

**NO LEAS ESTE FICHERO EN GITHUB, LAS
GUIAS ESTÁN PUBLICADAS EN**

<http://guiasrails.es> .

Comenzando con Rails

Esta guía cubre el comienzo y la ejecución de
Ruby on Rails.

Después de leer esta guía, tu conocerás:

**Como instalar Rails, crear una nueva
aplicación Rails, y conectar tu
aplicación a una base de datos.**

**La disposición general de una
aplicación Rails.**

**Los principios básicos del patrón
MVC (Modelo, Vista, Controlador) y el
diseño RESTful.**

**Como rapidamente generar los
componentes iniciales de una
aplicación Rails.**

Chapters



1. [Supuestos de la Guía](#)

2. [¿Porqué Rails?](#)

3. [Creando un nuevo Proyecto Rails](#)

[Instalando Rails](#)

[Creando la Aplicación Blog](#)

4. [¡Hola, Rails!](#)

[Levantando el Servidor Web](#)

[Decir "Hola", Rails](#)

[Configurando la página de Inicio de la aplicación](#)

5. [Empezar y Ejecutar](#)

[Estableciendo la base del trabajo](#)

[El primer formulario](#)

[Creando artículos](#)

[Creando el Modelo Artículo](#)

[Ejecutando una Migración](#)

[Grabando los datos en el controlador](#)

[Mostrando Artículos](#)

[Listando todos los artículos](#)

[Añadiendo enlaces](#)

[Añadiendo algunas validaciones](#)

[Actualizando Artículos](#)

[Utilizando partials para eliminar la duplicidad en las vistas](#)

[Borrando Artículos](#)

6. **Añadiendo un Segundo Modelo**

[Generando el Modelo](#)

[Modelos Asociados](#)

[Añadiendo una Ruta para los Comentarios](#)

[Generando un Controlador](#)

7. **Refactorizando**

[Mostrando Colecciones en Vistas Parciales](#)

[Desplegando el Formulario Parcial](#)

8. **Borrando Comentarios**

[Borrando Objetos Asociados](#)

9. **Seguridad**

[Autenticación Básica](#)

[Otra Consideración de Seguridad](#)

10. **¿Que es lo que sigue?**

11. **Configuración de Gotchas**

1 Supuestos de la Guía

Esta guía está diseñada para principiantes quienes quieren comenzar con Rails garabateando con el código. Esto no supone que tu tienes una experiencia previa con Rails. Sin embargo, para sacar el mejor partido del aprendizaje, tendrás que contar con ciertos prerequisites de instalación:

El lenguaje **Ruby** versión 1.9.3 o superior.

El sistema de paquetes **RubyGems**, el cual se instala con Ruby versiones 1.9 o posteriores.

Para aprender más acerca de RubyGems, por favor leer **Guías de RubyGems**.

Una instalación funcionando de **SQLite3 Database**.

Rails es un framework para aplicaciones web, que corre sobre el lenguaje de programación Ruby. Si no tienes experiencia previa con Ruby, podrás encontrar muy empinada la curva de aprendizaje. Aquí hay un listado de muchos sitios donde puedes aprender en línea Ruby:

La página oficial del Lenguaje de programación Ruby.
reSRC's un listado de libros de programación gratuitos

Tenga en cuenta que algunos recursos, mientras siguen siendo excelentes, cubren versiones de Ruby tan antiguas como la 1.6, y comunmente la 1.8, y podrían no incluir alguna de la sintaxis que verás en tu día a día de desarrollo en Rails.

2 ¿Porqué Rails?

Rails es un framework de desarrollo de aplicaciones web escrito en el lenguaje de Ruby. Está diseñado para simplificar la programación de aplicaciones web, asumiendo que todos los desarrolladores necesitan empezar. Este diseño hace que se tenga que escribir menos código que en otros frameworks.

Experimentados desarrolladores Rails dicen que el desarrollo de la aplicaciones web se vuelve mucho más divertido.

Rails es un software obstinado. Esto hace suponer que existe una mejor manera de hacer las cosas, y está diseñado para alentar este camino -y en algunos casos desalentar alternativas. Si tu aprendes el "Camino Rails" probablemente descubrirás un tremendo incremento en la productividad. Si tu persistes en traer viejos hábitos de otros lenguajes a tu desarrollo Rails, tratando de utilizar patrones aprendidos en otros sitios, es posible que tengas una experiencia menos feliz.

La filosofía Rails incluye dos principios rectores:

No te repitas a ti mismo (Don't Repeat Yourself): DRY es un principio del desarrollo de software el cual establece "Cada pieza del conocimiento debe ser única, sin ambigüedades, con su propia autoridad y representación dentro de un sistema". Al no escribir la misma información una y otra vez, nuestro código es más fácil de mantener, más reutilizable y con menos errores.

Convención sobre Configuración: Rails tiene sus opiniones acerca de el mejor de hacer muchas cosas en las aplicaciones web, y por defecto las configura como convenciones, en lugar de requerir que se especifiquen minuciosamente a través de ficheros de configuración sin fin.

3 Creando un nuevo Proyecto Rails

El mejor camino para utilizar esta guía es seguir cada paso como ocurre, ningún código o paso se dejará afuera para que puedas seguir literalmente paso por paso hasta el final.

Para seguir hasta el final este capítulo, crearás un proyecto Rails llamado `blog`, un (muy) sencillo weblog. Antes de que puedas empezar a construir una aplicación, necesitas estar seguro de que tienes instalado rails.

El ejemplo de abajo utiliza `$` para representar tu terminal de línea de comandos o consola en un sistema operativo tipo UNIX, aunque puede ser modificado para tener una apariencia diferente. Si tu estás utilizando Windows, tu terminal de línea de comandos puede parecerse a esto `c:\source_code>`

3.1 Instalando Rails

Abre una terminal de línea de comandos. En Mac OS X abre Terminal.app, en Windows elige "Ejecutar" desde tu menú de Inicio y escribe 'cmd.exe'. Cualquier comando precedido de un signo `$` podrá ser ejecutado en la línea de comandos. Verifica que tienes una versión actualizada de Ruby instalada:

Un numero de herramientas existen para ayudar a instalar rápido Ruby y Ruby on Rails, en tu sistema. Los usuarios de Windows users pueden utilizar [Instalador de Rails](#), mientras que los de Mac OS X pueden usar [Tokaido](#). Para más métodos de instalación para la mayoría de los sistemas operativos pueden mirar ruby-lang.org.

```
$ ruby -v
ruby 2.0.0p353
```

En muchos sistemas operativos tipo UNIX encuentras una aceptable version de SQLite3. En Windows, si tu estás instalando Rails a través del Instalador de Rails, ya tendrás SQLite instalado. Otros pueden encontrar instrucciones de instalación en la [Página de SQLite3](#). Verifica que está correctamente instalada en tu PATH:

```
$ sqlite3 --version
```

El programa mostrará su versión.

Para instalar Rails, utiliza el comando `gem install` proporcionado por RubyGems:

```
$ gem install rails
```

Para verificar que tienes todo instalado correctamente, deberías tener la posibilidad de ejecutar el siguiente comando:

```
$ rails --version
```

Si dice algo como "Rails 5.0.0", estás listo para continuar.

3.2 Creando la Aplicación Blog

Rails viene con un numero de scripts llamados generadores (generators) que están diseñados para hacer tu vida de desarrollador más facil creando todo lo que necesitas para empezar a trabajar en una tarea particular. Una de esos es el generador de nuevas aplicaciones Rails, el cual provee lo necesario para crear una aplicación rails fresca, y que no tengas que hacerlo manualmente tu mismo.

Para utilizar este generardor, abre la terminal de línea de comandos, navega hasta el directorio donde tengas permisos para crear ficheros, y escribe:

```
$ rails new blog
```

Esto creará una aplicación Rails llamada Blog en un directorio `blog` e instalará las gemas de las que dependa enunmeradas en el `Gemfile` utilizando `bundle install`.

Puedes ver todas las opciones de los comandos que el constructor de aplicaciones Rails acepta ejecutando `rails new -h`.

Después de crear la aplicación blog, entra en el directorio:

```
$ cd blog
```

El directorio `blog` tiene un numero de ficheros y carpetas auto-generados que constituyen la estructura de una aplicación Rails. La mayor parte del trabajo en este tutorial se hará en la carpeta `app`, pero aqui esta la relación básica entre cada fichero/carpeta y la función que cumple:

Fichero/Carpeta	Propósito
app/	Contiene los controladores, modelos, vistas, funciones de ayuda, envióadores de correo, y ficheros assets fuente (sin compilar) para tu aplicación. Te enfocarás en esta carpeta en lo que queda de este capítulo.
bin/	Contiene los scripts rails para comenzar tu aplicación, y puede contener otros scripts utilizados para configurar, desplegar y ejecutar tu aplicación.
config/	Configurar tu aplicación, rutas, base de datos, y más. Esto se cubre con mayor detalle en Configurando Aplicaciones Rails .
config.ru	Configuración Rack para que los servidores basados en Rack puedan arrancar la aplicación.
db/	Contiene tu actual esquema de datos, como también las migraciones de la base de datos.
Gemfile Gemfile.lock	Estos ficheros te permiten especificar las gemas que necesitas y la dependencias de tu aplicación Rails. Estos ficheros son utilizados por la gema Bundler. Para más información acerca de Bundler, ver Página de Bundler .
lib/	Modulos extendidos para tu aplicación.
log/	Ficheros de log de la aplicación.
public/	La única carpeta que es vista por el mundo exterior tal cual es. Contiene los ficheros estaticos y los ficheros assets compilados.
Rakefile	Este fichero contiene y graba las tareas que pueden ser ejecutadas a través de una linea de comando. Las definiciones de tareas son utilizadas a lo largo de todos de los componentes de Rails. Mejor que cambiar el Rakefile, tu deberías crear tu propias tareas añadiendo ficheros en la carpeta lib/tasks de tu aplicación.
README.rdoc	Este es un breve manual de instrucciones de tu aplicación. Tu puedes editar este fichero para decirle a otros que hace tu aplicación, como configurala, etc.
test/	Test unitarios, ficheros con datos de prueba y otros tipos de test. Esto está cubierto en Probando Aplicaciones Rails .
tmp/	Ficheros temporales (como los de cache e idenficadores de procesos pid).
vendor/	Un lugar por todo el código de terceros. En una aplicación rails típica esto incluye gemas vendidas.

4 ¡Hola, Rails!

Vamos a empezar, escribiendo un texto para verlo en la pantalla rápidamente. Necesitarás ejecutar tu servidor de aplicaciones rails.


4.1 Levantando el Servidor Web

Realmente, ya tienes una aplicación Rails funcional. Para ver como funciona, solo necesitas levantar el servidor web en tu ordenador de desarrollo. Puedes hacer esto ejecutando lo siguiente en el directorio `blog`:

```
$ bin/rails server
```

Compilar CoffeeScript y JavaScript asset comprimidos requiere que tengas software para ejecutar JavaScript disponible en tu sistema, en la ausencia de una verás un error `execjs` durante la compilación de los assets. Usualmente Mac OS X y Windows viene con su interprete para ejecución de JavaScript instalado. Rails añade la gema `therubyracer` al archivo `Gemfile` generado para las nuevas aplicaciones en una línea comentada, puedes descomentarla si la necesitas. `therubyrhino` es la gema recomendada para usuarios de JRuby y es añadida por defecto en el fichero `Gemfile` en aplicaciones generadas bajo JRuby. Puedes investigar todos los interpretes soportados en [ExecJS](#).

Esto levantará WEBrick, un servidor web distribuido con ruby por defecto. Para ver tu aplicación en acción, abre una ventana de navegador en la siguiente dirección <http://localhost:3000>. Deberías ver la página de información por defecto de Rails:



Welcome aboard

You're riding Ruby on Rails!

[About your application's environment](#)

Getting started

Here's how to get rolling:

- 1. Use rails generate to create your models and controllers**

To see all available options, run it without parameters.
- 2. Set up a root route to replace this page**

You're seeing this page because you're running in development mode and you haven't set a root route yet.

Routes are set up in `config/routes.rb`.
- 3. Configure your database**

If you're not using SQLite (the default), edit `config/database.yml` with your username and password.

Browse the documentation

- [Rails Guides](#)
- [Rails API](#)
- [Ruby core](#)
- [Ruby standard library](#)

Para parar el servidor web, teclea Ctrl+C en la terminal donde se está ejecutando el servidor. Para verificar que el servidor ha parado deberías ver el cursor activo en tu terminal otra vez. Para la mayoría de los sistemas tipo UNIX, incluyendo Mac OS X esto debería ser un signo `$`. En el modo de desarrollo, Rails generalmente no requiere que reinicies el servidor; los cambios que tu hagas en los ficheros se transmitirán automáticamente al servidor.

La página de "Bienvenido a bordo" es una *prueba de humo* para una nueva aplicación Rails: asegura que tienes configurado el software lo suficientemente bien para servir una página. También puedes hacer click sobre el enlace *About your application's environment* (acerca el entorno de tu aplicación) para ver un resumen del entorno de tu aplicación.

4.2 Decir "Hola", Rails

Para obtener que Rails diga "Hola", necesitas crear como mínimo un *controlador* y una *vista*.

El propósito de un controlador es recibir una petición específica para la aplicación. *El sistema de enrutamiento*, decide que controlador recibe cada petición. A menudo, hay mas de una ruta hacia cada controlador, y las diferentes rutas pueden ser servidas por diferentes *acciones*. El propósito de cada acción es recoger information para proveersela a una vista.

El propósito de una vista es mostrar esa información en un formato humano legible. Una importante distinción para hacer es que es en el *controlador*, no en la vista, donde la información se recoge. La vista debe solo mostrar esa información. Por defecto, las vistas están escritas en un lenguaje llamado eRuby (Ruby Embebido), el cual es procesado en el ciclo de la petición en Rails antes de ser enviada al usuario.

Para crear un nuevo controlador, necesitaras ejectuar el generador "controller" y decirle que quieres un controlador llamado "bienvenido" con una acción llamada "index", de la siguiente manera:

```
$ bin/rails generate controller bienvenido index
```

Rails creará varios ficheros y rutas por ti.

```
create  app/controllers/bienvenido_controller.rb
route   get 'bienvenido/index'
invoke  erb
create  app/views/bienvenido
create  app/views/bienvenido/index.html.erb
invoke  test_unit
create  test/controllers/bienvenido_controller_test.rb
invoke  helper
create  app/helpers/bienvenido_helper.rb
invoke  assets
invoke  coffee
create  app/assets/javascripts/bienvenido.coffee
invoke  scss
create  app/assets/stylesheets/bienvenido.scss
```

Lo más importante de esto es por supuesto el controlador, localizado en `app/controllers/bienvenido_controller.rb` y la vista, localizada en `app/views/bienvenido/index.html.erb`.

Abre el fichero `app/views/bienvenido/index.html.erb` con tu editor de texto. Borra todo el código existente en el fichero, y reemplazalo con la siguiente linea de código:

```
<h1>¡Hola, Rails!</h1>
```

4.3 Configurando la página de Inicio de la aplicación

Ahora que tenemos hecho el controlador y la vista, necesitamos decirle a Rails cuando queremos que muestre "¡Hola, Rails!". En nuestro caso queremos que lo haga cuando el navegante pida la raíz de nuestro sitio, es decir esta URL, <http://localhost:3000>. Hasta el momento, "Bienvenido a bordo" está ocupando ese lugar.

Para continuar, tienes que decirle a Rails donde tu página de inicio actual se localiza.

Abre el fichero `config/routes.rb` en tu editor.

```
Rails.application.routes.draw do
  get 'bienvenido/index'

  # The priority is based upon order of creation:
  # first created -> highest priority.
  #
  # You can have the root of your site routed with "root"
  # root 'welcome#index'
  #
  # ...
```

Esta es el fichero de *enrutamiento* de tu aplicación el cual mantiene las entradas en un lenguaje especial **DSL (domain-specific language)** que dice a Rails como conectar las peticiones entrantes con los controladores y las acciones. Este fichero contiene muchas líneas comentadas que son rutas de ejemplo, y una de ellas actualmente muestra como conectar a la raíz de tu sitio con una acción y controlador específicos. Encuentra la línea que comienza con `root`, descoméntala y reemplaza `welcome` por `bienvenido`. Debería verse algo como esto:

```
root 'bienvenido#index'
```

`root 'bienvenido#index'` le dice a Rails que derive las peticiones que llegan a la raíz de la aplicación hacia la acción `index` del controlador `bienvenido` y `get 'bienvenido/index'` le indica a Rails que las peticiones que llegan a <http://localhost:3000/bienvenido/index> se deriven a la acción `index` del controlador `bienvenido` también. Esto fue creado automáticamente cuando ejecutasteis por comando el `controller generator` (`rails generate controller bienvenido index`).

Levanta nuevamente el servidor web si lo habías parado para generar el controlador (`rails server`) y abre el navegador en <http://localhost:3000>. Verás el mensaje "¡Hola, Rails!" que has escrito en el fichero `app/views/bienvenido/index.html.erb`, indicando que la nueva ruta en efecto va a la acción `index` del `BienvenidoController` y muestra la vista correctamente.

Para más información acerca del enrutamiento, busca en [Enrutamiento Rails Desde Fuera Hacia Dentro](#).

5 Empezar y Ejecutar

Ahora que has visto como se crea un controlador, una acción y una vista, vamos a crear un poco más ambicioso.

En la aplicación Blog, crearás un nuevo *recurso*. Un recurso es el termino utilizado para una colección de objetos similares, como artículos, gente o animales. Puedes crear, leer, actualizar y destruir elementos de un recurso, estas operaciones son conocidas como operaciones *CRUD*.

Rails provee un método para `recursos` que puedes utilizarlo para declarar un recurso REST. Necesitas añadir el *recurso artículo* al fichero de configuración del enrutamiento `config/routes.rb` de la siguiente manera:

```
Rails.application.routes.draw do

  resources :articulos

  root 'bienvenido#index'
end
```

Si ejecutas `rake routes`, verás que como se han definido las rutas para todas las acciones RESTful estandar. El significado de la columna prefijada (y las otras columnas) lo veremos más adelante, pero ahora nota como Rails ha deducido la forma singular `articulo` y hace un uso significativo de esta.

```
$ bin/rake routes
Prefix Verb  URI Pattern                      Controller#Action
articulos GET   /articulos(.:format)            articulos#index
          POST   /articulos(.:format)            articulos#create
new_articulo GET   /articulos/new(.:format)         articulos#new
edit_articulo GET   /articulos/:id/edit(.:format)    articulos#edit
articulo GET   /articulos/:id(.:format)         articulos#show
          PATCH  /articulos/:id(.:format)         articulos#update
          PUT    /articulos/:id(.:format)         articulos#update
          DELETE /articulos/:id(.:format)         articulos#destroy
root GET    /                                bienvenido#index
```

En la próxima sección, añadiremos la capacidad de crear nuevos articulos en tu aplicación y será poble verlos. Estas son la "C" y la "R" de "CRUD": crear y leer (creation and reading en inglés). La forma de hacer esto se ve así:

Nuevo Artículo

Titular

Contenido

Guardar Artículo

Luce un poco básico por ahora, pero está bien. Nos ocuparemos de mejorar el estilo en cuanto podamos.

5.1 Estableciendo la base del trabajo

Primero, necesitas un espacio dentro de la aplicación para crear un nuevo artículo. Un gran lugar podría ser `/articulos/new`. Con la ruta ya definida, la petición ahora podría hacerse `/articulos/new` a la aplicación. Si vas a <http://localhost:3000/articulos/new> verás un error de enrutamiento:

Este error ocurre porque la ruta necesita tener un controlador definido para poder atender la petición. La solución a este particular

problema es simple: crear un controlador llamado `ArticulosController`. Tu puedes hacerlo ejecutando el siguiente comando:

```
$ bin/rails g controller articulos
```

Routing Error

uninitialized constant ArticulosController

Si abres el recientemente generado `app/controllers/articulos_controller.rb` verás un controlador bastante vacío:

```
class ArticulosController < ApplicationController
end
```

Un controlador es una simple clase que es definida como hija de `ApplicationController`. Es dentro de la clase donde definirás los métodos que se convertirán en acciones para este controlador. Estas acciones desempeñarán las operaciones CRUD sobre los artículos dentro de nuestro sistema.

Hay `public`, `private` y `protected` métodos en Ruby, pero únicamente los métodos `public` pueden ser acciones para los controladores. Para más detalles puedes investigar en [Programando en Ruby](#).

Si recargas ahora <http://localhost:3000/articulos/new>, obtendrás un nuevo error:

Unknown action

The action 'new' could not be found for ArticulosController

Este error indica que Rails no puede encontrar la acción `new` dentro del controlador `ArticulosController` que acabas de generar. Esto es porque cuando los controladores son generados por Rails están vacíos por defecto, a menos que le manifiestes tu deseo en el proceso de generación.

Para definir manualmente una acción dentro del controlador, todo lo que necesitas hacer es definir un método `new` dentro del controlador. Abre `app/controllers/articulos_controller.rb` y dentro de la clase `ArticulosController`, define el método `new` así que tu controlador lucirá ahora de esta manera:

```
class ArticulosController < ApplicationController
  def new
  end
end
```

Con el método `new` definido en `ArticulosController`, si tu recargas <http://localhost:3000/articulos/new> verás otro error:

Tu estás teniendo este error porque Rails espera que acciones llanas como estas tengan la correspondiente vista asociada para mostrar su información. Si no hay una vista disponible, Rails mostrará una excepción.

En la imagen de arriba, la línea de abajo está cortada. Vamos a ver el mensaje completo muestra lo siguiente:

Template is missing

Missing template `articulos/new`, `application/new` with `{locale:[en], formats:[html], handlers:[erb, :builder, :coffee]}`. Buscado en: `*/path/to/blog/app/views`

Desaparecida la plantilla `articulos/new`, `application/new` con `{locale:[en], formats:[html], handlers:[erb, :builder, :coffee]}`. Buscado en: `*/path/to/blog/app/views`

¡Esa es una gran cantidad de texto!

Vamos rápidamente a examinarlo para entender que significa cada parte.

La primera parte identifica que plantilla está perdida. En este caso, es la plantilla `articulos/new`. Rails primero buscará esta plantilla. Si no la encuentra, intentará cargar una plantilla llamada `application/new`. Busca también aquí porque `ArticulosController` es hijo de `ApplicationController`.

La siguiente parte del mensaje contiene un hash. La llave `:locale` en este hash simplemente indica en que idioma la plantilla será recuperada. Por defecto, es la plantilla inglesa - o "en" -. La siguiente clave, `:formats` especifica cual será el formato en que la plantilla será servida en la respuesta. El formato por defecto es `:html`, y así Rails buscará una plantilla `html`. La última llave, `:handlers`, está diciéndonos que *manipulador de plantillas* podrá ser utilizado para mostrar nuestro template. `:erb` es el más comúnmente utilizado para plantillas HTML, `:builder` es utilizado para plantillas XML, y `:coffee` se utiliza CoffeeScript para construir plantillas JavaScript.

El mensaje final nos dice donde Rails ha buscado las plantillas. Plantillas dentro de una aplicación Rails básica como esta son mantenidas en un solo lugar, pero en aplicaciones más complejas pueden estar en lugares muy diferentes.

La más plantilla más simple que podría funcionar en este caso podría ser localizada en `app/views/articulos/new.html.erb`. La extensión en el nombre del fichero en este caso son importantes: la primera extensión es el *formato* de la plantilla, y la segunda extensión es la *sub-rutina* que será utilizada. Rails está intentando encontrar una plantilla llamada `articulos/new` dentro de la vista `app/views` de la aplicación. El formato de esta plantilla solo puede ser `html` y la subrutina solo puede ser una `erb`, `builder` o `coffee`. Porque tu quieres crear un nuevo formulario HTML, deberás utilizar el lenguaje `ERB` que es diseñado para embeber Ruby en HTML.

Por lo tanto el fichero deberá ser llamado `articulos/new.html.erb` y es necesario guardarlo en el directorio `app/views` de la aplicación.

Ve adelante ahora y crea el fichero `app/views/articulos/new.html.erb` y escribe dentro el siguiente contenido:

```
<h1>Nuevo Artículo</h1>
```

Cuando recargues <http://localhost:3000/articulos/new> verás ahora como la página tiene un título. ¡La ruta, el controlador, la acción y la vista ahora están trabajando armoniosamente! Es hora de crear un formulario para crear artículos.

5.2 El primer formulario

Para crear el formulario dentro de esta plantilla, vamos a utilizar un *constructor de formularios*. El principal constructor de formularios para Rails es proporcionado por un método ayudante (helper) llamado `form_for`. Para utilizar este método, añade este código dentro de `app/views/articulos/new.html.erb`:

```

<%= form_for :articulo do |f| %>
  <p>
    <%= f.label :titular %><br>
    <%= f.text_field :titular %>
  </p>

  <p>
    <%= f.label :contenido %><br>
    <%= f.text_area :contenido %>
  </p>

  <p>
    <%= f.submit 'Guardar Artículo' %>
  </p>
<% end %>

```

Si tu refecargas la página ahora, verás exactamente el mismo formulario que en el ejemplo. ¡Construir formularios en Rails es realmente muy fácil!

Cuando tu llamas al `form_for`, le pasas como parametro un identificador de objeto para este formulario. En este caso, es el símbolo `:articulo`. Este le dice al método de ayuda `form_for` para que es el formulario. Dentro del bloque de este método, el objeto `FormBuilder` - representado por la `f` - es utilizado para construir dos etiquetas y dos campos de texto, uno para el título y otro para el texto del artículo. Finalmente, para enviar el formulario con el objeto `f`, se crea un botón `submit` en él.

En español le he añadido el texto 'Guardar Artículo' como parámetro a `f.submit` simplemente para que muestre el texto *Guardar Artículo* en lugar de *Save Artículo* en inglés. Sin este parámetro como está en la guía original también funcionaría pero no se vería como queremos.

Hay un problema con este formulario sin embargo. Si inspeccionas al HTML que se generó, viendo el código fuente de la página, verás que el attribute `action` de este formulario esta apuntando a `/articulos/new`. Este es un problema porque esta ruta va a la misma página en la que tu estás en este momento, y esta ruta debería ser usada para mostrar el formulario de un artículo nuevo.

El formulario necesita utilizar una URL diferente para ir a un lugar diferente. Est puede hacerse simplemente con la opción `:url` del método `form_for`. Es típico en Rails, que la acción que es utilizada para el envío de un nuevo formulario se llame "create", y por tanto el formulario debería apuntar a esta acción.

Edita línea `form_for` dentro de `app/views/articulos/new.html.erb` para que queda así:

```

<%= form_for :articulo, url: articulos_path do |f| %>

```

En este ejemplo, al método de ayuda `articulos_path` se le pasa la opción `:url`. Si quieres saber lo que Rails hará con esto, echemos un vistazo en la salida del comando `rake routes`:

```

$ bin/rake routes

```

Prefix Verb	URI Pattern	Controller#Action
articulos GET	/articulos(.:format)	articulos#index
POST	/articulos(.:format)	articulos#create
new_articulo GET	/articulos/new(.:format)	articulos#new
edit_articulo GET	/articulos/:id/edit(.:format)	articulos#edit
articulo GET	/articulos/:id(.:format)	articulos#show

PATCH	/articulos/:id(.:format)	articulos#update
PUT	/articulos/:id(.:format)	articulos#update
DELETE	/articulos/:id(.:format)	articulos#destroy
root GET	/	bienvenido#index

El metodo de ayuda `articulos_path` le dice a Rails que apunte el formulario hacia el patrón URI asociado con el prefijo `articulos`; y el formulario será (por defecto) enviado por una petición `POST` hacia esa ruta. Esta fue asociada con la acción `create` del actual controlador, el `ArticulosController`.

Con el formulario an sus definidas rutas asociadas, estarás habilitado a rellenar el fomulario y luego de hacer click en el botón submit a comenzar el proceso de crear el nuevo artículo, así que a seguir adelante y hacerlo. Cuando envíes el formulario, deberías ver un error conocido:

Unknown action

The action 'create' could not be found for ArticulosController

Necesitas ahora crear la acción `create` dentro de el `ArticulosController` para que esto funcione.

5.3 Creando artículos

Para hacer que se vaya el "Unknown action", puedes definir una acción `create` dentro de la clase `ArticulosController` en el fichero `app/controllers/articulos_controller.rb`, por debajo de la acción `new`, como se muestra:

```
class ArticulosController < ApplicationController
  def new
    end

  def create
    end
end
```

Si estás reenviando el formulario ahora, verás otro error conocido: la plantilla esta perdida. Está bien, podemos ignorar esto por ahora. Que la acción `create` deberia action guardar el nuevo artículo en la base de datos.

Cuando el formulario es enviado, los campos del formulario son enviados por Rails como *parámetros*. Estos parámetros puden ser invocados dentro de la acción en el controlador, tipicamente para realizar una tarea específica. Para ver como esos parametros se ven, cambia la acción `create` por esto:

```
def create
  render plain: params[:articulo].inspect
end
```

El metodo `render` aqui está tomando un simple hash con una clave `plain` y el valor de `params[:articulo].inspect`. El metodo `params` es un objeto, que representa los parametros (o campos)

que vienen desde el formulario. El método `params` retorna un objeto

`ActiveSupport::HashWithIndifferentAccess`, el cual te habilita a acceder a las claves del hash utilizando o cadenas de texto o símbolos. En esta situación, los únicos parámetros que importan son aquellos que vienen desde el formulario.

Asegurate de tener una sólida comprensión del método `params`, porque lo utilizarás con bastante regularidad. Vamos a considerar este ejemplo de URL: <http://www.example.com/?username=dhh&email=dhh@email.com>. En esta URL, `params[:username]` sería igual a "dhh" y `params[:email]` sería igual a "dhh@email.com".

Si tu reenvías el formulario, una vez más no tendrás ahora el error de plantilla perdida. En su lugar, tendrás algo que se verá así:

```
{ "titular" => "Primer artículo", "contenido" => "Este es mi primer artículo." }
```

Esta acción esta ahora mostrando los parametros del artículo que ha llegado desde el formulario. Sin embargo, esto no es realmente útil. Si puedes ver los parametros pero nada en particular se está haciendo con ellos.

5.4 Creando el Modelo Artículo

Los modelos en Rails utilizan un nombre en singular, y se corresponden con las las tablas de la base de datos que utilizan el nombre pero en plural. Rails provee un generador para crear modelos, que la mayoría de los desarrolladores Rails tienden a utilizar cuando crean nuevos modelos. Para crear un nuevo modelo, ejecuta este comando en tu terminal:

```
$ bin/rails generate model Artículo titular:string contenido:text
```

Con este comando estamos diciéndole a Rails que queremos un modelo `Articulo`, junto con un atributo *titular* del tipo string, y un atributo *contenido* del tipo text. Estos atributos son automáticamente añadidos a la tabla `articulos` en la base de datos y mapeados en el modelo `Articulo`.

Rails responde creando un grupo de ficheros. Por ahora, nosotros solo estamos interesados en `app/models/articulo.rb` y en `db/migrate/20140120191729_create_articulos.rb` (el nombre de tu fichero será un poco diferente). El último es el responsable de crear la estructura de base de datos, la cual veremos más adelante.

Active Record is lo suficientemente inteligente para crear automáticamente una relación entre el nombre de las columnas de la base de datos y los atributos del modelo, esto significa que no deberás declarar atributos dentro de los modelos Rails, esto será hecho automáticamente por Active Record.

5.5 Ejecutando una Migración

Como acabamos de ver, `rails generate model` creo un fichero *de migración de la base de datos* dentro del directorio `db/migrate`. Las migraciones son clases Ruby que están diseñadas para hacer simple de crear y modificar las tablas de la base de datos. Rails utiliza comandos rake para ejecutar las migraciones, y es posible deshacer una migración después de que fue aplicada a la base de datos. Los nombres de los ficheros de migraciones incluyen una marca de tiempo para asegurar que serán procesados en el orden en que fueron creados.

Si miras dentro del fichero `db/migrate/YYYYMMDDHHMMSS_create_articulos.rb` (recuerda, el nombre de tu fichero será levemente diferente), encontrarás esto:

```
class CreateArticulos < ActiveRecord::Migration
  def change
    create_table :articulos do |t|
      t.string :titular
      t.text :contenido

      t.timestamps null: false
    end
  end
end
```

La migración de arriba crea un metodo llamado `change` que será llamado cuando ejecutes esta migración. La acción definida en este método es también reversible, esto significa que Rails conoce como revertir el cambio hecho por esta migración, en caso de que quieras retroceder luego más adelante. Cuando ejecutes la migración ella creará una tabla `articulos` con una columna string y otra de tipo text. También creará dos campos timestamp para permitir a Rails guardar las horas en que se crea y se modifica el artículo.

Para más información acerca de las migraciones, sigue a [Migraciones de la Base de Datos en Rails](#).

En este punto, puedes usar el comando rake para ejecutar la migración:

```
$ bin/rake db:migrate
```

Rails ejecutará el comando de esta migración y te dirá que ha creado la tabla Articulos.

```
== CreateArticulos: migrating
=====
-- create_table(:articulos)
   -> 0.0019s
== CreateArticulos: migrated (0.0020s)
=====
```

Porque estás trabajando en un ambiente de desarrollo por defecto, este comando será aplicado a la base de datos definida en la sección `development` del fichero `config/database.yml`. Si tu quieres ejecutar la migración en otro ambiente, por ejemplo producción, debes añadir explícitamente el entorno en el comando: `rake db:migrate RAILS_ENV=production`.

5.6 Grabando los datos en el controlador

Regresando al `ArticulosController`, necesitamos cambiar la acción `create` para que la utilice el nuevo modelo `Articulo` para grabar los datos en la base de datos. Abre `app/controllers/articulos_controller.rb` y modifica la acción `create` de la siguiente manera:

```
def create
  @articulo = Articulo.new(params[:articulo])

  @articulo.save
```

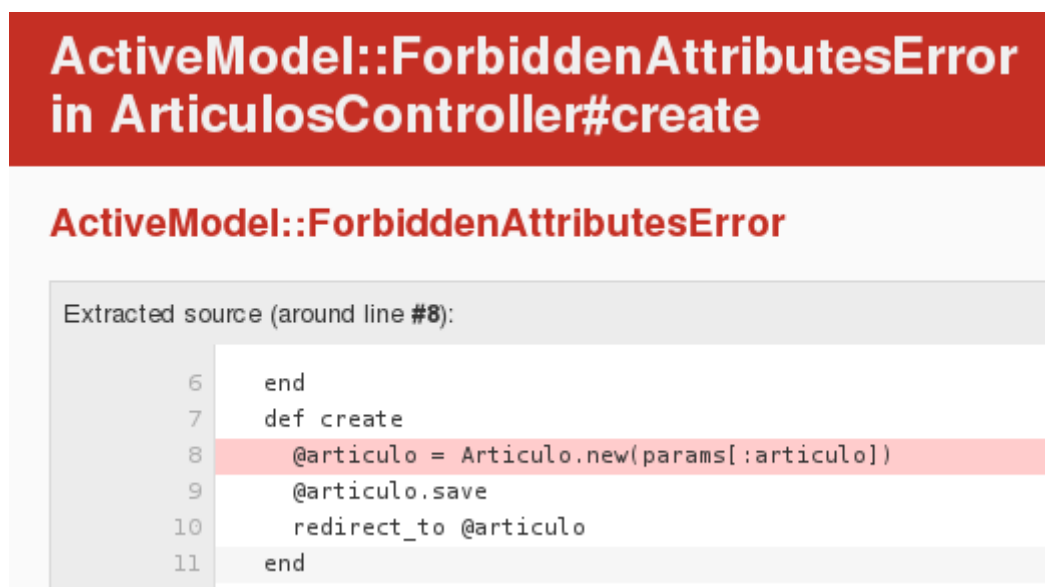
```
    redirect_to @articulo
  end
```

Esto es lo que está pasando: todos los modelos Rails pueden ser inicializados con sus respectivos atributos, los cuales serán automáticamente relacionados con sus respectivas columnas en la base de datos. En la primera línea hemos hecho justo esto (recuerda que `params[:articulo]` contiene los atributos que nos interesa dentro). Luego, `@articulo.save` es el responsable de guardar el modelo en la base de datos. Finalmente redirigimos al usuario a la acción `show`, que definiremos luego.

Tu podrías preguntar porqué la `A` en `Articulo.new` es mayúsculas en el código de encima, mientras la mayoría de referencias a artículos son en minúsculas en este capítulo. En este contexto, nos estamos refiriendo a la clase llamada `Articulo` que está definida en `app/models/articulo.rb`. Los nombres de las clases en Ruby deben comenzar con una letra mayúsculas.

Como veremos más adelante, `@articulo.save` retorna un valor booleano que indica si el artículo fue guardado en la base de datos o no.

Si ahora vas a <http://localhost:3000/articulos/new> estarás *casi siempre* posibilitado para crear un artículo. ¡Pruebalo! Deberías ver el siguiente error:



ActiveModel::ForbiddenAttributesError in ArticulosController#create

ActiveModel::ForbiddenAttributesError

Extracted source (around line #8):

```
6   end
7   def create
8     @articulo = Articulo.new(params[:articulo])
9     @articulo.save
10    redirect_to @articulo
11  end
```

Rails tiene varias características que te ayudarán a escribir aplicaciones seguras, y estás ejecutando una de ellas en este momento. Es llamada **parametros fuertes**, que requiere que le digamos a

Rails exactamente que parametros están permitidos dentro de las acciones de nuestro controlador.

¿Porque tienes que preocuparte? La posibilidad de grabar asignando automáticamente a todos los controladores parámetros de tu modelo de un tirón hace el trabajo de los programadores más fácil, pero esta conveniencia también permite el uso malicioso. ¿Que pasaría si la petición para crear un nuevo artículo se utilizara para incluir campos extra con valores que violaran la integridad de la aplicación? Podrían ser asignados masivamente en el modelo y luego en la base de datos junto con los datos buenos, y potencialmente podrían romper la aplicación o algo peor.

Nosotros tenemos una lista blanca en nuestro controlador con los parametros para prevenir que otros datos se asignen masivamente. En este caso, queremos ambos, permitir y requerir los parámetros `titular` y `text` para validar el uso de `create`. En la sintaxis conoceremos a `require` y `permit`. El cambio solo abarcará una línea en la acción `create`:

```
@articulo = Articulo.new(params.require(:articulo).permit(:titular,
```



```
:contenido))
```

Esto a menudo se hace dentro de un método propio para que pueda ser reutilizado en varias acciones en el mismo controlador, por ejemplo `create` y `update`. Encima y más allá de los problemas de asignación masiva, el método es además escrito como `private` para asegurarse que no puede ser llamado fuera del contexto de su propias intenciones. Aquí está el resultado:

```
def create
  @articulo = Articulo.new(articulo_params)

  @articulo.save
  redirect_to @articulo
end

private
def articulo_params
  params.require(:articulo).permit(:titular, :contenido)
end
```

Para más informacion, sobre lo que indicamos recién se puede leer [este artículo en el blog acerca de los Strong Parameters](#).

5.7 Mostrando Artículos

Si tu envías el formulario otra vez ahora, Rails se quejará al no encontrar la acción `show`. Esto no es lo que queremos, así que vamos añadir la acción `show` antes de continuar.

Como hemos visto en la salida de `rake routes`, la ruta para la acción `show` es la siguiente:

```
articulo GET    /articulos/:id(.:format)  articulos#show
```

La sintaxis especial `:id` le dice a rails que esta ruta espera un parametro `:id`, la que en nuestro caso será el id de un artículo.

Como lo hemos hecho antes, nosotros necesitamos añadir la acción `show` en `app/controllers/articulos_controller.rb` y en su respectiva vista.

Una práctica frecuente es colocar las acciones CRUD estandard en cada controlador en el siguiente orden: `index`, `show`, `new`, `edit`, `create`, `update` y `destroy`. Puedes utilizar el orden que elijas, pero teniendo en mente que son métodos públicos; como mencionamos antes en este capítulo, ellos deben ser ubicados antes que cualquier método privado en el controlador de cualquier metodo `private` o `protected` para que funcionen.

Entonces, vamos a añadir a la acción `show`, lo siguiente:

```
class ArticulosController < ApplicationController
  def show
    @articulo = Articulo.find(params[:id])
  end

  def new
```

```
end
```

```
# recortado para abreviar
```

Un par de cosas para tener en cuenta. Utilizamos `Articulo.find` para encontrar el artículo que nos interesa, pasándole como entrada el parámetro `params[:id]` obtenido del parametro de la petición `:id`. También utilizamos una variable de instancia (prefijada con `@`) para mantener una referencia al objeto artículo. Hacemos esto porque Rails pasará todas las variables de instancia a la vista.

Ahora, creamos un nuevo fichero `app/views/articulos/show.html.erb` con el siguiente contenido:

```
<p>
  <strong>Titular:</strong>
  <%= @articulo.titular %>
</p>

<p>
  <strong>Contenido:</strong>
  <%= @articulo.contenido %>
</p>
```

Con este cambio, debería ser posible finalmente crear nuevos artículos. ¡Visita <http://localhost:3000/articulos/new> y dale una oportunidad!

Titular: ¡Rails es impresionante!

Contenido: Realmente lo es.

5.8 Listando todos los artículos

Todavía necesitamos una manera de listar todos los artículos, así que vamos a hacerlo. La ruta que para esto muestra la salida de `rake routes` es:

```
articulos GET    /articulos(.:format)          articulos#index
```

Añade la correspondiente acción `index` para la ruta dentro de `ArticulosController` en el fichero `app/controllers/articulos_controller.rb`. Cuando escribimos una acción `index`, la práctica habitual es que sea el primer método del controlador. Hagamoslo:

```
class ArticulosController < ApplicationController
  def index
    @articulos = Articulo.all
  end

  def show
    @articulo = Articulo.find(params[:id])
  end

  def new
  end

  # recortado para abreviar
```

Y luego por último, añade la vista para esta acción en el fichero `app/views/articulos/index.html.erb`:

```
<h1>Listando Artículos</h1>

<table>
  <tr>
    <th>Titular</th>
    <th>Contenido</th>
  </tr>

  <% @articulos.each do |articulo| %>
    <tr>
      <td><%= articulo.titular %></td>
      <td><%= articulo.contenido %></td>
    </tr>
  <% end %>
</table>
```

Ahora si vas a <http://localhost:3000/articulos> verás un listado de todos los artículos que has creado.

5.9 Añadiendo enlaces

Ahora puedes crear, mostrar y listar artículos. Ahora vamos a añadir algunos enlaces para navegar a través de estas páginas.

Abre `app/views/bienvenido/index.html.erb` y modifica lo siguiente:

```
<h1>¡Hola, Rails!</h1>
<%= link_to 'Mi Blog', controller: 'articulos' %>
```

El método `link_to` es uno de los ayudantes que tiene Rails para las vistas. Crea un hipervínculo basado en el texto a mostrar y a donde se quiere llegar - en este caso, al listado de artículos.

Vamos a añadir otros enlaces convenientes a nuestras vistas, comenzando por este enlace "Nuevo Artículo" link to `app/views/articulos/index.html.erb`, ubicándolo arriba de la etiqueta `<table>`:

```
<%= link_to 'Nuevo Artículo', new_articulo_path %>
```

Este enlace nos permitirá llegar al formulario para crear un nuevo artículo.

Ahora, añadiremos un nuevo enlace en `app/views/articulos/new.html.erb`, debajo del formulario, para regresar a la acción `index`:

```
<%= form_for :articulo, url: articulos_path do |f| %>
  ...
<% end %>

<%= link_to 'Volver', articulos_path %>
```

Finalmente, añadimos un enlace a la plantilla `app/views/articulos/show.html.erb` para regresar a la acción `index`, así la gente que este mirando un artículo podrá volver a la vista del listado otra vez:

```
<p>
  <strong>Titular:</strong>
  <%= @articulo.titular %>
</p>

<p>
  <strong>Contenido:</strong>
  <%= @articulo.contenido %>
</p>

<%= link_to 'Volver', articulos_path %>
```

Si quieres enlazar a un acción dentro del mismo controlador, no necesitas especificar la opción `:controller`, porque Rails utilizará el controlador actual por defecto.

En modo de desarrollo (en el cual estamos trabajando por defecto), Rails recarga tu aplicación en cada llamada del navegador, así que no necesitas parar y reiniciar el servidor web cuando haces un cambio.

5.10 Añadiendo algunas validaciones

El fichero del modelo, `app/models/articulo.rb` es lo más simple que se puede ser:

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

No hay mucho en este fichero - pero nota que la clase `Articulo` hereda de `ActiveRecord::Base`. `ActiveRecord` suministra gran cantidad de funcionalidades a tu modelo Rails libremente, incluyendo las operaciones básicas de la base de datos CRUD (Crear, Leer, Actualizar y Borrar), validación de datos, también un sofisticado soporte a las búsquedas y habilidades para relacionar multiples modelos con otros.

Rails incluye métodos para ayudar a validar los datos que envías a los modelos. Abre el fichero `app/models/articulo.rb` y editálo:

```
class Articulo < ActiveRecord::Base
  validates :titular, presence: true,
                  length: { minimum: 5 }
end
```

Estos cambios nos aseguran que todos los artículos tendrán un título de como mínimo 5 caracteres de largo. Rails puede validar una variedad de condiciones en el modelo, incluyendo cuando se requieren que no se repitan los valores (índice único), su formato, y la existencia de objetos asociados. Las validaciones son cubiertas con detalle en [Validaciones de Active Record](#).

Con la validación ahora en su lugar, cuando tu llamas a `@articulo.save` con un artículo inválido, retornará `false`. Si abres el `app/controllers/articulos_controller.rb` nuevamente, notarás que no estamos revisando el resultado de llamada `@articulo.save` dentro de la acción `create`. Si `@articulo.save` falla en esta situación, necesitamos que el formulario se vuelva a mostrar al usuario.

Para hacer esto, cambia las acciones `new` y `create` dentro del controlador `app/controllers/articulos_controller.rb` así:

```
def new
  @articulo = Articulo.new
end

def create
  @articulo = Articulo.new(articulo_params)

  if @articulo.save
    redirect_to @articulo
  else
    render 'new'
  end
end

private
def articulo_params
  params.require(:articulo).permit(:titular, :contenido)
end
```

La acción `new` está ahora creando una nueva instancia de la variable llamada `@articulo`, y verás esto en pocos tiempo.

Nota que dentro de la acción `create` utilizamos `render` en lugar de `redirect_to` cuando `save` retorna `false`. El método `render` es utilizado así que el objeto `@articulo` se retorna a la plantilla `new` cuando se despliega. Este despliegue es hecho en la misma petición que envía el formulario, mientras que `redirect_to` le dirá al navegador que emita otra petición.

Si recargas <http://localhost:3000/articulos/new> y pruebas grabar un artículo sin un título, Rails te enviará de regreso al formulario, pero esto no es muy amigable. Necesitas decirle al usuario que hay algo ha ido mal. Para hacer esto, modificarás `app/views/articulos/new.html.erb` para controlar los mensajes de error:

```
<%= form_for :articulo, url: articulos_path do |f| %>

  <% if @articulo.errors.any? %>
    <div id="error_explanation">
      <h2>
        <%= pluralize(@articulo.errors.count, "error") %> han impedido que
el artículo sea grabado:
      </h2>
      <ul>
        <% @articulo.errors.full_messages.each do |msg| %>
          <li><%= msg %></li>
        <% end %>
      </ul>
    </div>
  <% end %>

  <p>
    <%= f.label :titular %><br>
    <%= f.text_field :titular %>
  </p>

  <p>
    <%= f.label :contenido %><br>
```

```

    <%= f.text_area :contenido %>
  </p>

  <p>
    <%= f.submit 'Guardar Comentario' %>
  </p>

<% end %>

<%= link_to 'Volver', articulos_path %>

```

Algunas cosas están en marcha. Estamos controlando si hay errores con `@articulo.errors.any?`, y en caso afirmativo mostraremos un listado de todos ellos con `@articulo.errors.full_messages`.

`pluralize` es un método de apoyo rails que toma un numero y una cadena de texto como argumentos. Si el número es mayor que uno, la cadena será automáticamente convertida al plural.

La razón por la que hemos añadido `@articulo = Artículo.new` en el `ArticulosController` es que en caso contrario `@articulo` podría ser nulo en nuestra vista, y la llamada a `@articulo.errors.any?` podría arrojar un error.

Rails automáticamente envuelve los campos que contienen errores en un div con la clase `field_with_errors`. Puedes definir los estilos css para mostrarlos en forma destacada.

Ahora tenemos un lindo mensaje de error al guardar un artículo sin un título si lo intentas hacer desde el formulario del artículo en <http://localhost:3000/articulos/new>:

Nuevo Artículo

2 errors han impedido que el artículo sea grabado:

- Titular can't be blank
- Titular is too short (minimum is 5 characters)

Los mensajes de error salen por defecto en inglés, si quieres saber como mostrarlos en español puedes investigar en la guía

Internacionalización en Rails API.

5.11 Actualizando Artículos

Hemos cubierto la parte "CR" del CRUD. Ahora vamos a enfocarnos en la parte "U", actualizando (Updating) artículos.

El primer paso que daremos será añadir una acción `edit` al controlador `ArticulosController`, generalmente entre las acciones `new` y `create`, como se muestra aquí:

```

def new
  @articulo = Artículo.new
end

def edit

```

```

    @articulo = Articulo.find(params[:id])
  end

  def create
    @articulo = Articulo.new(articulo_params)

    if @articulo.save
      redirect_to @articulo
    else
      render 'new'
    end
  end
end

```

La vista contendrá un formulario similar al que utilizamos cuando creamos nuevos artículos. Crea un fichero llamado `app/views/articulos/edit.html.erb` y has que tenga lo siguiente:

```

<h1>Editando un artículo</h1>

<%= form_for :articulo, url: articulo_path(@articulo), method: :patch do
  |f| %>

  <% if @articulo.errors.any? %>
    <div id="error_explanation">
      <h2>
        <%= pluralize(@articulo.errors.count, "error") %> han impedido que
el artículo sea grabado:
      </h2>
      <ul>
        <% @articulo.errors.full_messages.each do |msg| %>
          <li><%= msg %></li>
        <% end %>
      </ul>
    </div>
  <% end %>

  <p>
    <%= f.label :titular %><br>
    <%= f.text_field :titular %>
  </p>

  <p>
    <%= f.label :contenido %><br>
    <%= f.text_area :contenido %>
  </p>

  <p>
    <%= f.submit 'Guardar Artículo' %>
  </p>

<% end %>

<%= link_to 'Volver', articulos_path %>

```

Esta vez apuntaremos el formulario a la acción `update`, la cual no hemos definido aún, pero lo haremos pronto.

La opción `method: :patch` le dice a Rails que queremos que este formulario sea enviado via el metodo HTTP `PATCH` es el método que esperarías si hacemos **update** a un recurso de acuerdo al protocolo REST.

El primer parametro de `form_for` puede ser un objeto, es decir, `@articulo` lo que podría causar que este metodo de ayuda rellene el formulario con los campos del propio objeto. Pasarlo en un símbolo (`:articulo`) con el mismo nombre de la variable de instancia (`@articulo`) también automáticamente conduce al mismo comportamiento. Esto es lo que esta ocurriendo aquí.

Si quieres más detalles puedes encontrarlos en [documentacion de form_for](#).

Siguiendo, necesitamos crear la acción `update` en el controlador

`app/controllers/articulos_controller.rb`. Añadela entre la acción `create` y el método `private`:

```
def create
  @articulo = Articulo.new(articulo_params)

  if @articulo.save
    redirect_to @articulo
  else
    render 'new'
  end
end

def update
  @articulo = Articulo.find(params[:id])

  if @articulo.update(articulo_params)
    redirect_to @articulo
  else
    render 'edit'
  end
end

private
def articulo_params
  params.require(:articulo).permit(:titular, :contenido)
end
```

El nuevo método, `update`, es utilizado cuando tu quieres actualizar un registro que ya existía, y acepta un hash que contiene los atributos que tu quieres actualizar. Como antes, si aquí hubo un error en la actualización del artículo querremos mostrar el formulario de nuevo al usuario.

Vamos a reutilizar el método `articulo_params` que hemos definido antes para la acción `create`.

No necesitarás pasarle todos los atributos a `update`. Por ejemplo, si tu llamas a

`@articulo.update(titular: 'Un nuevo título')` Rails podría solo actualizar el atributo `titular`, dejando los otros atributos intactos.

Finalmente, queremos mostrar un enlace a la acción `edit` en el listado de todos los artículos, entonces vamos a añadirlo ahora a `app/views/articulos/index.html.erb`, aparecerá al lado del enlace "Ver":

```
<table>
  <tr>
    <th>Titular</th>
    <th>Conenido</th>
    <th colspan="2"></th>
  </tr>

  <% @articulos.each do |articulo| %>
    <tr>
```



```

      <td><%= articulo.titular %></td>
      <td><%= articulo.contenido %></td>
      <td><%= link_to 'Ver', articulo_path(articulo) %></td>
      <td><%= link_to 'Editar', edit_articulo_path(articulo) %></td>
    </tr>
  <% end %>
</table>

```

Y también añadiremos uno a la plantilla `app/views/articulos/show.html.erb`, y también habrá un enlace "Editar" en la página del artículo. Escribe esto al final de la plantilla:

```

...

<%= link_to 'Volver', articulos_path %> |
<%= link_to 'Editar', edit_articulo_path(@articulo) %>

```

Como nuestra aplicación se ve hasta aquí:

Listado de Artículos

[Nuevo Artículo](#)

Titular	Conenido	
Bienvenido a Rails	Ejemplo	Ver Editar
iRails es impresionante!	Realmente lo es.	Ver Editar

5.12 Utilizando partials para eliminar la duplicidad en las vistas

Nuestra página `edit` se ve muy parecida a la página `new`; en realidad, ambas comparten el mismo código para el

despliegue del formulario. Vamos a quitar esta duplicidad utilizando una vista parcial. Por convención, los ficheros partial tienen como prefijo un guión bajo "_".

Puedes leer más acerca de los partials en el capítulo [Plantillas y Renderizado en Rails](#).

Crea un nuevo fichero `app/views/articulos/_form.html.erb` con el siguiente contenido:

```

<%= form_for @articulo do |f| %>

  <% if @articulo.errors.any? %>
    <div id="error_explanation">
      <h2>
        <%= pluralize(@articulo.errors.count, "error") %> han impedido que
        el artículo sea grabado:
      </h2>
      <ul>
        <% @articulo.errors.full_messages.each do |msg| %>
          <li><%= msg %></li>
        <% end %>
      </ul>
    </div>
  <% end %>

  <p>
    <%= f.label :titular %><br>

```

```

    <%= f.text_field :titular %>
  </p>

  <p>
    <%= f.label :contenido %><br>
    <%= f.text_area :contenido %>
  </p>

  <p>
    <%= f.submit 'Guardar Artículo' %>
  </p>

<% end %>

```

Todo menos la declaración `form_for` permanece igual. La razón por la que podemos hacerlo más reducido, simplificando la declaración `form_for` soportando dos formularios en uno, es que `@articulo` es un *recurso* correspondiente a toda la configuración de las rutas RESTful, y Rails es capaz de inferir que URI y método utilizar en cada caso.

Para más información acerca del uso de `form_for`, mirar [Estilo orientado a recursos](#).

Ahora, vamos a actualizar la vista `app/views/articulos/new.html.erb` con este nuevo partial, rescribiéndola completamente:

```

<h1>Nuevo Artículo</h1>

<%= render 'form' %>

<%= link_to 'Volver', articulos_path %>

```

Luego hacemos lo mismo en la vista `app/views/articulos/edit.html.erb`:

```

<h1>Editar el artículo</h1>

<%= render 'form' %>

<%= link_to 'Volver', articulos_path %>

```

5.13 Borrando Artículos

Ahora estamos listos para cubrir la parte "D" del CRUD, borrando (deleting) artículos de la base de datos. Siguiendo la convención REST, la ruta para borrar artículos según la salida del comando `rake routes` es:

```
DELETE /articulos/:id(.:format)      articulos#destroy
```

El método `delete` según el enrutamiento debería ser utilizado para borrar recursos. Si esto fuera la típica ruta `get`, este podría ser invocado por la gente para que utilizando URLs maliciosas como esta:

```
<a href='http://example.com/articulos/1/destroy'>¡Mira este gato!</a>
```

Nosotros utilizamos el método `delete` para borrar recursos, y esta ruta es dirigida a la acción `destroy` dentro del controlador `app/controllers/articulos_controller.rb`, el cual no existe aún. El método `destroy` es generalmente la última acción CRUD en el controlador, y como las otras acciones públicas CRUD, debe ser escrita antes de cualquier método `private` o `protected`. Vamos a añadirlo:

```
def destroy
  @articulo = Artículo.find(params[:id])
  @articulo.destroy

  redirect_to articulos_path
end
```

El controlador completo `ArticulosController` en el fichero `app/controllers/articulos_controller.rb` ahora lucirá así:

```
class ArticulosController < ApplicationController
  def index
    @articulos = Artículo.all
  end

  def show
    @articulo = Artículo.find(params[:id])
  end

  def new
    @articulo = Artículo.new
  end

  def edit
    @articulo = Artículo.find(params[:id])
  end

  def create
    @articulo = Artículo.new(articulo_params)

    if @articulo.save
      redirect_to @articulo
    else
      render 'new'
    end
  end

  def update
    @articulo = Artículo.find(params[:id])

    if @articulo.update(articulo_params)
      redirect_to @articulo
    else
      render 'edit'
    end
  end

  def destroy
    @articulo = Artículo.find(params[:id])
    @articulo.destroy

    redirect_to articulos_path
  end

  private
```

```

    def articulo_params
      params.require(:articulo).permit(:titular, :contenido)
    end
  end
end

```

Con el método `destroy` puedes destruir un objeto Active Record cuando quieras borrarlo de la base de datos. Noa que no necesitamos añadir una vista para esta acción al redirigir al final a la acción `index`.

Finalmente, añade un enlace 'Borrar' en tu plantilla `index` (`app/views/articulos/index.html.erb`) para envolver todas las acciones junta.

```

<h1>Listando Artículos</h1>
<%= link_to 'Nuevo artículo', new_articulo_path %>
<table>
  <tr>
    <th>Titular</th>
    <th>Contenido</th>
    <th colspan="3"></th>
  </tr>

  <% @articulos.each do |articulo| %>
    <tr>
      <td><%= articulo.titular %></td>
      <td><%= articulo.contenido %></td>
      <td><%= link_to 'Ver', articulo_path(articulo) %></td>
      <td><%= link_to 'Editar', edit_articulo_path(articulo) %></td>
      <td><%= link_to 'Borrar', articulo_path(articulo),
        method: :delete,
        data: { confirm: '¿Estás seguro?' } %></td>
    </tr>
  <% end %>
</table>

```

Aquí estamos utilizando el ayudante `link_to` de diferente manera. Le pasaremos la ruta mencionada de segundo argumento. Las opciones del método `:method` y `'data-confirm'` son utilizadas como atributos HTML5 así que cuando el enlace es pinchado, Rails mostrará primero un dialogo de confirmación al usuario y luego enviará al enlace con el metodo `delete`. Esto es hecho a través de un fichero JavaScript `jquery_ujs` que fue automaticamente incluido en la plantilla principal (`app/views/layouts/application.html.erb`) cuando has creado la aplicación. Sin este fichero, la caja con el diálogo de confirmación no aparecería.

Felicitaciones, ahora puedes crear, mostrar, listar y borrar artículos.

En general, Rails anima a utilizar los objetos recursos en lugar de declarar las rutas manualmente. Para más información acerca de las rutas, mirar [Rutas Rails desde afuera hacia adentro](#).

6 Añadiendo un Segundo Modelo

Es hora de añadir un segundo modelo a la aplicación. El segundo modelo manejará los comentarios de los artículos.

6.1 Generando el Modelo

Vamos a mirar el mismo generador que hemos utilizado antes cuando creamos el modelo `Articulo`. En este momento crearemos un modelo `Comentario` donde se guardarán los comentarios de cada artículo.

Listando Artículos

[Nuevo artículo](#)

Titular	Contenido	
Bienvenido a Rails	Ejemplo	Ver Editar Borrar
¡Rails es impresionante! Realmente lo es.		Ver Editar Borrar

¿Estás seguro?

Cancelar

Aceptar

Ejecuta el siguiente comando en tu terminal:

```
$ bin/rails generate model Comentario comentarista:string contenido:text  
articulo:references
```

Este comando generará cuatro ficheros:

Fichero	Propósito
db/migrate/20140120201010_create_comentarios.rb	La migración para crear la tabla comentarios en tu base de datos (el nombre de tu fichero incluirá una marca de tiempo diferente)
app/models/comentario.rb	El modelo Comentario
test/models/comentario_test.rb	Arnés de pruebas para el modelo comentarios
test/fixtures/comentarios.yml	Comentarios para utilizar en pruebas

Primero, echar un vistazo a `app/models/comentario.rb`:

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

Este es muy parecido al modelo `Articulo` que hemos visto antes. La diferencia es la línea `belongs_to :articulo`, la cual configura una *asociación* Active Record. Aprenderás un poco acerca de las asociaciones en el próximo apartado de esta guía.

Además del modelo, Rails también generó una migración para crear la correspondiente tabla en la base de datos:

```
class CreateComentarios < ActiveRecord::Migration
  def change
    create_table :comentarios do |t|
      t.string :comentarista
      t.text :contenido

      # esta línea añade un entero llamado `articulo_id`.
      t.references :articulo, index: true

      t.timestamps null: false
    end
    add_foreign_key :comentarios, :articulos
  end
end
```

La línea `t.references` configura una clave foránea para la asociación entre dos modelos. También se crea un índice para esta asociación en esta columna.

Sigamos adelante y ejecutemos la migración:

```
$ bin/rake db:migrate
```

Rails es lo suficientemente inteligente para ejecutar solo las migraciones que no se han ejecutado todavía en la actual base de datos, así que en este caso verás lo siguiente:

```
== CreateComentarios: migrating
=====
-- create_table(:comentarios)
--> 0.0115s
-- add_foreign_key(:comentarios, :articulos)
--> 0.0000s
== CreateComentarios: migrated (0.0119s)
=====
```

6.2 Modelos Asociados

Las asociaciones Active Record te permiten declarar fácilmente las relaciones entre dos modelos. En el caso de los comentarios y los artículos, puedes escribir la relación de esta manera:

Cada comentario pertenece a un artículo.
Un artículo tiene muchos comentarios.

En realidad, es muy cercana la sintaxis que Rails utiliza para declarar esta asociación. Acabas de ver la línea de código dentro del modelo `Comentario` (`app/models/comentario.rb`) que hace que cada comentario pertenezca a un Artículo:

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

Necesitarás editar `app/models/articulo.rb` para añadir la otra parte de la asociación:

```
class Articulo < ActiveRecord::Base
  has_many :comentarios
  validates :titular, presence: true,
              length: { minimum: 5 }
end
```

Estas dos declaraciones habilitan un poco de mantenimiento automático. Por ejemplo, si tu tienes una instancia de la variable `@articulo` conteniendo un artículo, puedes obtener todos los comentarios que pertenecen a este artículo como un arreglo utilizando `@articulo.comentarios`.

Para más información sobre las asociaciones Active Record, ver la guía [Asociaciones Active Record](#).

6.3 Añadiendo una Ruta para los Comentarios

Como en el controlador `bienvenido`, necesitamos añadir una ruta para que así Rails conozca donde queremos navegar para ver los `comentarios`. Abre el fichero `config/routes.rb` y edita lo siguiente:

```
resources :articulos do
  resources :comentarios
end
```

Esto crea `comentarios` como un *recurso anidado* dentro de `articulos`. Esta es la otra parte de capturar las relaciones de herencia que existen entre artículos y comentarios.

Para más información acerca de enrutamiento, ver el capítulo [Enrutamiento en Rails](#).

6.4 Generando un Controlador

Con el modelo en la mano, puedes poner la atención en crear el correspondiente controlador. Ora vez, utilizaremos el mismo generador que antes:

```
$ bin/rails generate controller Comentarios
```

Esto creará cinco ficheros y un directorio vacío:

Fichero/Directorio	Propósito
<code>app/controllers/comentarios_controller.rb</code>	El controlador Comentarios
<code>app/views/comentarios/</code>	Las vistas del controlador se guardarán aquí
<code>test/controllers/comentarios_controller_test.rb</code>	Las pruebas para el controlador
<code>app/helpers/comentarios_helper.rb</code>	El código de apoyo ayudante de las vistas

Fichero/Directorio	Propósito
app/assets/javascripts/comentario.coffee	CoffeeScript para el controlador
app/assets/stylesheets/comentario.scss	Hojas de estilo en cascada para el controlador

Como cualquier blog, nuestros lectores crearán sus comentarios directamente después de leer el artículo, y una vez que añadan sus comentarios, los enviaremos de regreso a la página donde se muestra el artículo para que vea su comentario en la lista. Debido a esto, nuestro `ComentariosController` está aquí para proveernos un método para crear comentarios y borrar los comentarios spam cuando lleguen.

Así primero, retocaremos la plantilla show del artículo (`app/views/articulos/show.html.erb`) para que nos deje hacer un nuevo comentario:

```
<p>
  <strong>Titular:</strong>
  <%= @articulo.titular %>
</p>

<p>
  <strong>Contenido:</strong>
  <%= @articulo.contenido %>
</p>

<h2>Añadir un comentario:</h2>
<%= form_for([@articulo, @articulo.comentarios.build]) do |f| %>
  <p>
    <%= f.label :comentarista %><br>
    <%= f.text_field :comentarista %>
  </p>
  <p>
    <%= f.label :contenido %><br>
    <%= f.text_area :contenido %>
  </p>
  <p>
    <%= f.submit 'Guardar Comentario' %>
  </p>
<% end %>

<%= link_to 'Volver', articulos_path %> |
<%= link_to 'Editar', edit_articulo_path(@articulo) %>
```

Esto añade un formulario en la página show del `Articulo` que crea un nuevo comentario llamado por la acción `create` del `ComentariosController`. El `form_for` llamado aquí utiliza un arreglo, con el que se construirá una ruta anidada, tal como `/articulos/1/comentarios`.

Vamos a retocar la acción `create` en `app/controllers/comentarios_controller.rb`:

```
class ComentariosController < ApplicationController
  def create
    @articulo = Articulo.find(params[:articulo_id])
    @comentario = @articulo.comentarios.create(comentario_params)
    redirect_to articulo_path(@articulo)
  end

  private
```



```

    def comentario_params
      params.require(:comentario).permit(:comentarioer, :contenido)
    end
  end
end

```

Puedes ver un poco más de complejidad aquí que en el controlador de artículos. Este es un efecto colateral del anidamiento que has configurado. Cada petición de crear un comentario tiene que mantener el artículo al que se añade, así la llamada inicial al método `find` del modelo `Articulo` es para obtener el artículo en cuestión.

Además, el código tiene la ventaja de contar con algunos de los métodos disponibles para una asociación. Utilizamos el método `create` en `@articulo.comentarios` para crear y guardar el comentario. Esto automáticamente vinculará el comentario con el artículo particular al que pertenece.

Una vez hagamos el nuevo comentario, enviaremos al usuario de regreso al artículo original utilizando el ayudante `articulo_path(@articulo)`. Como hemos visto, esta llamará a la acción `show` del `ArticulosController` que activará el despliegue de la plantilla `show.html.erb`. Eso es donde queremos que se muestre el comentario, entonces vamos a añadir a `app/views/articulos/show.html.erb`.

```

<p>
  <strong>Titular:</strong>
  <%= @articulo.titular %>
</p>

<p>
  <strong>Contenido:</strong>
  <%= @articulo.contenido %>
</p>

<h2>Comentarios</h2>
<% @articulo.comentarios.each do |comentario| %>
  <p>
    <strong>Comentarista:</strong>
    <%= comentario.comentarista %>

    <p>
      <strong>Comentario:</strong>
      <%= comentario.contenido %>
    </p>
  </p>
<% end %>

<h2>Añadir comentario:</h2>
<%= form_for([@articulo, @articulo.comentarios.build]) do |f| %>
  <p>
    <%= f.label :comentarista %><br>
    <%= f.text_field :comentarista %>
  </p>
  <p>
    <%= f.label :contenido %><br>
    <%= f.text_area :contenido %>
  </p>
  <p>
    <%= f.submit 'Guardar Comentario' %>
  </p>
<% end %>

<%= link_to 'Editar Artículo', edit_articulo_path(@articulo) %> |
<%= link_to 'Volver a Artículos', articulos_path %>

```

Ahora puedes añadir artículos y comentarios a tu blog y se ven en el lugar que les corresponde..

Titular: Bienvenido a Rails

Contenido: Ejemplo

Comentarios

Comentarista: amigo programador

Comentario: Que buen ejemplo

[Borrar Comentario](#)

Añade un comentario:

Comentarista

Contenido

Guardar Comentario

[Editar Artículo](#) | [Volver a Articulos](#)

7 Refactorizando

Ahora que tenemos artículos y comentarios funcionando, echemos un vistazo a la plantilla

`app/views/articulos/show.html.erb`. Se está volviendo muy largo y esto no es bueno. Podemos utilizar `partials` (vistas parciales) para limpiarlo.

7.1 Mostrando Colecciones en Vistas Parciales

Primero, vamos haremos un `partial` comentario para poder mostrar todos los comentarios de un artículo. Crea el fichero `app/views/comentarios/_comentario.html.erb` y escribe lo siguiente dentro:

```
<p>
  <strong>Comentarista:</strong>
  <%= comentario.comentarista %>
</p>

<p>
  <strong>Comentario:</strong>
  <%= comentario.contenido %>
</p>
```

Luego puedes cambiar `app/views/articulos/show.html.erb` para que se vea de esta manera:

```
<p>
  <strong>Titular:</strong>
  <%= @articulo.titular %>
</p>

<p>
  <strong>Contenido:</strong>
  <%= @articulo.text %>
</p>

<h2>Comentarios</h2>
<%= render @articulo.comentarios %>

<h2>Añadir un comentario:</h2>
```

```

<%= form_for([@articulo, @articulo.comentarios.build]) do |f| %>
  <p>
    <%= f.label :comentarista %><br>
    <%= f.text_field :comentarista %>
  </p>
  <p>
    <%= f.label :contenido %><br>
    <%= f.text_area :contenido %>
  </p>
  <p>
    <%= f.submit 'Guardar Comentario' %>
  </p>
<% end %>

<%= link_to 'Editar Artículo', edit_articulo_path(@articulo) %> |
<%= link_to 'Volver a Artículos', articulos_path %>

```

Este ahora desplegará el la vista parcial en `app/views/comentarios/_comentario.html.erb` una por cada comentario que estén en la colección `@articulo.comentarios`. Como el método `render` itera sobre la colección `@articulo.comentarios`, este asigna a cada comentario una variable local llamada en la misma vista parcial, en este caso `comentario` la cual esta luego disponible en la vista parcial para que lo podamos mostrar.

7.2 Desplegando el Formulario Parcial

Vamos a mover también aquella sección de nuevo comentario fuera en su propia vista parcial. Otra vez, crea un fichero `app/views/comentarios/_form.html.erb` que contenga:

```

<%= form_for([@articulo, @articulo.comentarios.build]) do |f| %>
  <p>
    <%= f.label :comentarista %><br>
    <%= f.text_field :comentarista %>
  </p>
  <p>
    <%= f.label :contenido %><br>
    <%= f.text_area :contenido %>
  </p>
  <p>
    <%= f.submit 'Guardar Comentario' %>
  </p>
<% end %>

```

Entonces harás que se vea `app/views/articulos/show.html.erb` de la siguiente manera:

```

<p>
  <strong>Titular:</strong>
  <%= @articulo.titular %>
</p>

<p>
  <strong>Contenido:</strong>
  <%= @articulo.contenido %>
</p>

<h2>Comentarios</h2>
<%= render @articulo.comentarios %>

```

```

<h2>Añade un comentario:</h2>
<%= render 'comentarios/form' %>

<%= link_to 'Editar Artículo', edit_articulo_path(@articulo) %> |
<%= link_to 'Volver a Artículos', articulos_path %>

```

El segundo render que se acaba de definir es la plantilla parcial que queremos desplegar, `comentarios/form`. Rails es lo suficientemente inteligente para detectar la barria diagonal de la cadena y se da cuenta que lo que quieres incluir es el fichero `_form.html.erb` del directorio `app/views/comentarios`.

El objeto `@articulo` está disponible para cualquier vista parcial en la vista porque lo hemos definido como una variable de instancia.

8 Borrando Comentarios

Otra característica importante de un blog es que pueda borrar comentarios spam. Para hacer esto, necesitamos implementar un enlace de algún tipo en la vista y una acción `destroy` en el controlador `ComentariosController`.

Entonces primero, vamos a añadir el enlace borrar en la vista parcial

`app/views/comentarios/_comentario.html.erb`:

```

<p>
  <strong>Comentarista:</strong>
  <%= comentario.comentarista %>
</p>

<p>
  <strong>Comentario:</strong>
  <%= comentario.contenido %>
</p>

<p>
  <%= link_to 'Borrar Comentario', [comentario.articulo, comentario],
    method: :delete,
    data: { confirm: '¿Estás seguro?' } %>
</p>

```

Pinchando sobre el enlace "Borrar Comentario" se dispara una petición `DELETE` `/articulos/:articulo_id/comentarios/:id` a nuestro `ComentariosController`, el cual puede ser utilizado para encontrar el comentario que necesitamos borrar, entonces vamos a añadir la acción `destroy` a nuestro controlador (`app/controllers/comentarios_controller.rb`):

```

class ComentariosController < ApplicationController
  def create
    @articulo = Artículo.find(params[:articulo_id])
    @comentario = @articulo.comentarios.create(comentario_params)
    redirect_to articulo_path(@articulo)
  end

  def destroy
    @articulo = Artículo.find(params[:articulo_id])
    @comentario = @articulo.comentarios.find(params[:id])
    @comentario.destroy
    redirect_to articulo_path(@articulo)
  end
end

```

```

end

private
  def comentario_params
    params.require(:comentario).permit(:comentarista, :contenido)
  end
end

```

La acción `destroy` encontrará el artículo que estamos mirando, localiza el comentario, encontrará el comentario dentro de la colección `@articulo.comentarios`, y luego de borrarla de la base de datos nos enviará de regreso a la acción `show` del artículo.

8.1 Borrando Objetos Asociados

Si tu estás borrando un artículo, sus comentarios asociados también es necesario borrarlos, de otro modo ocuparán espacio en la base de datos. Para lograr este cometido Rails te permite utilizar la opción `dependent` de la asociación. Modifica el modelo `Articulo`, `app/models/articulo.rb`, como sigue:

```

class Articulo < ActiveRecord::Base
  has_many :comentarios, dependent: :destroy
  validates :titular, presence: true,
              length: { minimum: 5 }
end

```

9 Seguridad

9.1 Autenticación Básica

Si estás publicando tu blog online, cualquier persona podría añadir, editar y borrar artículos o borrar comentarios.

Rails provee un sistema de autenticarse en HTTP muy simple que te permitirá trabajar en esta situación.

En el `ArticulosController` necesitamos tener una manera para bloquear el acceso a varias acciones si la persona no esta autenticada. Aquí podemos utilizar el método `http_basic_authenticate_with` de Rails, que permite acceder a las acciones solicitadas solo si pasamos la autenticación.

Para utilizar el sistema de autenticación, vamos a especificarlo al principio de nuestro `ArticulosController` en `app/controllers/articulos_controller.rb`. En nuestro caso, queremos autenticar al usuario en todas las acciones exceptuando las acciones `index` y `show`, entonces escribiremos esto:

```

class ArticulosController < ApplicationController

  http_basic_authenticate_with name: "dhh", password: "secreto", except:
    [:index, :show]

  def index
    @articulos = Articulo.all
  end

  # recortado para abreviar

```

También queremos permitir que solo los usuarios autenticados puedan borrar comentarios, entonces en el `ComentariosController` (`app/controllers/comentarios_controller.rb`) escribimos:

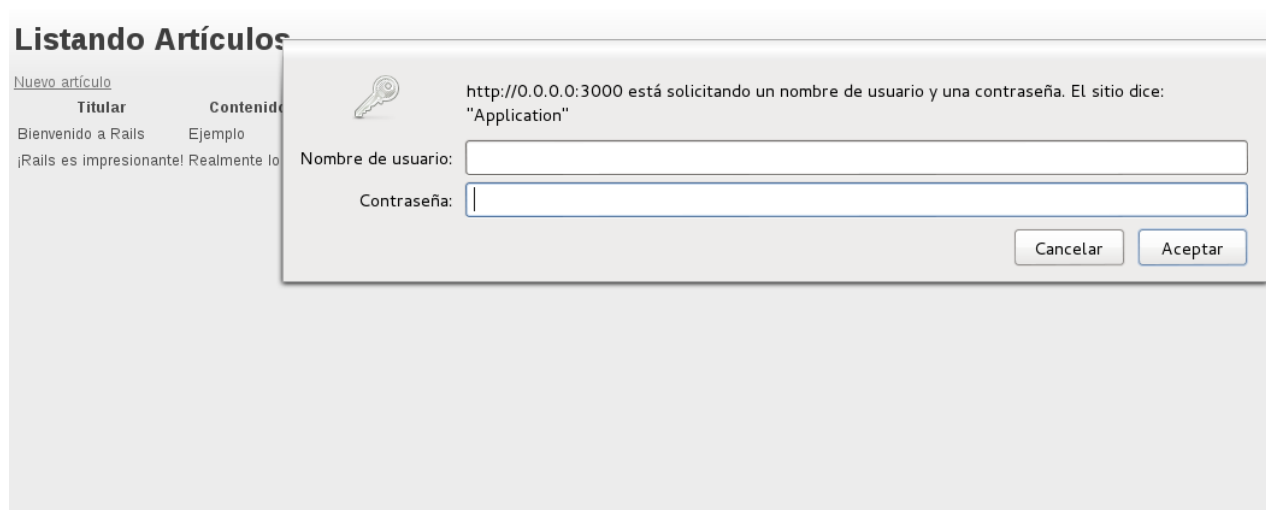
```
class ComentariosController < ApplicationController

  http_basic_authenticate_with name: "dhh", password: "secreto", only:
:destroy

  def create
    @articulo = Articulo.find(params[:articulo_id])
    # ...
  end

  # recortado para abreviar
```

Ahora si tratas de crear un nuevo artículo, serás recibido por un requerimiento de autenticación básica HTTP:



Otros métodos de autenticación están disponibles para aplicaciones Rails. Dos muy populares son el motor rails [Devise](#) y la gema [Authlogic](#), junto con otros tantos.

9.2 Otra Consideración de Seguridad

La seguridad, especialmente en aplicaciones web, es un campo amplio y detallado. La seguridad en tus aplicaciones rails está cubierta en mayor profundidad en el capítulo [Guía de Seguridad en Ruby on Rails](#).

10 ¿Que es lo que sigue?

Ahora que has visto tu primera aplicación Rails, deberías sentirte libre para actualizarte y experimentar por ti mismo.

Recuerda que no debes hacer todo sin ayuda. Cuando necesites ayuda para comenzar y desenvolverte con Rails, sientete libre de consultar estos recursos de soporte:

Las [Guías de Ruby on Rails](#)

El [Tutorial de Ruby on Rails](#)

Las [Listas de correo de Ruby on Rails](#)

El canal [#rubyonrails](#) en irc.freenode.net

Rails también viene con una ayuda para construir con la utilidad de la línea de comandos:

Ejecutar `rake doc:guides` pondrá una copia entera de las Guías Rails en la carpeta `doc/guides` de tu aplicación. Abre `doc/guides/index.html` en tu navegador web para explorar las Guías.

Ejecutar `rake doc:rails` pondrá una copia entera de la documentación API de Rails en la carpeta `doc/api` de tu aplicación. Abre `doc/api/index.html` en tu navegador web para explorar la documentación API.

Para poder generar las Guías Rails localmente con el comando `rake doc:guides` necesitas instalar las gemas Redcarpet y Nokogiri. Añadelas a tu fichero `Gemfile` y ejecuta `bundle install` y estarás listo para empezar.

11 Configuración de Gotchas

La manera más sencilla de trabajar con Rails es grabar todos los datos externos como UTF-8. Si no lo haces, las librerías Ruby y Rails a menudo podrán convertir tus datos nativos en UTF-8, pero esto no siempre funciona de forma fiable, entonces lo mejor es asegurarse que todos los datos exteriores sean UTF-8.

Si has cometido un error en esta area, lo más común es encontrar un diamante negro con signo de interrogación en el navegador. Otro síntoma común es que un se ve caracter como "Ã¼" en lugar de "ü". Rails toma un numero interno de pasos para mitigar las causas más comunes de estos problemas que pueden ser automáticamente detectados y corregidos. Sin embargo, si tu tienes datos externos que no son grabados como UTF-8, esto puede ocasionalmente resultar en esta clases de problemas que no pueden ser detectados automaticamente por Rails y corregidos.

Dos muy comunes fuentes de datos que no son UTF-8:

Tu editor de textos: La mayoría de los editores de texto (tal como TextMate), por defecto graban los ficheros como UTF-8. Si tu editor de textos no lo hace, esto puede devolver caracteres especiales en tus plantillas (tal como `é`) que aparecerá como un diamante con un signo de pregunta dentro en tu navegador. Esto también se aplica a tus ficheros de traducción i18n. La mayoría de los editores que no funcionan grabando por defecto en UTF-8 (tal como algunas versiones de Dreamweaver) ofrecen una manera de cambiar los caracteres por defecto a UTF-8. Entonces hazlo.

Tu base de datos: Rails por defecto convierte los datos desde tu base de datos en UTF-8 en el límite. Sin embargo, si tu base de datos no está utilizando internamente UTF-8, podría darse el caso de que no sea posible guardar todos los caracteres que los usuarios ingresen. Por instancia, si tu base de datos está utilizando Latin-1 internamente, y sus usuarios ingresan un caracter en Ruso, Hebreo o Japonés, los datos serán perdidos para siempre una vez que entren en la base de datos. Si es posible, utiliza UTF-8 como la forma de grabar interna en tu base de datos.

Participa

Se te anima a ayudar a mejorar la calidad de esta guía.

Por favor contribuye si hay errores tipográficos o de conceptos. Para empezar, puedes leer nuestra sección de colaboración con la sección de documentación [contribuciones a la documentación](#).

También es posible encontrar contenidos incompletos, o cosas que no están actualizadas. Por favor añade cualquier documentación faltante en master. Asegúrate de revisar las guías [Las guías paralelas](#)

primero para verificar si las cuestiones ya se han subido o no a la rama principal. Comprueba las [**Directrices de Ruby on Rails**](#) para el estilo y convenciones.

Si por cualquier razón encuentras algo para arreglar, pero no lo puedes por ti mismo, por favor, abre una incidencia [**abrir una incidencia**](#).

Y por último pero no menos importante, cualquier tipo de discusión con respecto a la documentación de Ruby on Rails es muy bienvenida en la lista de correo RubyOnRails-docs [**lista de correo rubyonrails-docs**](#).

Este trabajo está bajo la licencia [**Creative Commons Attribution-ShareAlike 4.0 International**](#)

"Rails", "Ruby on Rails", y el logo de Rails logo son marcas comerciales de David Heinemeier Hansson. Todos los derechos reservados.