photos/my_photos', to: 'photos#my_photos',   as: 'my_photos'
  resources :photos
end
```

>> Esto añadirá un /es/ a todas las rutas que contenga hasta el `end`.

# GRACIAS

www.keepcoding.io