Tutorial de introducción a Ruby on Rails

Ruby on Rails es un entorno de desarrollo web de código abierto que está optimizado para satisfacción de los programadores y de la productividad. Te permite escribir un buen código favoreciendo la convención antes que la configuración.

En este tutorial introductorio aprenderemos qué es Ruby, qué es Rails y cómo crear una aplicación de ejemplo.

¿Qué es Ruby on Rails?

Ruby on Rails es un entorno de desarrollo web, de código abierto (open source, software libre), que nos permite construir aplicaciones web flexibles y robustas rápidamente.

Las dos piezas principales de este entorno son dos: por un lado Ruby, y por otro, Rails.

¿Qué es Ruby?

Ruby es un lenguaje de programación enfocado a la simplicidad y a la productividad, con una sintaxis elegante y natural, que le hace muy fácil de entender. Es un lenguaje de script (no compilado), totalmente orientado a objetos.

El lenguaje Ruby fué creado por Yukihiro matz Matsumoto en 1995 cogiendo lo que más le gustaba de otros lenguajes como Perl, Smalltalk, Eiffel, Ada y Lisp.

Mientras que otros lenguajes tienen tipos primitivos (como números, booleanos, ...) que no son objetos, aquí todo es un objeto y por tanto se le pueden asociar propiedades y métodos, así como redefinir o extender su comportamiento.

Veamos algunos ejemplos:

Ejemplo 1

```
5.times { print "Me encanta Ruby!" }
```

En este primer ejemplo vemos cómo el número 5 tiene un método "times" con el que tenemos un bucle, al que le pasamos el bloque entre llaves para que sea ejecutado 5 veces.

Ejemplo 2

```
(1..100).each do |i| print i end
```

En este segundo ejemplo, vemos cómo los rangos de números (1..100) tienen un método "each" con el que podemos también hacer un bucle, cogiendo cada valor en cada vuelta en el parámetro i.

Ejemplo 3

```
class Numeric def mas(x) self.+(x) end def par? ((self/2)*2) == self end def impar? !par? end end n = 5.mas 6 # n ahora vale 11 n.par? # devolverá false n.impar? # devolverá true
```

Aquí vemos la posibilidad de redefinir el lenguaje. Aunque la manera normal de sumar dos números es mediante el operador "+", hemos añadido un método "mas" a la clase Numeric de donde descienden todos los números para poder sumarlos con esta otra sintaxis. Como vemos, en realidad el operador "+" es azucar sintáctico para el método ".+" de la clase Numeric.

Pero más importante que redefinirlo, sin duda es poder extenderlo: vemos en el anterior ejemplo cómo podemos ampliar la clase Numeric con métodos que nos indiquen si el número es par o impar.

Ruby es un lenguaje completo, maduro y estable con muchísimas más características como manejo de excepciones, recolector de basura, interacción con librerías externas, multithreading independientemente del sistema operativo, y es altamente portable (funciona en GNU/Linux, UNIX, Mac OS X, Windows 95/98/Me/NT/2000/XP/Vista, DOS, BeOS, OS/2).

Para profundizar en el lenguaje Ruby, visita la web oficial de Ruby y echa un vistazo a la presentación que hizo Sergio Gil en la Conferencia Rails 2006, titulada Ruby Mola (y por qué).

¿Qué es Rails?

Rails es un framework [?] creado para el desarrollo de aplicaciones web. Para entendernos, es una serie de utilidades, de herramientas para crear aplicaciones web más rápidamente que haciéndolo desde cero. Usando un framework evitamos reinventar la rueda, nos ahorramos muchísimo trabajo y podemos concentrarnos en lo que verdaderamente importa: la lógica de la aplicación, no en escribir una y otra vez los mismos formularios y scripts de proceso para crear, leer, modificar y eliminar datos en nuestra base de datos.

Frameworks para desarrollo web hay muchos: Cake para PHP, Django para Python, Spring para Java, y muchos otros. Rails es un framework que usa el lenguaje Ruby, y por eso se conoce al conjunto como Ruby on Rails. Pero también existen otros frameworks para Ruby que no son Rails, como puede ser el microframework Camping.

Rails fué desarrollado por el danés David Heinemeier, como una herramienta para facilitarle el trabajo al programar la aplicación web Basecamp para la empresa 37 Signals. En julio de 2004 liberó el código como software libre, y en febrero de 2005 comenzó a aceptar colaboraciones para mejorar Rails, formándose un amplio equipo de programadores, el *core team*.

David Heinemeier y el Rails Core Team

Entre las características de Ruby on Rails podemos destacar que está basado en el patrón de diseño MVC (Modelo-Vista-Controlador [?]) con el que se separa la lógica de la presentación, que nos permite dejar de pensar siempre en SQL para pasar a pensar en objetos, que dispone de utilidades para generar rápidamente interfaces de administración, es independiente de la base de datos, podemos realizar tests continuos para garantizar que nuestra aplicación funciona como esperamos, se puede extender mediante el uso de plugins, se integra muy fácilmente con librerías de efectos como script.aculo.us... y todo ello escribiendo muy pocas líneas de código, muy claras y fáciles de entender.

Vamos a ver todo esto mediante una serie de ejemplos prácticos, para no perdernos demasiado en la teoría.

Crear nuestra aplicación

El primer paso para crear una nueva aplicación Rails es crear la estructura base. Para ello abriremos un terminal de comandos y escribiremos:

```
$ rails blog
```

Esto generará un directorio llamado "blog", que es el nombre de nuestra aplicación, y dentro un montón de subdirectorios y ficheros que forman la base de nuestra aplicación. Entra en este nuevo directorio y veamos los principales elementos:

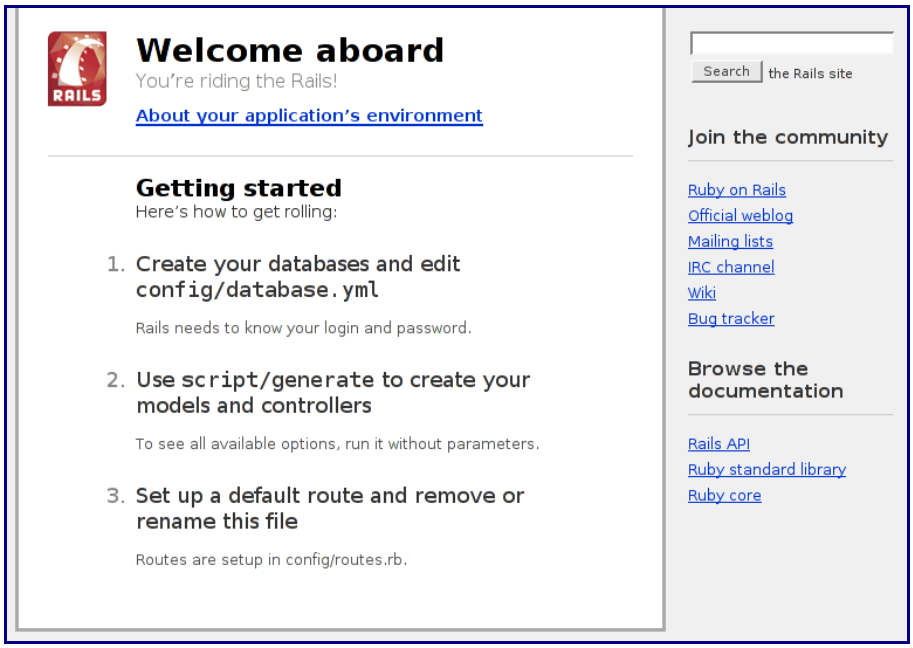
```
$ cd blog $ ls app config doc log Rakefile script tmp components db lib public README test vendor
```

Dentro de **app** encontrarás los ficheros correspondientes a los modelos, vistas, controladores y helpers, que forman la parte principal de nuestra aplicación (de ahí el nombre de "app", que viene de "application"). En **config** están los ficheros de configuración que definen el acceso a base de datos y otros, en **log** encontraremos los ficheros de registro de servidor (logs), la carpeta **public** contendrá elementos estáticos como imágenes, hojas de estilo y javascript...

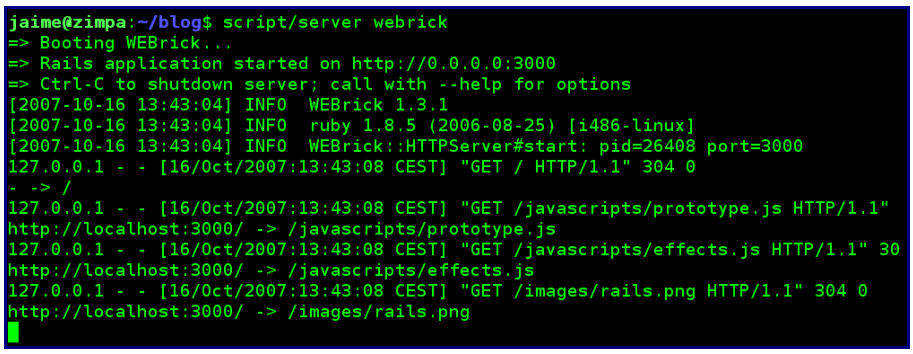
Fíjate que también existe una carpeta **script**. Contiene unas cuantas herramientas de Rails que nos ayudarán en nuestro desarrollo. Vamos a probar la primera:

```
$ script/server
```

A continuación, abre tu navegador web y dirígete a la dirección http://localhost:3000. Lo que verás es la página de bienvenida de Ruby on Rails dentro de tu nueva aplicación, y esto es porque Rails viene con su propio servidor web de serie, llamado WEBrick. Si vuelves al terminal podrás observar cómo WEBrick ha ido respondiendo a las peticiones del navegador web sirviendo los ficheros solicitados.



La página de bienvenida a nuestra aplicación Rails



WEBrick en funcionamiento

Mantén este terminal abierto con el servidor web en funcionamiento, y abre un nuevo terminal para poder seguir usando las herramientas de Rails.

Nuestra aplicación accederá a una base de datos, por lo que antes de continuar debemos crearla. Emplearemos MySQL... crea una base de datos llamada "blog_development" con un usuario "blogusr" y clave "blogpwd" usando tu herramienta favorita (phpMyAdmin, CocoaMySql...). Nosotros lo haremos directamente desde línea de comandos:

```
$ mysql -uroot -p (...) create database blog_development default character set utf8 collate utf8_unicode_ci; grant all on blog_development.* to 'blogusr'@'localhost' identified by 'blogpwd' ;
```

Una vez exista esta base de datos, abriremos el fichero **config/database.yml** y apuntaremos los datos en el apartado **development**:

 # config/database.yml

```
development: adapter: mysql database: blog_development username: blogusr password: blogpwd socket: /var/run/mysqld/mysqld.sock
```

Fíjate que en este fichero tenemos tres secciones principales, que corresponden a los tres entornos que se manejan en una aplicación Rails: development (desarrollo), test (pruebas) y production (producción). Cada entorno tendrá su propia base de datos independiente, de manera que podamos desarrollar tranquilos sin cambiar nada de la base de datos de producción, y podamos testear en una base de datos que se regenerará cada vez que lancemos la batería de tests. Pero por ahora usaremos sólo la base de desarrollo.

Crear artículos

Queremos crear un blog, así que lo primero que necesitamos es poder escribir nuestros artículos. Para nosotros, un artículo tendrá un título y un texto principal, así que comenzaremos con esto. Lo primero que necesitamos es generar el modelo Articulo, para lo que usaremos el script de generación de modelos:

```
$ script/generate model Articulo exists app/models/ exists test/unit/ exists test/fixtures/ create app/models/articulo.rb create test/unit/articulo_test.rb create test/fixtures/articulos.yml create db/migrate create db/migrate/001_create_articulos.rb
```

Vemos que al lanzarlo, el script comprueba que existan los subdirectorios necesarios, y crea varios ficheros: en orden, **articulo.rb** contendrá la lógica del modelo; los archivos dentro de **test** nos facilitan el testeo de esta lógica, y el fichero creado dentro de **db/migrate** nos facilita especificar la estructura de la tabla que almacenará los artículos en la base de datos. Vamos a abrir y rellenar este fichero:

 # db/migrate/001_create_articulos.rb

```
class CreateArticulos < ActiveRecord::Migration def self.up create_table :articulos do |t| t.column :titulo, :string t.column :cuerpo, :text end end def self.down drop_table :articulos end end
```

Vemos que en este fichero define dos métodos, uno llamado **self.up** que realizará los cambios necesarios, y otro llamado **self.down** que deshace estos mismos cambios en caso de que quisiéramos echarnos atrás.

Las acciones que realizamos dentro de self.up son muy sencillas de leer: si lo leemos en inglés casi nos sale la frase "*crea una tabla articulos y haz una columna título que es una cadena y una columna cuerpo que es un texto*". Por otro lado, self.down dice simplemente "*tira la tabla articulos*".

Además de las columnas que definamos aquí, Rails incluirá automáticamente para cada tabla una columna "id" para el identificador único de cada fila de la tabla.

Nos podemos fijar también en que nosotros pedimos definir el modelo Articulo (en singular), y Rails siguiendo sus convenciones, supone que nos pondremos de acuerdo en llamar a la tabla "articulos", como el plural del modelo. Rails es lo suficientemente listo como para saber hacer el plural incluso en casos irregulares como Person -> people. También podemos definir nuestras propias reglas de pluralización para definir las reglas gramaticales de otros idiomas y hacer que por ejemplo pluralice correctamente un modelo Camion como camiones, en lugar de "camions".

Para aplicar esta definición del modelo Articulo en la base de datos, ejecutaremos:

```
$ rake db:migrate (in /home/jaime/blog) == CreateArticulos: migrating ============================================= -- create_table(:articulos) -> 0.0990s == CreateArticulos: migrated (0.0993s) ========================================
```

Al ejecutar esta orden, Rails leerá el fichero config/database.yml para saber cómo acceder a la base de datos, y aplicará las órdenes definidas en la migración para crear las tablas en MySQL. Si en el fichero de configuración hubiéramos especificado otro tipo de base de datos (PostgreSQL, Oracle, Sqlite, SQL Server...) en lugar de MySQL, Rails habría sabido cómo crear las tablas; él se ocupa de la sintaxis SQL de cada tipo de base de datos y nosotros podemos definirlo a un nivel más alto de abstracción.

Ya tenemos nuestro modelo Articulo creado y la tabla que almacenará sus datos; ahora necesitamos un controlador para administrar los artículos, que nos permita crearlos, modificarlos, borrarlos...

Para ello ejecutaremos:

```
$ script/generate controller Admin exists app/controllers/ exists
app/helpers/ create app/views/admin exists test/functional/ create
app/controllers/admin_controller.rb                          create
test/functional/admin_controller_test.rb                     create
app/helpers/admin_helper.rb
```

Acabamos de generar nuestro primer controlador, al que hemos bautizado como "admin". Por ahora está vacío, no hace nada, como podemos comprobar visitando http://localhost:3000/admin: nos indicará que ninguna acción ha respondido al método index.

Para comenzar rápidamente, vamos a hacer uso de una característica de Rails llamada **scaffold**, que se traduce por "andamio", y es precisamente eso: al igual que al construir edificios uno se apoya en andamios al comienzo de la construcción, y posteriormente termina quitándolos cuando la construcción está avanzada y ya no los necesitamos. Un scaffold nos genera listados, formularios y lógica de proceso para las operaciones más comunes en una interfaz de administración: crear, visualizar, modificar y eliminar.

Edita el fichero **app/controllers/admin_controller.rb** y ponlo como en el siguiente ejemplo:

 # app/controllers/admin_controller.rb

```
class AdminController < ApplicationController scaffold :articulo
end
```

A continuación, reinicia WEBrick (para la mayoría de los cambios no es necesario reiniciarlo, pero como hemos modificado el fichero de configuración de base de datos, lo necesitaremos hacer), y dirígete de nuevo a http://localhost:3000/admin... prueba el interfaz y crea unos cuantos artículos de prueba, modifícalos, elimina, etc... Si creas más de 10 podrás ver cómo aparecen automáticamente los enlaces de paginación. Todo esto te lo acabas de ahorrar, no necesitas programarlo ya que el scaffold de Rails lo ha generado por tí.

Scaffold de administración de artículos

Después de trastear un rato con este interfaz de administración, caemos en la cuenta de que nos gustaría que el modelo Articulo tuviera más campos. Nos gustaría poder almacenar la fecha y hora en que se creó el artículo, así como la fecha y hora de la última modificación. Para ello, lo que necesitamos es una nueva migración donde definamos los cambios en la base de datos:

```
$ script/generate migration fechas_en_articulos exists db/migrate create db/migrate/002_fechas_en_articulos.rb
```

Editamos el archivo generado y lo dejamos así:

 # db/migrate/002_fechas_en_articulos.rb

```
class FechasEnArticulos < ActiveRecord::Migration def self.up add_column :articulos, :created_at, :datetime add_column :articulos, :updated_at, :datetime end def self.down remove_column :articulos, :created_at remove_column :articulos, :updated_at end end
```

Este código se lee fácilmente: añadir a la tabla "articulos" dos columnas "created_at" y "updated_at" de tipo fecha y hora; y si nos echamos atrás, eliminar esas columnas. ¿Por qué le hemos puesto nombres en inglés a estos campos de fecha? Pues para aprovecharnos de que si lo hacemos así, Rails se encargará automáticamente de rellenar estos campos de fecha sin que tengamos que programarlo, cada vez que se cree o actualice un artículo.

Para aplicar estos cambios ejecutaremos:

```
$ rake db:migrate (in /home/jaime/blog) == FechasEnArticulos: migrating ============================================= -- add_column(:articulos, :created_at, :datetime) -> 0.0453s -- add_column(:articulos, :updated_at, :datetime) -> 0.0066s == FechasEnArticulos: migrated (0.0522s) ======================================
```

Si ahora volvemos al interfaz de administración, veremos que estos dos nuevos campos se han incorporado a los listados y formularios, y funcionan como esperábamos. Evidentemente, los artículos creados con anterioridad no tendrán almacenadas las fechas puesto que estos campos no existían cuando los creamos.

Los scaffolds dinámicos son de mucha utilidad para arrancar rápidamente e ir viendo cómo queda la aplicación mientras vamos pensando qué campos necesitamos, pero llega un momento en que es mejor disponer de los códigos en ficheros de manera que los podamos modificar a nuestro antojo para personalizar el aspecto (por ejemplo traducir las frases en inglés!). Para ello ejecutamos:

```
$ script/generate scaffold Articulo Admin exists app/controllers/ exists app/helpers/ exists app/views/admin exists app/views/layouts/ exists test/functional/ dependency model exists app/models/ exists test/unit/ exists test/fixtures/ identical app/models/articulo.rb identical test/unit/articulo_test.rb identical test/fixtures/articulos.yml create app/views/admin/_form.rhtml create app/views/admin/list.rhtml create app/views/admin/show.rhtml create app/views/admin/new.rhtml create app/views/admin/edit.rhtml overwrite app/controllers/admin_controller.rb? [Ynaqd] y force app/controllers/admin_controller.rb overwrite test/functional/admin_controller_test.rb? [Ynaqd] y force test/functional/admin_controller_test.rb identical app/helpers/admin_helper.rb create app/views/layouts/admin.rhtml create public/stylesheets/scaffold.css
```

Vemos que el script nos solicita nuestro permiso para sobreescribir un par de ficheros existentes, que se crearon al generar el controlador. Contestaremos afirmativamente en ambos casos con "y".

Examinemos rápidamente los códigos que nos ha generado este scaffold. Abre de nuevo el fichero **app/controllers/admin_controller.rb**:

 # app/controllers/admin_controller.rb

```
class AdminController < ApplicationController def index list render :action => 'list' end # GETs should be safe (see http://www.w3.org/2001/tag/doc/whenToUseGet.html) verify :method => :post, :only => [ :destroy, :create, :update ], :redirect_to => { :action => :list } def list @articulo_pages, @articulos = paginate :articulos, :per_page => 10 end def show @articulo = Articulo.find(params[:id]) end def new @articulo = Articulo.new end def create @articulo = Articulo.new(params[:articulo]) if @articulo.save flash[:notice] = 'Articulo was successfully created.' redirect_to :action => 'list' else render :action => 'new' end end def edit @articulo = Articulo.find(params[:id]) end def update @articulo = Articulo.find(params[:id]) if @articulo.update_attributes(params[:articulo]) flash[:notice] = 'Articulo was successfully updated.' redirect_to :action => 'show', :id => @articulo else render :action => 'edit' end end def destroy Articulo.find(params[:id]).destroy redirect_to :action => 'list' end end
```

Viéndolo en orden, vemos que define todos los métodos que necesitamos para administrar los artículos. No intentes comprender ahora toda la sintaxis empleada, quédate con la idea general:

- **index** corresponde a la portada principal, que hace lo mismo que list y pinta ("render") lo mismo que esa acción.
- **list** genera un listado paginado con los artículos.
- **show** busca el artículo cuyo id se haya indicado por parámetro y lo almacena en la variable @articulo.
- **new** prepara un nuevo Articulo, vacío.
- **create** crea un nuevo Articulo con los parámetros que le hayan pasado y lo salva en la base. Si tiene éxito, prepara un mensaje de "OK" y redirige al listado; si no, redirige de vuelta a la acción "new".
- **update** busca el artículo por el id indicado, y se intenta actualizar con los parámetros pasados. Si tiene éxito, redirige a "show", y si no, de vuelta a "edit".
- **destroy** busca el artículo por id, lo elimina y redirige al listado.

A cada acción del controlador le corresponde una vista. Encontrarás los ficheros correspondientes en app/views/admin. Si los abres, verás que son simples ficheros HTML, con partes de código Ruby embebido, usando las etiquetas **<%** y **%>**, similar a otros lenguajes de script para web.

- **list.rhtml** muestra el listado de artículos.
- **show.rhtml** muestra una ficha de artículo.
- **new.rhtml** muestra la página para crear un artículo.
- **edit.rhtml** muestra la página para editar un artículo.
- **_form.rhtml** contiene el formulario de artículo, compartido por las acciones new y edit. Esto es lo que se conoce como "partial", ya que contiene un código parcial, que se inserta dentro de otro para completarse.

Podemos modificar todos estos ficheros a nuestro antojo. Por ejemplo, prueba a cambiar **list.rhtml** por lo siguiente para tener un listado algo más sencillo:

 # app/views/admin/list.rhtml

```
<h1>Los artículos de mi blog</h1> <% for articulo in
@articulos.reverse %> <h2><%= link_to articulo.titulo, :action
=> :show, :id => articulo %></h2> <p><%=
simple_format(h(articulo.cuerpo)) %></p> <p><%= link_to 'Borrar',
{ :action => 'destroy', :id => articulo }, :confirm => '¿Seguro?',
:method => :post %></p> <hr /> <% end %> <%= link_to 'Anterior', {
:page => @articulo_pages.current.previous } if
@articulo_pages.current.previous %> <%= link_to 'Siguiente',
{ :page => @articulo_pages.current.next } if
@articulo_pages.current.next %> <br /> <%= link_to
'Nuevo', :action => 'new' %>
```

Parte pública del blog

Ya tenemos un interfaz web que nos permite administrar los artículos. Esta zona será privada (posteriormente podríamos ponerle protección por usuario y contraseña); necesitamos una parte pública que muestre los artículos.

Vamos a crear para ello un nuevo controlador, que tendrá por el momento sólo 2 acciones: "index" (la página principal que mostrará los artículos) y "detalles" (la página que mostrará todos los detalles de un artículo):

```
$ script/generate controller Blog index detalles exists app/controllers/ exists app/helpers/ create app/views/blog exists test/functional/ create app/controllers/blog_controller.rb create test/functional/blog_controller_test.rb create app/helpers/blog_helper.rb create app/views/blog/index.rhtml create app/views/blog/detalles.rhtml
```

Ahora abriremos el controlador, **app/controllers/blog_controller.rb**, y definiremos las dos acciones:

 # app/controllers/blog_controller.rb

```
class BlogController < ApplicationController def index @articulos = Articulo.find(:all, :order => 'id DESC') end def detalles @articulo = Articulo.find(params[:id]) end end
```

Como vemos, las acciones son muy sencillas: en index, creamos el objeto **@articulos** que contendrá todos (*:all*) los artículos, ordenados a la inversa (*:order => 'id DESC'*) para mostrar los últimos arriba. La acción "detalles" creará el objeto @articulo buscándolo por el id pasado como parámetro.

Ahora necesitamos crear las vistas, que cogerán estos objetos que hemos preparado en el controlador y mostrarán su información por pantalla. Comencemos por la vista correspondiente a la acción "index", editando el fichero **app/views/blog/index.rhtml**. Como verás es un simple bucle que recorre @articulos, y para cada uno muestra su título, cuerpo y una línea de separación al final. El título se muestra en forma de enlace a la acción "detalles", con el id del artículo en curso dentro del bucle. El cuerpo se muestra usando tres "helpers" de texto: "h" se encarga de asegurar que el html mostrado es seguro (para evitar buena parte de ataques); "truncate" se encarga de recortar el texto en caso de que sea más largo que la longitud especificada, y "simple_format" se encarga de dar un formato muy simple a un texto para mostrarlo como HTML, por ejemplo convirtiendo los saltos de línea en párrafos.

 # app/views/blog/index.rhtml

```
<% for articulo in @articulos %> <h1><%= link_to articulo.titulo, :action => :detalles, :id => articulo %></h1> <%= simple_format(truncate(h(articulo.cuerpo),200)) %> <hr /> <% end %>
```

El listado de artículos

A continuación, edita **app/views/blog/detalles.rhtml** para crear la ficha de artículo. El código en esta ocasión muestra el título del artículo sin ser enlace, el cuerpo sin cortar, y además abajo la fecha en el formato dia-mes-año [hora:minuto].

```
<h1><%= @articulo.titulo %></h1> <%= simple_format(h(@articulo.cuerpo)) %> <p><em><%= @articulo.created_at.strftime("%d-%m-%Y [%H:%M]") %></em></p>
```

Detalle de artículo

Comentarios

Para finalizar el blog, necesitamos poder recoger y mostrar los comentarios que los visitantes quieran dejar en cada artículo. Es hora de crear un nuevo modelo: Comentario. Un comentario tendrá un texto, el nombre del usuario, la fecha de creación, y el id del artículo al que está asociado.

```
$ script/generate model Comentario exists app/models/ exists test/unit/ exists test/fixtures/ create app/models/comentario.rb create test/unit/comentario_test.rb create test/fixtures/comentarios.yml exists db/migrate create db/migrate/003_create_comentarios.rb
```

Editamos el fichero de migración:

 # db/migrate/003_create_comentarios.rb

```
class CreateComentarios < ActiveRecord::Migration def self.up create_table :comentarios do |t| t.column :texto, :text t.column :nombre, :string t.column :created_at, :datetime t.column :articulo_id, :integer end end def self.down drop_table :comentarios end end
```

Ejecutamos la migración para crear la nueva tabla de comentarios...

```
$ rake db:migrate (in /home/jaime/blog) == CreateComentarios: migrating ============================================= -- create_table(:comentarios) -> 0.1017s == CreateComentarios: migrated (0.1019s) ======================================
```

Hemos creado la nueva tabla de comentarios, que incluye un campo para relacionarlos con el artículo que le corresponda. Ahora hemos de indicar qué tipo de relación queremos entre artículos y comentarios, editando los modelos. Queremos que un artículo pueda tener varios comentarios, y que un comentario pertenezca a un único artículo, así que esto se traduce en lo siguiente:

 # app/models/articulo.rb

```
class Articulo < ActiveRecord::Base has_many :comentarios end
```

 # app/models/comentario.rb

```
class Comentario < ActiveRecord::Base belongs_to :articulo end
```

A continuación necesitamos un formulario para permitir a los usuarios crear comentarios para un artículo. Editaremos la vista de detalle de artículo:

 # app/views/blog/detalles.rhtml

```
<h1><%= @articulo.titulo %></h1> <%= simple_format(h(@articulo.cuerpo)) %> <p><em><%= @articulo.created_at.strftime("%d-%m-%Y [%H:%M]") %></em></p> <h2>Comentarios</h2> <% for comentario in @articulo.comentarios %> <%= simple_format(h(comentario.texto)) %> <p><em><%= comentario.nombre %></em>, <%= comentario.created_at.strftime("%d-%m-%Y [%H:%M]") %></p> <% end %> <h3>Añade tu comentario a este artículo!</h3> <% form_tag :action => 'comenta', :id => @articulo do %> <p><label for="comentario_nombre">Nombre</label><br/> <%= text_field 'comentario', 'nombre' %></p> <p><label for="comentario_texto">Mensaje</label><br/> <%= text_area 'comentario', 'texto', {:rows => 5, :cols => 80} %></p> <%= submit_tag 'Enviar comentario' %> <% end %>
```

En primer lugar, hemos creado una sección que muestra los comentarios del artículo. Fíjate en que encontrar los comentarios de un artículo es tan sencillo como usar **@articulo.comentarios**, y realizar un bucle sobre ellos.

En segundo lugar, mostramos el formulario de creación de comentarios. La acción del formulario apunta a la acción "comenta", con el id del artículo actual. Dentro del formulario incluimos un área de texto para el nombre del usuario y un área de texto para el mensaje.

Sólo nos falta definir la acción "comenta". Editaremos el controlador del blog y la añadimos al final:

 # app/controllers/blog_controller.rb

```
class BlogController < ApplicationController def index @articulos = Articulo.find(:all, :order => 'id DESC') end def detalles @articulo = Articulo.find(params[:id]) end def comenta Articulo.find(params[:id]).comentarios.create(params[:comentario]) redirect_to :action => :detalles, :id => params[:id] end end
```

La nueva acción "comenta" busca el artículo con el id indicado por parámetro, y en sus comentarios crea uno con los parámetros del comentario (nombre y texto).

Suficiente por hoy...

Nuestro blog está listo! Puedes probar a crear artículos desde "/admin" y a verlos en la parte pública "/blog", dejando comentarios y viendo cómo aparecen para cada uno.

Si quieres puedes descargar el código del blog que hemos creado para tenerlo de referencia.

En siguientes entregas del tutorial de introducción aprenderemos a usar tests y validaciones, usar layouts, AJAX, RSS y varios plugins.

Hasta la próxima!