Ruby on Rails

Controller Actions

Overview

- Index
- Show
- Destroy
- New
- Create
- Edit
- Update

INDEX

Index

- Obtiene todos los posts
- Busca implícitamente al template **index.html.erb** para renderizar

```ruby
class PostsController < ApplicationController
  # GET /posts
  # GET /posts.json
  def index
    @posts = Post.all
  end
```

index.html.erb

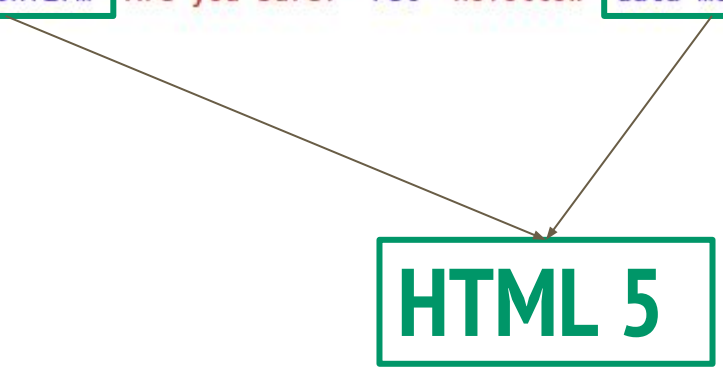
post = post_path

Si los navegadores solo soportan los métodos GET/POST, como le decimos a rails que maneje un request como DELETE?

```
<tbody>
  <% @posts.each do |post| %>
    <tr>
      <td><%= post.title %></td>
      <td><%= post.content %></td>
      <td><%= link_to 'Show', post %></td>
      <td><%= link_to 'Edit', edit_post_path(post) %></td>
      <td><%= link_to 'Destroy', post, method: :delete, data: { confirm: 'Are you sure?' } %></td>
    </tr>
  <% end %>
</tbody>
```

index.html.erb

```
<tbody>
      <tr>
        <td>RoR: Introduction</td>
        <td>This is the content!</td>
        <td><a href="/posts/1">Show</a></td>
        <td><a href="/posts/1/edit">Edit</a></td>
        <td><a data-confirm="Are you sure?" rel="nofollow" data-method="delete" href="/posts/1">Destroy</a><
        /td>
      </tr>
  </tbody>
```



GitHub, Inc. [US] | https://github.com/rails/jbuilder

This repository Search Pull requests Issues Gist

rails / jbuilder

Watch 86 Star 2,796 Fork 265

Code Issues 22 Pull requests 10 Projects 0 Wiki Pulse Graphs

Jbuilder: generate JSON objects with a Builder-style DSL

562 commits 6 branches 55 releases 81 contributors MIT

Branch: master New pull request Create new file Upload files Find file Clone or download

davidrunger committed with **rwz** Fix typo/wording in README.md (#367) ... Latest commit ed5e4c6 23 days ago

gemfiles	Bump up rails 5 version in appraisals	10 months ago
lib	Fixed a typo I noticed while debugging a few of my templates. paramat…	2 months ago
test	Avoid warning on test (#360)	2 months ago
.gitignore	Add pkg to gitignore	2 years ago
.travis.yml	Bump up ruby versions for Travis	4 months ago

index.json.builder

index.json.jbuilder

```
1  json.array! @posts, partial: 'posts/post', as: :post
```

localhost:3000/posts.json


```
[
  - {
        id: 1,
        title: "RoR: Introduction",
        content: "This is the content!",
        created_at: "2016-11-22T11:05:13.001Z",
        updated_at: "2016-11-22T11:05:13.001Z",
        url: "http://localhost:3000/posts/1.json"
    }
]
```

Entonces

- El action INDEX permite traer los recurso de la capa de acceso a datos.
- Implícitamente invoca al template HTML o JSON.

SHOW

Show

Obtiene un post específico basado en el parámetro **id** que es pasado como parte de la URL.

Implícitamente busca al template **show.html.erb** para renderizar la respuesta.

Show

```ruby
class PostsController < ApplicationController
  before_action :set_post, only: [:show, :edit, :update, :destroy]

  # GET /posts/1
  # GET /posts/1.json
  def show
  end

  private
    # Use callbacks to share common setup or constraints between actions.
    def set_post
      @post = Post.find(params[:id])
    end

    # Never trust parameters from the scary internet, only allow the white list through.
    def post_params
      params.require(:post).permit(:title, :content)
    end
end
```

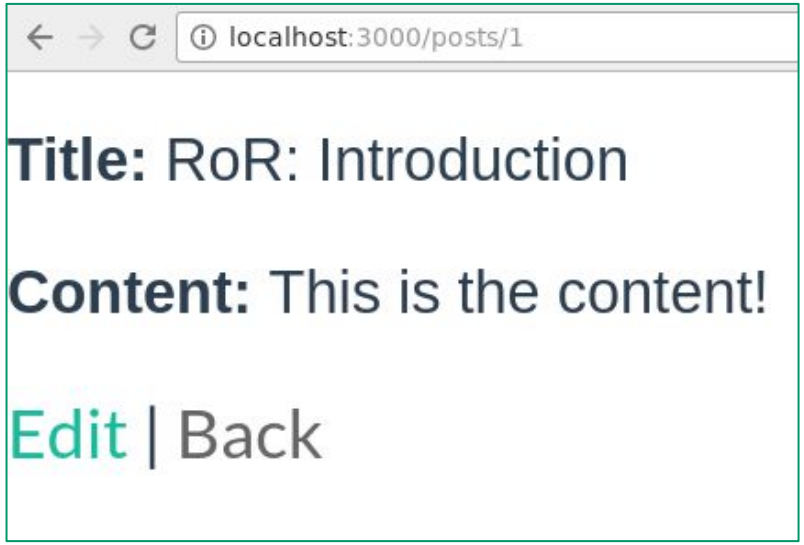
show.html.erb

```
<p id="notice"><%= notice %></p>

<p>
  <strong>Title:</strong>
  <%= @post.title %>
</p>

<p>
  <strong>Content:</strong>
  <%= @post.content %>
</p>

<%= link_to 'Edit', edit_post_path(@post) %> |
<%= link_to 'Back', posts_path %>
```



show.json.jbuilder

show.json.jbuilder

```
1  json.partial! "posts/post", post: @post
2
```


```
{
    id: 1,
    title: "RoR: Introduction",
    content: "This is the content!",
    created_at: "2016-11-22T11:05:13.001Z",
    updated_at: "2016-11-22T11:05:13.001Z",
    url: "http://localhost:3000/posts/1.json"
}
```

respond_to

- Es un helper de rails que **especifica cómo se va a responder un request** basado en el formato que está solicitando el request.
- Toma un **bloque adicional** donde el argumento es el formato (html, json, xml, etc.).
- El bloque especifica **cómo manejar** cada formato:
  - `format.format_name - matching template`
  - ```
    format.format_name {
    do_something_other_than_just_displaying_the_matching_template}
    ```
    ```

redirect_to

- En lugar de renderizar el template - envía una respuesta al navegador: “Ir a esta dirección”
- Usualmente lleva la URL absoluta como parámetro
- Podría ser una URL regular (como http://google.com) o una ruta nombrada (named route)
- Si el parámetro es un objeto - Rails intenta generar una url para ese objeto.

DESTROY

destroy

```ruby
class PostsController < ApplicationController
  before_action :set_post, only: [:show, :edit, :update, :destroy]


  # DELETE /posts/1
  # DELETE /posts/1.json
  def destroy
    @post.destroy
    respond_to do |format|
      format.html { redirect_to posts_url, notice: 'Post was successfully destroyed.' }
      format.json { head :no_content }
    end
  end

  private
    # Use callbacks to share common setup or constraints between actions.
    def set_post
      @post = Post.find(params[:id])
    end
```

Por qué redirect?

A pesar de que la redirección es un paso extra - a veces tiene sentido.

Ejemplos obvios:

- Cuando queremos que un cliente esté habilitado para marcar una cierta página o no se tiene un template específico para mostrar (destroy action) y entonces preferimos que se redireccione a una página genérica.

Entonces..

- El action Show obtiene un registro específico de la base de datos y lo despliega en un template HTML o JSON
- El action Destroy elimina el registro de la base de datos y redirige el navegador a otra página.

NEW

New

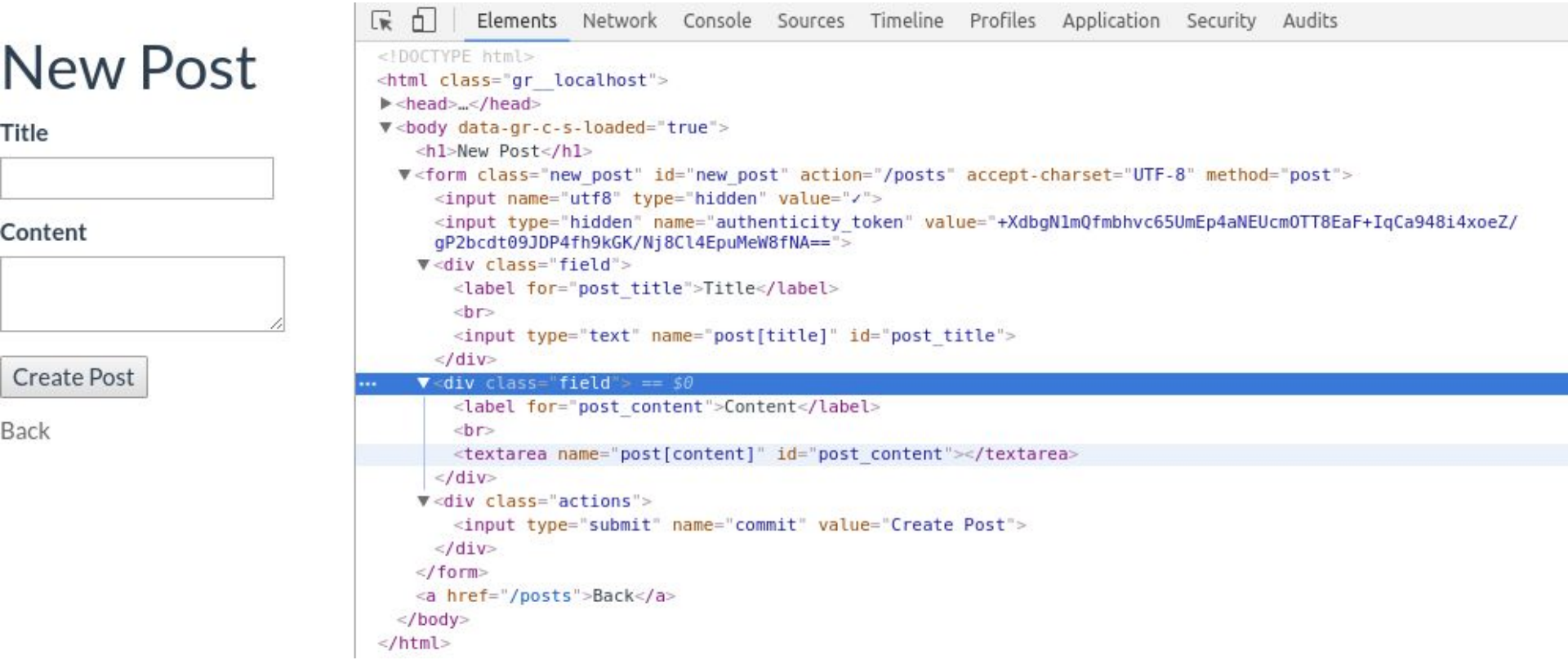
- Crea un nuevo objeto “post” vacío.
- Busca implícitamente por el template `new.html.erb`

```ruby
class PostsController < ApplicationController

  # GET /posts/new
  def new
    @post = Post.new
  end
```

new.html.erb

```
<h1>New Post</h1>

<%= render 'form' %>

<%= link_to 'Back', posts_path %>
```

Se explican los partials más adelante.


New Post
Title
Content
Create Post
Back
Elements
Network
Console
Sources
Timeline
Profiles
Application
Security
Audits
<!DOCTYPE html>
<html class="gr__localhost">
<head>…</head>
<body data-gr-c-s-loaded="true">
<h1>New Post</h1>
<form class="new_post" id="new_post" action="/posts" accept-charset="UTF-8" method="post">
<input name="utf8" type="hidden" value="✓">
<input type="hidden" name="authenticity_token" value="+XdbgN1mQfmbhvc65UmEp4aNEUcmOTT8EaF+IqCa948i4xoeZ/gP2bcdt09JDP4fh9kGK/Nj8Cl4EpuMeW8fNA==">

<label for="post_title">Title</label>

<input type="text" name="post[title]" id="post_title">

 == $0
<label for="post_content">Content</label>

<textarea name="post[content]" id="post_content"></textarea>

<input type="submit" name="commit" value="Create Post">

</form>
Back
</body>
</html>

CREATE

Create

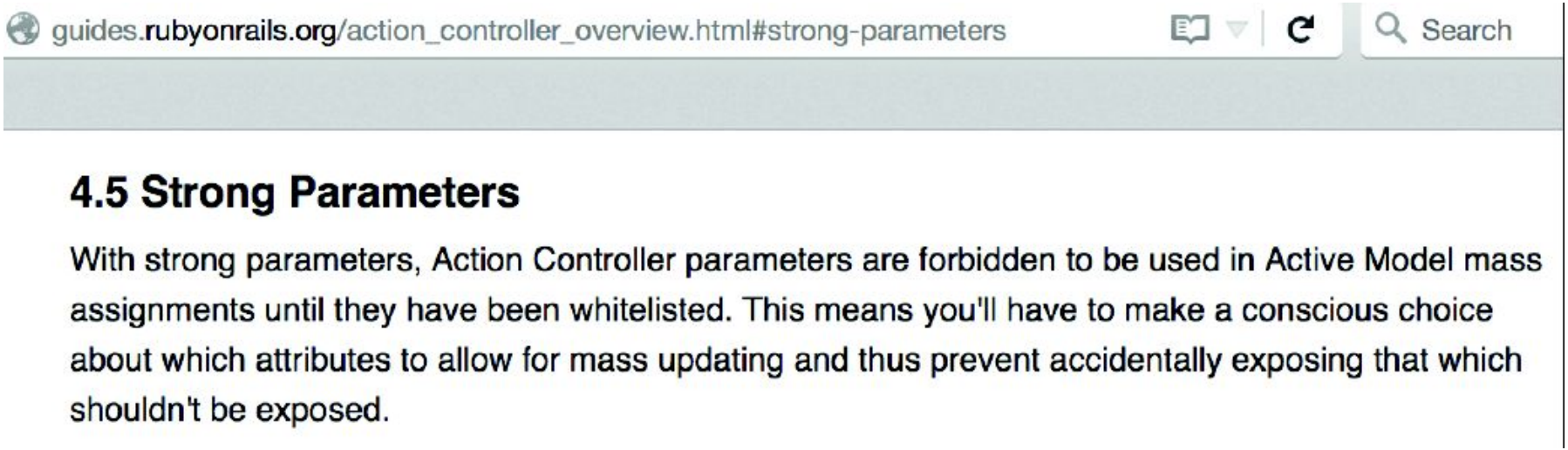
- **Crea un nuevo objeto** “post”con los parámetros que son pasados en el form **new**.
- Intenta **guardar** en la **base de datos**
- Si es **exitoso**, se redirecciona al template **show**.
- Si es **erróneo**, se renderiza de nuevo el template **new**
    - Por qué habría un error? No pasa las validaciones por ejemplo.

Entonces..

- El action new provee un formulario que debe ser llenado para crear un nuevo recurso.
- El action create acepta los parámetros pasados desde el formulario.

Strong Parameters

Strong parameters



Create action

```ruby
# POST /posts
# POST /posts.json
def create
  @post = Post.new(post_params)

  respond_to do |format|
    if @post.save
      format.html { redirect_to @post, notice: 'Post was successfully created.' }
      format.json { render :show, status: :created, location: @post }
    else
      format.html { render :new }
      format.json { render json: @post.errors, status: :unprocessable_entity }
    end
  end
end
  # Never trust parameters from the scary internet, only allow the white list through.
  def post_params
    params.require(:post).permit(:title, :content)
  end
```

Qué pasa cuando no está implementado?

ActiveModel::ForbiddenAttributesError in PostsController#update

ActiveModel::ForbiddenAttributesError

Extracted source (around line #**44**):

```
42    def update
43      respond_to do |format|
44        if @post.update(post_params)
45          format.html { redirect_to @post, notice: 'Post was successfully updated.' }
46          format.json { render :show, status: :ok, location: @post }
47        else
```

Rails.root: /home/vanessa/git/capacitaciones/ruby-on-rails-intro/modulo-5/sources/blog_web_app

Application Trace | Framework Trace | Full Trace

app/controllers/posts_controller.rb:44:in `block in update'
app/controllers/posts_controller.rb:43:in `update'

Flash

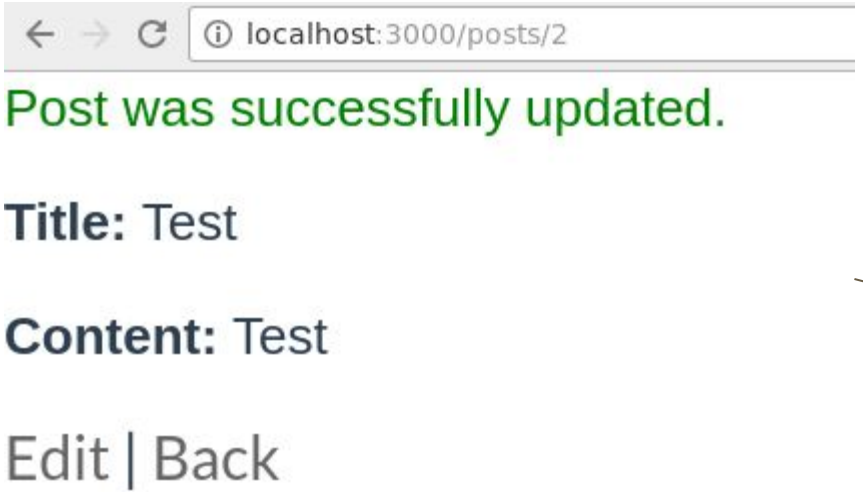
Flash

Problema: Queremos redireccionar a un usuario a una página diferente, pero al mismo tiempo darle un mensaje. Por ejemplo: “Se ha creado el post”.

Solución: flash - un hash donde los datos se persisten por exactamente UN request DESPUÉS del request actual.

Flash

- Se puede introducir el contenido en flash haciendo:
  - flash[:attribute] = value
- Dos atributos que son comúnmente incluidos:
  - :notice (good)
  - :alert (bad)
- Estos dos atributos son tan comunes, que redirect_to toma los parámetros :notice o :alert


localhost:3000/posts/2
Post was successfully updated.
Title: Test
Content: Test
Edit | Back

Notice!

Entonces

- Los parámetros son definidos en un whitelist. (strong parameters)
- Flash persiste por exactamente un request después del actual los datos incluidos en el hash.

EDIT

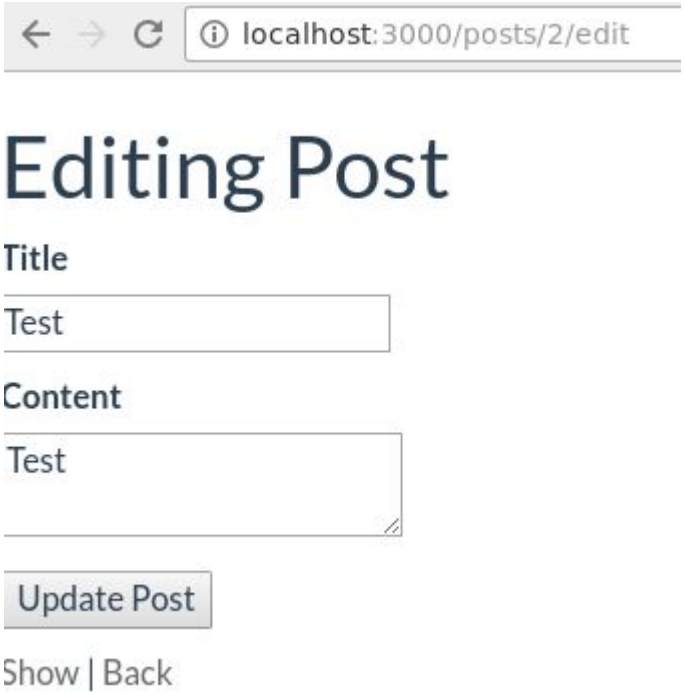
Edit

- Obtiene un objeto post basado en el id que viene como parámetro.
- Implícitamente busca el template edit.html.erb

Edit

```ruby
class PostsController < ApplicationController
  before_action :set_post, only: [:show, :edit, :update, :destroy]


  # GET /posts/1/edit
  def edit
  end

  private
  # Use callbacks to share common setup or constraints between actions.
  def set_post
    @post = Post.find(params[:id])
  end

  # Never trust parameters from the scary internet, only allow the white list through.
  def post_params
    params.require(:post).permit(:title, :content)
  end
end
```

```
<h1>Editing Post</h1>

<%= render 'form' %>

<%= link_to 'Show', @post %> |
<%= link_to 'Back', posts_path %>
```


localhost:3000/posts/2/edit
Editing Post
Title
Test
Content
Test
Update Post
Show | Back

UPDATE

Update

- Obtiene un post existente utilizando el parámetro id
- Actualiza el objeto “post” con los parametros que fueron enviados por el form.
- Intenta guardar o re-guardar el objeto a la base de datos.
- Si es exitoso, redirecciona al template show
- Si es erróneo, renderiza de nuevo el acction edit.

Update

```ruby
class PostsController < ApplicationController
  before_action :set_post, only: [:show, :edit, :update, :destroy]
  # PATCH/PUT /posts/1
  # PATCH/PUT /posts/1.json
  def update
    respond_to do |format|
      if @post.update(post_params)
        format.html { redirect_to @post, notice: 'Post was successfully updated.' }
        format.json { render :show, status: :ok, location: @post }
      else
        format.html { render :edit }
        format.json { render json: @post.errors, status: :unprocessable_entity }
      end
    end
  end


  private
  # Use callbacks to share common setup or constraints between actions.
  def set_post
    @post = Post.find(params[:id])
  end

  # Never trust parameters from the scary internet, only allow the white list through.
  def post_params
    params.require(:post).permit(:title, :content)
  end
end
```

Entonces

- edit/update es muy similar a new/create con la excepción de que existe un id de un recurso existente al que se le deben aplicar los cambios.
- Strong parameters se aplican tanto a la actualización como a la creación del recurso.

Partials

Partials

Rails comparte el principio DRY (Don’t repeat yourself)

Ya conocemos sobre el template **`application.html.erb`**, el cual nos habilita a **mantener el código del layout** de toda la aplicación en un solo lugar.

Pero sería bueno saber cómo reutilizar porciones de código de vistas que pudieran repetirse en varios templates.

Por ejemplo, los forms de **`edit`** y **`new`** - comparten código!

Partials

Los partials son **similares** a templates regulares, pero tienen capacidades más **refinadas**.

Son llamados con guión bajo **_**

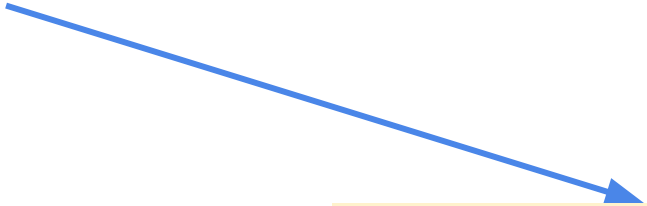
Se renderizan utilizando **render 'partialname'** (sin guión bajo)

render acepta un **segundo argumento**, un **hash** de variables locales que serán utilizadas en el partial.

Object Partial

El similar a pasar variables dentro del hash, es decir se puede renderizar un objeto específico.

`<%= render @post %>` va a renderizar un partial en **`app/views/posts/_post.html.erb`** y a asignar automáticamente una variable post local.



Convención!

Render collection of partials

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

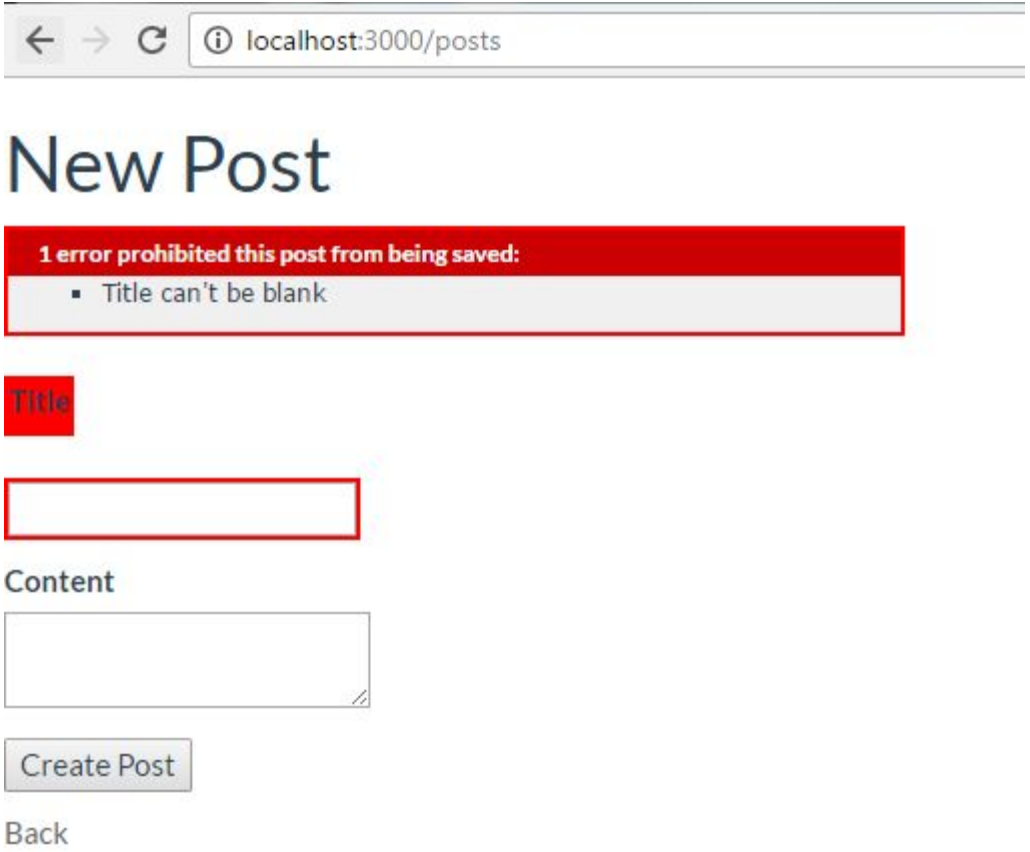
es equivalente a

```
<% @posts.each do |post| %>

<%= render post %>

<% end %>
```

_form.html.erb - Display errors

```
<%= form_for(@post) do |f| %>
  <% 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 |message| %>
        <li><%= message %></li>
      <% end %>
      </ul>
    </div>
  <% end %>

  <div class="field">
    <%= f.label :title %><br>
    <%= f.text_field :title %>
  </div>
  <div class="field">
    <%= f.label :content %><br>
    <%= f.text_area :content %>
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>
```

```
class Post < ActiveRecord::Base
    validates :title, presence: true

end
```


localhost:3000/posts
New Post
1 error prohibited this post from being saved:
Title can't be blank
Title
Content
Create Post
Back

Entonces

Partial es como una porción de template reutilizable que tiene un _ en el nombre y acepta parámetros cuando se va a renderizar.

Form Helpers and Layouts

Forms

```
<%= form_for(@post) do |f| %>
  <% 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 |message| %>
        <li><%= message %></li>
      <% end %>
      </ul>
    </div>
  <% end %>

  <div class="field">
    <%= f.label :title %><br>
    <%= f.text_field :title %>
  </div>
  <div class="field">
    <%= f.label :content %><br>
    <%= f.text_area :content %>
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>
```

Form con parámetros que matchean con los atributos del modelo

Form con parámetros que matchean con los atributos del modelo

Form Helpers

`form_for`

- Genera un tag form para el objeto recibido como parámetro
- A diferencia de un formulario HTML regular (que si no especificas utiliza GET), Rails utiliza POST por defecto.
- Esto tiene sentido:
  - En un post un password no es pasado como parte de la URL.
  - Cualquier cosa que termine modificando algo en el lado servidor debería ser un POST.

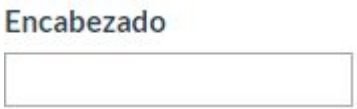
Form Helpers - f.label

`f.label`

Básicamente genera un tag HTML para el **`atributo`** provisto.

Para personalizar la descripción del label, se puede pasar un **`string`** como segundo parámetro

```
<div class="field">
  <%= f.label :title, "Encabezado" %><br>
  <%= f.text_field :title %>
</div>
```



Form Helpers - f.text_field

f.text_field

Genera un input type=text

Utiliza el parámetro :placeholder, como una entrada de hash, para especificar un texto por defecto que estará hasta que el usuario de clic al campo.

```
<div class="field">
  <%= f.label :title, "Heading" %><br>
  <%= f.text_field :title, placeholder: "Esto es un prueba" %>
</div>
```

f.text_area

f.text_area

Genera un text-area en lugar de un text field

Por defecto tiene un tamaño de 40 columnas por 20 filas.

Se puede especificar el tamaño con el atributo :size colsXrows

```
<div class="field">
  <%= f.label :content %><br>
  <%= f.text_area :content, size:"10x3" %>
</div>
```

Date helpers

`f.date_select`

- Conjunto de etiquetas (year, month, day) preseleccionados para acceder a un atributo en la base de datos.
- Existen muchas formas de formatear las fechas: f.time_select
- `f.datetime_select`
- `distance_of_time_in_words_to_now`

Ver ActionView::Helpers::DateHelper docs
http://api.rubyonrails.org/classes/ActionView/Helpers/DateHelper.html

Otros

- `search_field`
- `telephone_field`
- `url_field`
- `email_field`
- `number_field`
- `range_field`

Algunos de estos son dependientes del navegador

f.submit

- Genera un Submit button
- Acepta el nombre del button com primer argumento
- Si no se provee un nombre - genera un uno basado en el modelo y tipo de acción. Ej: “Create post”, “Update post”

http://guides.rubyonrails.org/form_helpers.html

Layouts

El layout application.html.erb es aplicado por defecto en cualquier view.

El Layout que matchee el nombre del controller será aplicado en caso de que esté presente (sobreescribiendo application.html.erb)

También se puede utilizar el método **`layout`** dentro del controller (fuera de cualquier action) para setear un layout para todo el controller.

```
layout ‘some_layout’
```

Layouts / Rendering

Se puede incluir el layout para un action específico utilizando:

```
render layout: ‘my_layout’
```

Si no se desea un determinado layout - (y fue definido en el controller entero) se puede setear a false.

```
render layout: false
```

Form Helpers

Son una manera rápida de generar formularios

Los layouts nos permiten mostrar una parte “común” del template, para toda la aplicación, para un determinado controller o para un action en particular.