

Libro de Introducción a Ruby on Rails

Autor: Uniwebsidad

PDF basado en:

<https://uniwebsidad.com/libros/introduccion-rails/>

Índice de contenidos

1. Antes de empezar
2. ¿Qué es Rails?
3. Creando un nuevo proyecto
4. Hola, Rails!
5. Creando un nuevo post
6. Añadiendo otro modelo
7. Refactorización
8. Borrando comentarios
9. Seguridad
10. Siguiendo pasos

Capítulo 1. Antes de empezar

Esta guía está diseñada para principiantes que quieren comenzar con Ruby on Rails desde cero. No es necesario que tengas ninguna experiencia previa con Rails. Sin embargo, para obtener el máximo de esta guía, tu ordenador sí que debe cumplir algunos requisitos técnicos. En concreto, debes disponer de:

- El lenguaje de programación [Ruby](#) en su versión 1.9.3 o mayor.
- El sistema de paquetes [RubyGems](#). Si quieres aprender más acerca de RubyGems, puedes leer la guía en inglés: [RubyGems User Guide](#).
- La base de datos [SQLite3](#).

Rails es un framework para aplicaciones web que corre sobre el lenguaje de programación Ruby. Si no tienes ninguna experiencia con Ruby, existen algunos recursos gratis en la Internet para aprender Ruby, incluyendo:

- [Aprende a Programar con Ruby](#).
- [Mr. Neighborly's Humble Little Ruby Book](#).
- [Programming Ruby](#).
- [Why's \(Poignant\) Guide to Ruby](#).

Capítulo 2. ¿Qué es Rails?

Rails es un framework de desarrollo de aplicaciones web escrito en el lenguaje de programación Ruby. Está diseñado para hacer que la programación de aplicaciones web sea más fácil, haciendo supuestos sobre lo que cada desarrollador necesita para comenzar. Te permite escribir menos código realizando más que muchos otros lenguajes y frameworks. Además, expertos desarrolladores en Rails reportan que hace que el desarrollo de aplicaciones web sea más divertido.

Rails es un software dogmático. Éste asume que existe una forma "mejor" de hacer las cosas, y está diseñado para fomentar esa forma - y en algunos casos para desalentar alternativas.

Si aprendes "El Modo Rails" probablemente descubrirás un tremendo incremento en tu productividad. Si persistes trayendo viejos hábitos de otros lenguajes a tu desarrollo en Rails, e intentas usar patrones aprendidos en otros lugares, podrías tener una experiencia menos agradable.

La filosofía de Rails se basa en estos dos principios:

- DRY (del inglés, *"Don't Repeat Yourself"*) - sugiere que escribir el mismo código una y otra vez es una mala práctica.
- "Convención sobre Configuración" - significa que Rails hace algunas suposiciones sobre lo que quieres hacer y cómo vas a hacerlo, en lugar de requerir que especifiques cada pequeña cosa a través de un sin fin de archivos de configuración.

Capítulo 3. Creando un nuevo proyecto

La mejor forma de usar esta guía es seguir cada paso tal y como se muestra. No hemos ahorrado ninguna línea de código en las explicaciones, así que puedes seguirla literalmente paso a paso. [Ver el código completo](#).

Al seguir esta guía, vas a crear un proyecto Rails llamado `blog`, que es un motor de blogs muy sencillo. Antes de que puedas comenzar a crear la aplicación, asegúrate de tener Rails instalado.

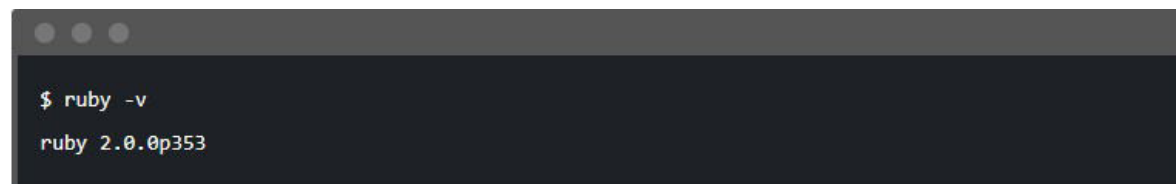
Truco

Los ejemplos que se muestran a continuación usan el carácter `$` para representar a la consola de comandos de los sistemas operativos basados en UNIX. Si estás usando Windows, la línea de comandos se verá como `c:\>`.

3.1. Instalando Rails

Lo primero que debes hacer es abrir una consola de comandos. En Mac OS X eso significa abrir la aplicación `Terminal.app`. En Windows pincha sobre el menú de inicio y elige la opción `Ejecutar` y teclea `cmd.exe`.

Las líneas que empiezan por `$` son los comandos que debes ejecutar en la consola. Lo primero es comprobar que tienes instalada alguna versión reciente de Ruby:

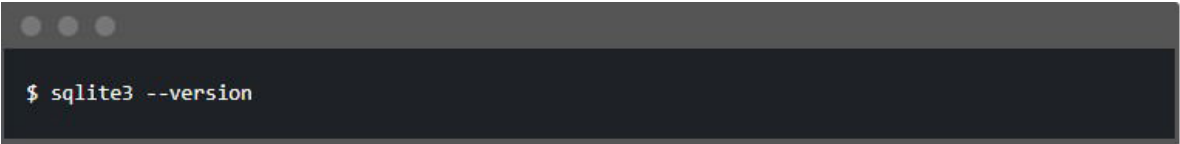


```
$ ruby -v
ruby 2.0.0p353
```

Truco

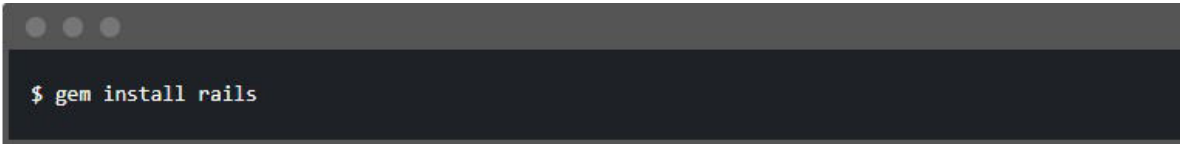
Existen varias herramientas para instalar Ruby en tu sistema operativo. Los usuarios de Windows pueden utilizar [el instalador de Rails](#), mientras que los usuarios de Mac OS X pueden usar [Tokaido](#).

Además, tu sistema debe contar con SQLite 3 instalado correctamente. Para comprobar si es así, ejecuta el siguiente comando:



```
$ sqlite3 --version
```

Una vez comprobado que tienes tanto Ruby como SQLite 3, para instalar Rails, usa el comando proporcionado por RubyGems `gem install`:



```
$ gem install rails
```

Para verificar que tu instalación esté correcta, deberías poder ejecutar lo siguiente:



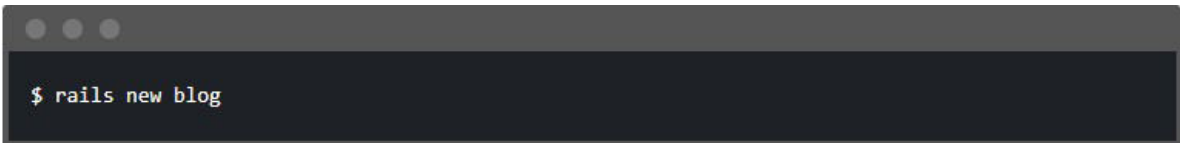
```
$ rails --version
```

Si dice algo como *"Rails 4.1.1"*, estás listo para continuar.

3.2. Creando la aplicación de Blog

Rails viene con un número de generadores que están diseñados para hacer tu ciclo de desarrollo más fácil. Uno de esos es el generador de nuevas aplicaciones, que crea la estructura base de una aplicación Rails, por lo que no tienes que escribirla por ti mismo.

Para usar este generador, abre la consola de comandos, navega hacia el directorio en donde tienes permiso para crear archivos, y ejecuta:



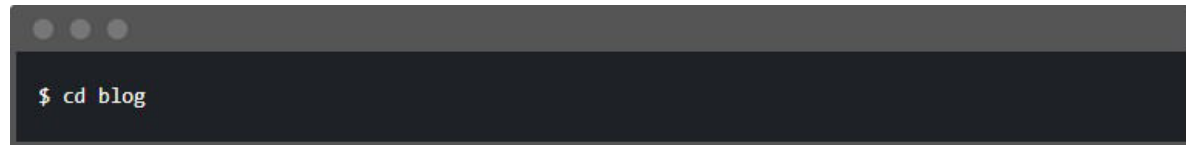
```
$ rails new blog
```

Esto creará una aplicación Rails llamada `Blog` en un directorio llamado `blog` e instalará las dependencias (gemas) que están mencionadas en el `Gemfile` usando el comando `bundle install`.

Truco

Puedes ver todas las opciones que el generador de nuevas aplicaciones provee, ejecutando `rails new -h`.

Después de crear la aplicación, entra a su directorio para continuar trabajando directamente en la aplicación:

A terminal window with a dark background. The prompt is a dollar sign followed by the command 'cd blog'.

El comando `rails new blog` que acabamos de ejecutar, creó una carpeta en tu directorio de trabajo llamado `blog`. El directorio `blog` tiene un número de archivos autogenerados y carpetas que conforman la estructura de una aplicación Rails.

La mayoría del trabajo en este tutorial se llevará a cabo en la carpeta `app/`, así que primero hagamos un repaso rápido al propósito de cada archivo y carpeta de Rails:

Archivo/Carpeta	Propósito
<code>app/</code>	Contiene los controllers, models, views, helpers, mailers y assets para tu aplicación. Te centrarás en esta carpeta por el resto de esta guía.
<code>config/</code>	Configura las reglas de ejecución de la aplicación, rutas, base de datos y más. Este tema es cubierto en mayor detalle en Configuring Rails Applications .
<code>config.ru</code>	Configuración Rack para servidores basados en Rack usados para iniciar la aplicación.
<code>db/</code>	Contiene el esquema actual de tu base de datos, así como las migraciones de la base de datos.
<code>doc/</code>	Documentación detallada de tu aplicación.

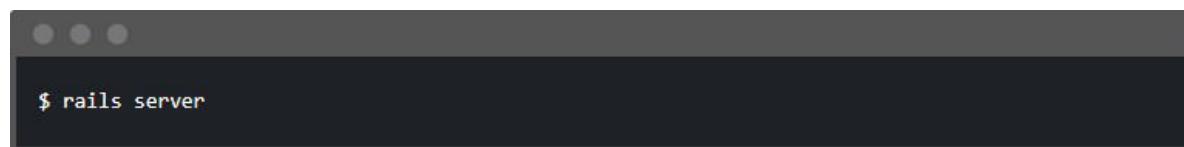
Archivo/Carpeta	Propósito
Gemfile Gemfile.lock	Estos archivos te permiten especificar qué dependencias de gemas son necesitadas para tu aplicación Rails. Estos archivos son usados por la gema Bundler, ver Sitio web de Bundler
lib/	Módulos extendidos para tu aplicación.
log/	Archivos de Log de tu aplicación.
public/	La única carpeta vista por el mundo tal como es. Contiene los archivos estáticos y assets compilados.
Rakefile	Este archivo localiza y carga tareas que pueden ser ejecutadas desde la línea de comandos. La lista de tareas son definidas a través de los componentes de Rails. En vez de cambiar el Rakefile, deberías agregar tus propias tareas, añadiendo archivos al directorio lib/tasks de tu aplicación.
README.rdoc	Este es un breve manual de instrucciones para tu aplicación. Deberías editar este archivo para comunicar a otros lo que tu aplicación hace, cómo configurarla y demás.
script/	Contiene el script de Rails que inicia tu aplicación y contiene otros scripts usados para deployar o correr tu aplicación.
test/	Pruebas unitarias, fixtures y otras pruebas. Éstos son cubiertos en Testing Rails Applications .
tmp/	Archivos temporales (como archivos de caché, PID y archivos de sesiones).
vendor/	Lugar para código de terceros. En una típica aplicación Rails, ésta incluye librerías y plugins.

Capítulo 4. Hola, Rails!

Para comenzar, vamos a mostrar algo de texto en la pantalla rápidamente. Para hacer ésto, necesitas tener tu servidor de aplicación Rails corriendo.

4.1. Iniciando el Servidor Web

En realidad ya tienes una aplicación Rails funcional, Para verla, necesitas iniciar un servidor web en tu máquina de desarrollo. Puedes hacerlo ejecutando:

A terminal window with a dark background and light text. The prompt is a dollar sign followed by the command 'rails server'.

Truco

Compilar CoffeeScript a JavaScript requiere de herramientas adicionales. Si no dispones de ellas, verás un error de tipo `execjs`. Normalmente Mac OS X y Windows ya incluyen estas utilidades. Rails agrega la gema `therubyracer` al `Gemfile` en una línea comentada para nuevas aplicaciones y puedes descomentarla si la necesitas.

`therubyrhino` es el *runtime de JavaScript* recomendado para usuarios de JRuby y es añadido por defecto al `Gemfile` en aplicaciones generadas bajo JRuby. Puedes investigar acerca de todos los runtimes soportados en [ExecJS](#).

Esto lanzará WEBrick, un servidor web incorporado en Ruby por defecto. Para ver tu aplicación en acción, abre tu navegador preferido y accede a <http://localhost:3000>. Deberías ver la página de información por defecto de Rails.

Truco

Para detener el servidor web, presiona `Ctrl+C` en la ventana de la línea de comandos donde se está ejecutando. En modo de desarrollo, Rails generalmente no requiere reiniciar el servidor web; los cambios realizados serán tomados automáticamente por el servidor.

La página "*Welcome Aboard*" es la primera prueba para una nueva aplicación Rails: Ésta asegura que tienes el software configurado correctamente para servir una página. También puedes hacer click en el link *About your application's enviroment* para ver un resumen del entorno de tu aplicación.

4.2. Hola Mundo

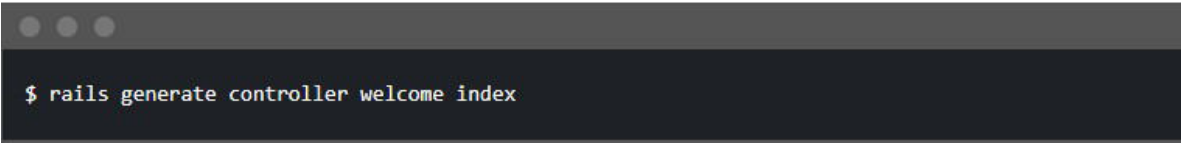
Para conseguir que Rails diga "*Hola*", necesitas crear como mínimo un *controlador* y una *vista*.

El propósito de un controlador es recibir las peticiones (*requests*) de la aplicación. El enrutamiento (*routing*) decide qué controlador recibe qué petición.

A menudo, hay más de una ruta para cada controlador, y diferentes rutas pueden ser servidas por diferentes acciones (*actions*). El propósito de cada acción es obtener información para pasarla después a la vista.

El propósito de una vista es mostrar la información en un formato legible para los humanos. Una distinción importante que hacer es que es el *controlador*, y no la vista, donde la información es recolectada. La vista sólo debería mostrar la información. Por defecto, las plantillas de las vistas están escritas en un lenguaje llamado *ERB* (del inglés, *Embedded Ruby*), que se procesa automáticamente para cada petición servida por Rails.

Para crear un nuevo controlador, necesitas ejecutar el generador de controladores y decirle que quieres un controlador llamado por ejemplo `welcome` con una acción llamada `index`. Para ello, ejecuta lo siguiente:



```
$ rails generate controller welcome index
```

Rails creará una serie de archivos y añadirá una ruta por ti.

```
create app/controllers/welcome_controller.rb
route get "welcome/index"
invoke erb
create app/views/welcome
create app/views/welcome/index.html.erb
invoke test_unit
create test/functional/welcome_controller_test.rb
invoke helper
create app/helpers/welcome_helper.rb
invoke test_unit
create test/unit/helpers/welcome_helper_test.rb
invoke assets
invoke coffee
create app/assets/javascripts/welcome.js.coffee
invoke scss
create app/assets/stylesheets/welcome.css.scss
```

Los archivos más importantes de éstos son por supuesto el controlador, que se encuentra en `app/controllers/welcome_controller.rb` y la vista, que se encuentra en `app/views/welcome/index.html.erb`.

Abre el archivo `app/views/welcome/index.html.erb` en tu editor de texto y editalo para que contenga sólo esta línea de código:

```
<h1>Hello, Rails!</h1>
```

4.3. Estableciendo la página principal

Ahora que hemos hecho el *controlador* y la *vista*, necesitamos decirle a Rails cuándo queremos que se muestre Hello Rails. En nuestro caso, deseamos mostrarlo al acceder a la portada de nuestro sitio, <http://localhost:3000>. Por el momento, sin embargo, la página "Welcome Aboard" está ocupando ese lugar.

Para arreglarlo, borra el archivo `index.html` ubicado dentro de la carpeta `public` de la aplicación.

Necesitas hacer esto debido a que Rails servirá preferentemente cualquier archivo estático en la carpeta `public` que coincida con una ruta de la aplicación.

El archivo `index.html` es especial, ya que se sirve si llega una petición a la ruta raíz de la aplicación (en `http://localhost:3000`). Si haces otra petición (por ejemplo `http://localhost:3000/welcome`), se serviría el archivo estático `public/welcome.html`, pero sólo si existiera.

A continuación, tienes que indicarle a Rails donde está ubicada tu página principal. Para ello, abre el archivo `config/routes` en tu editor:

```
Blog::Application.routes.draw do
```

```
  get "welcome/index"
```

```
  # The priority is based upon order of creation:
```

```
  # first created -> highest priority.
```

```
  # ...
```

```
  # You can have the root of your site routed with "root"
```

```
  # just remember to delete public/index.html.
```

```
  # root :to => "welcome#index"
```

Éste es el *archivo de enrutamiento* de tu aplicación, que mantiene entradas con un DSL especial (*domain-specific language*) que le dicen a Rails cómo conectar peticiones entrantes a *controladores* y *acciones*.

Este archivo contiene muchas rutas de ejemplo en líneas comentadas, y una de ellas en realidad muestra como conectar la raíz de tu sitio a un *controlador* y *acción* específicos. Encuentra la línea iniciando con `root :to` y descoméntala. Debería verse como lo siguiente:

```
root :to => "welcome#index"
```

El `root :to => "welcome#index"` le indica a Rails que debe asociar las peticiones de la raíz de la aplicación a la *acción* `index` del *controlador* `welcome` y `get "welcome/index"` le indica a Rails que asocie peticiones de <http://localhost:3000/welcome/index> a la *acción* `index` del *controlador* `welcome`. Este fue creado al inicio cuando ejecutaste el generador del *controlador* (`rails generate controller welcome index`).

Si accedes a la dirección <http://localhost:3000> en tu navegador, verás el mensaje Hello, Rails! que colocaste dentro de la

vista `app/views/welcome/index.html.erb`, indicando que esta nueva ruta está en realidad pasando a la acción `index` del controlador `Welcome` y está renderizando la vista correctamente.

Capítulo 5. Creando un nuevo post

Ahora que has visto cómo crear un controlador, una acción y una vista, vamos a crear algo un poco más complejo.

En la aplicación de blog, ahora vas a crear un nuevo recurso (*resource*). Un recurso es el término usado para una colección de objetos similares, como artículos, personas o animales. Puedes crear, leer, actualizar y eliminar objetos para un recurso y estas operaciones son referidas como operaciones *CRUD* (del inglés, *Create, Read, Update, Destroy*).

Rails incluye un método llamado `resources` que puedes utilizar para declarar un recurso REST estándar. Este es el aspecto que debería tener el archivo `config/routes.rb` después de haber declarado el recurso `article`:

```
Blog::Application.routes.draw do
```

```
  resources :articles
```

```
  root 'welcome#index'
```

```
end
```

Si ejecutas el comando `rake routes`, verás que se han definido varias rutas para todas las acciones estándar de las aplicaciones RESTful. Más adelante se explicará el significado de cada columna. Por ahora sólo fíjate en que Rails ha inferido que la palabra en singular del recurso `articles` es `article`, con lo que el resultado es el siguiente:

```
$ bin/rake routes
```

Prefix	Verb	URI Pattern	Controller#Action
articles	GET	/articles(.:format)	articles#index
	POST	/articles(.:format)	articles#create
new_article	GET	/articles/new(.:format)	articles#new
edit_article	GET	/articles/:id/edit(.:format)	articles#edit
article	GET	/articles/:id(.:format)	articles#show
	PATCH	/articles/:id(.:format)	articles#update
	PUT	/articles/:id(.:format)	articles#update
	DELETE	/articles/:id(.:format)	articles#destroy
root	GET	/	welcome#index

En las próximas secciones añadirás la funcionalidad de crear artículos en la aplicación y también podrás visualizarlos. Estas dos acciones son, respectivamente, las letras *C* y *R* del conjunto de acciones CRUD. El formulario para crear los artículos tendrá este aspecto:

New Article

Title

Text

Save Article

Por el momento será un formulario muy simple. Pero tranquilo porque después ya le aplicaremos estilos para mejorarlo.

5.1. Primeros pasos

La primera cosa que necesitaremos para crear un nuevo artículo, es crear una sección especial para esto en nuestra aplicación. Un buen lugar podría ser `/posts/new`. Si tratas de navegar a esa dirección ahora - visitando <http://localhost:3000/articles/new> - Rails mostrará el siguiente mensaje de error:

Routing Error

uninitialized constant ArticlesController

Este error se produce porque la ruta necesita tener asociado un controlador para poder servir las peticiones. Así que la solución es tan sencilla como crear un controlador llamado `ArticlesController`. Si no quieres crearlo a mano, ejecuta el siguiente comando:

```
$ bin/rails g controller articles
```

Si ahora abres el archivo `app/controllers/articles_controller.rb` que se acaba de generar, verás el siguiente controlador vacío:

```
class ArticlesController < ApplicationController  
  
end
```

Un controlador es simplemente una clase que hereda de `ApplicationController`. Dentro de esta clase defines los métodos que se convertirán en las acciones del controlador. Estas acciones son las que se encargan de las operaciones CRUD sobre los artículos de la aplicación.

Nota

Ruby permite definir métodos públicos, privados y protegidos, pero las acciones de los controladores solo pueden ser métodos públicos.

Si ahora refrescas la página <http://localhost:3000/articles/new>, verás de nuevo un error, pero diferente al anterior:

Unknown action

The action 'new' could not be found for ArticlesController

Este error significa que Rails no puede encontrar la acción `new` dentro del controlador `ArticlesController` que acabas de generar. El motivo es que cuando se genera un controlador automáticamente, su contenido está vacío a menos que le indiques qué acciones quieres incluir.

Para añadir una acción a mano en un controlador, lo único que necesitas es crear un nuevo método en el controlador. Abre el archivo `app/controllers/articles_controller.rb` y dentro de la clase `ArticlesController` define un nuevo método de esta manera:

```
def new
```

```
end
```

Refresca de nuevo la página <http://localhost:3000/articles/new> y verás que a pesar de haber añadido el método `new`, sigues viendo un mensaje de error:

Template is missing

Missing template articles/new, application/new with :builder. :coffee}. Searched in: *

La causa de este error es que Rails espera que las acciones de los controladores tengan asociadas unas *vistas* para mostrar su información. Cuando no existe la vista, Rails muestra el siguiente mensaje de error.

```
Missing template articles/new, application/new with
```



```
{:locale[:en], :formats[:html], :handlers[:erb, :builder, :coffee]}.
```

```
Searched in: * "/path/to/blog/app/views"
```

Vaya rollo de texto, ¿verdad? Repasemos rápidamente cada parte de ese texto.

La primera parte identifica la plantilla que está faltando. En este caso, es la plantilla `articles/new`. Rails primero buscará esta plantilla. Si no la encuentra, luego tratará de cargar la plantilla llamada `application/new`. El motivo es que tu controlador `ArticlesController` hereda del controlador base `ApplicationController`.

La siguiente parte del mensaje contiene un mapa (en inglés, *hash*). La clave `:locale` en este mapa simplemente indica el idioma para el que se debe obtener la plantilla. Por defecto, está configurado en inglés (código `en`).

La siguiente clave, `:formats` especifica el formato de la plantilla que será servido en la respuesta. El formato por defecto es `:html`, por lo que Rails busca una plantilla HTML. La última clave, `:handlers`, nos dice qué *manejador de plantilla* (en inglés, *template handler*) se puede usar para renderizar la plantilla. `:erb` es el manejador más usado para plantillas HTML, `:builder` se usa para plantillas XML, y `:coffee` usa CoffeeScript para construir plantillas JavaScript.

La última parte del mensaje de error indica los lugares en los que Rails ha buscado las plantillas. En una aplicación Rails como esta las plantillas se guardan en un único directorio. Pero en aplicaciones más complejas, las plantillas podrían encontrarse repartidas en varias localizaciones.

La plantilla más simple que funcionaría en nuestro caso sería una localizada en `app/views/articles/new.html.erb`. La extensión de este nombre de archivo es muy importante: la primera extensión define el *formato* de la plantilla, y la segunda extensión el *manejador de plantilla* que se usará.

Rails tratará de encontrar una plantilla llamada `articles/new` en el directorio `app/views`. El formato para esta plantilla sólo puede ser `html` y el manejador debería ser `erb`, `builder` o `coffee`. Ya que quieres crear un nuevo formulario HTML, tendrás que usar el lenguaje ERB. Por lo tanto el archivo debería llamarse `articles/new.html.erb` y debe encontrarse dentro de la carpeta `app/views` de la aplicación.

Ahora crea un nuevo archivo en `app/views/articles/new.html.erb` y escribe el siguiente contenido en él:

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

Cuando recargues <http://localhost:3000/posts/new>, verás que la página tiene un título. La ruta, el controlador, la acción y la vista ahora están funcionando todos de manera conjunta y por eso ya no se muestra ningún error. Ahora es el momento de crear un formulario para crear artículos.

5.2. El primer formulario

Para crear un formulario dentro de esta plantilla, se utiliza un *constructor de formularios* (en inglés, "*form builder*"). El principal constructor de formularios de Rails incluye un *helper* llamado `form_for`. Para hacer uso de este método, añade el siguiente código a `app/views/posts/new.html.erb`:

```
<%= form_for :post do |f| %>

  <p>

    <%= f.label :title %><br>

    <%= f.text_field :title %>

  </p>

  <p>

    <%= f.label :text %><br>

    <%= f.text_area :text %>

  </p>

  <p>

    <%= f.submit %>

  </p>

<% end %>
```

Si recargas la página, ahora verás el mismo formulario que en el ejemplo. ¡Crear formularios en Rails es así de sencillo!

Cuando llamas a `form_for`, le pasas un objeto identificador para este formulario, que en este caso, es el símbolo `:article`. Esto le indica al método `form_for` para quién es este formulario.

Dentro del bloque para este método, el objeto `FormBuilder` - representado por `f` - se usa para construir dos títulos y dos cajas de texto, una para el título del artículo y otra para su contenido. Finalmente, la llamada al método `submit` en el objeto `f` crea un botón para enviar el formulario.

Sin embargo, existe un problema con este formulario. Si ves el código HTML generado en la página, verás que el atributo `action` de el formulario apunta a `/articles/new`. Ésto es un problema porque esta ruta va a la misma página donde te encuentras en este momento, cuando en realidad la ruta sólo debería usarse para mostrar el formulario que crea nuevos artículos.

El formulario necesita una URL distinta para poder dirigirse hacia algún otro lugar. Conseguirlo es bastante sencillo, ya que basta utilizar la opción `:url` del método `form_for`. En Rails la acción que se usa normalmente para crear nuevos elementos se llama `create` (en español, "crear"), por lo que el formulario debería apuntar hacia esa acción.

Edita la línea de `form_for` de la vista `app/views/posts/new.html.erb` para que se vea así:

```
<%= form_for :article, url: articles_path do |f| %>
```

En este ejemplo se pasa el *helper* `articles_path` a la opción `:url`. Para comprender qué es lo que hace Rails con este objeto, recuerda cómo era el resultado de ejecutar el comando `rake routes`:

```
$ bin/rake routes

      Prefix Verb   URI Pattern               Controller#Action
articles GET    /articles(.:format)      articles#index
          POST    /articles(.:format)      articles#create
new_article GET    /articles/new(.:format)  articles#new
edit_article GET    /articles/:id/edit(.:format) articles#edit
article GET    /articles/:id(.:format)  articles#show
          PATCH   /articles/:id(.:format)  articles#update
          PUT     /articles/:id(.:format)  articles#update
          DELETE  /articles/:id(.:format)  articles#destroy
root GET    /                        welcome#index
```

El *helper* `articles_path` le dice a Rails que apunte el formulario a la URI asociada con el prefijo `articles`, por lo que el formulario enviará por defecto peticiones POST a esa ruta. Como ves, esta ruta está asociada con la acción `create` del controlador actual, `ArticlesController`.

Con el formulario y su ruta asociada definidas, ahora ya puedes rellenar el formulario y pinchar en el botón de enviar para empezar el proceso de creación de un nuevo artículo. No obstante, cuando envíes el formulario verás el siguiente mensaje de error que ya debería resultarte familiar:



Para solucionarlo debemos crear la acción `create` en `ArticlesController`.

5.3. Creando artículos

Para hacer que el error "*Unknown action*" desaparezca, definí una acción `create` dentro de la clase `ArticlesController` en `app/controllers/articles_controller.rb`, debajo de la acción `new`:

```
class PostsController < ApplicationController

  def new

  end

  def create

  end

end
```

Si ahora reenvías el formulario, verás otro error común: falta la plantilla. Por el momento vamos a ignorar este error, ya que la acción `create` sí que está al menos guardando el nuevo artículo en la base de datos.

Cuando se envía un formulario, los campos del formulario se envían a Rails como *parámetros*. Estos parámetros se pueden obtener dentro de las acciones del controlador, ya que normalmente tu código los necesita para completar sus tareas. Para ver cómo son parámetros, cambiar la acción `create` a esto:

```
def create

  render plain: params[:article].inspect

end
```

Al método `render` se le pasa un *hash* muy simple con la clave `text` y el valor `params[:post].inspect`. El método `params` es el objeto que representa a los parámetros (o campos) que vienen desde el formulario. El método `params` devuelve un objeto `ActiveSupport::HashWithIndifferentAccess`, que te permite acceder a las claves del *hash* usando tanto cadenas de texto como símbolos. En nuestro caso los únicos parámetros que importan son los que vienen del formulario.

Si reenvías el formulario una vez más, ya no obtendrás el error que indica que falta la plantilla. En su lugar verás algo como lo siguiente:

```
{"title"=>"First article!", "text"=>"This is my first article."}
```

Ahora esta acción muestra los parámetros que se envían mediante el formulario que crea nuevos artículo. En cualquier caso, de momento esto no nos sirve para nada. Aunque es cierto que puedes ver el contenido de los parámetros, no estamos haciendo nada con ellos, por lo que su valor se pierde.

5.4. Creando el modelo Article

Los modelos en Rails se escriben en singular y los nombres de sus tablas asociadas se convierten automáticamente en plural. Rails incluye un generador para crear modelos que lo utilizan la mayoría de programadores Rails para crear al menos el *esqueleto* del modelo. Para crear el nuevo modelo `Article` ejecuta el siguiente comando en la consola:

```
$ bin/rails generate model Article title:string text:text
```

Con ese comando le decimos a Rails que queremos un modelo llamado `Article`, que contenga un atributo `title` de tipo cadena de texto y un atributo `text` de tipo texto largo. Esos atributos se añaden automáticamente a la tabla `articles` en la base de datos y se asocian al modelo `Article`.

El resultado de ejecutar este comando es que Rails ha creado varios archivos. Por el momento sólo nos interesan `app/models/article.rb` y `db/migrate/20140120191729_create_articles.rb` (en tu caso el nombre de este archivo será ligeramente diferente). Este último se encarga de crear la estructura de la base de datos, que es lo próximo que vamos a analizar.

Truco El *Active Record* es lo suficientemente *inteligente* para asociar automáticamente el nombre de las columnas a los atributos del modelo, lo que significa que no tienes que declarar los atributos dentro de los modelos de Rails, ya que *Active Record* lo hará por tí automáticamente.

5.5. Migraciones

Como hemos visto, el comando `rails generate model` crea un archivo de migración de base de datos en el directorio `db/migrate`. Las migraciones son clases de Ruby que están diseñadas para simplificar la creación y modificación de las tablas de la base de datos.

Rails usa comandos `rake` para ejecutar migraciones, y es posible deshacer una migración después de que ha sido aplicada a la base de datos. Los archivos de

migraciones incluyen un sello de tiempo con la fecha y hora para que siempre sean procesadas en el mismo orden que fueron creadas.

Si miras el contenido del archivo `db/migrate/20140120191729_create_articles.rb` (recuerda que en tu caso el nombre de este archivo será ligeramente diferente), esto es lo que encontrarás:

```
class CreateArticles < ActiveRecord::Migration

  def change

    create_table :articles do |t|

      t.string :title

      t.text :text

      t.timestamps

    end

  end

end
```

La migración de arriba crea un método llamado `change` que se invoca cuando ejecutas la migración. La acción definida en este método es reversible, lo cual significa que Rails sabe cómo deshacer los cambios realizados en esta migración en caso que necesites hacerlo más adelante.

Cuando ejecutas esta migración se crea una tabla `articles` con una columna tipo `string` y otra columna tipo `text`. También crea dos campos de tipo `timestamp` para que Rails pueda guardar la fecha y hora en la que los artículos se crean y se modifican.

Nota

Puedes encontrar más información sobre las migraciones en el artículo [Rails Database Migrations](#)

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

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

Rails ejecutará este comando de migración y te indicará que ha creado la tabla `Articles`.

```
== CreateArticles: migrating =====
-- create_table(:articles)
   -> 0.0019s
== CreateArticles: migrated (0.0020s) =====
```

Nota

Como por defecto siempre estás trabajando en el entorno de desarrollo, este comando se aplica sobre la base de datos definida en la sección `development` del archivo `config/database.yml`. Si quieres ejecutar migraciones en otro entorno, por ejemplo en producción, debes indicarlo en forma explícita cuando invoques el comando: `rake db:migrate RAILS_ENV=production`.

5.6. Guardando datos en el controlador

En `ArticlesController` debemos cambiar ahora la clase `create` para que utilice el nuevo modelo `Article` al guardar la información en la base de datos. Abre el archivo `app/controllers/articles_controller.rb` y modifica la acción `create` para que tenga el siguiente contenido:

```
def create

  @article = Article.new(params[:article])

  @article.save

  redirect_to @article

end
```

Lo que está sucediendo ahora es que cada modelo Rails se inicializa con sus respectivos atributos, que se asocian automáticamente con las columnas

correspondientes de la base de datos. La primera línea del código anterior hace exactamente eso (recuerda que `params[:article]` contiene los atributos que nos interesan). Después, `@article.save` se encarga de guardar el modelo en la base de datos. Por último, redirigimos al usuario a la acción `show`, que definiremos más adelante.

Truco

Como veremos después, `@article.save` devuelve una variable *booleana* que indica si el modelo fue guardado o no.

Si recargas de nuevo la página <http://localhost:3000/articles/new> serás casi capaz de crear un nuevo artículo. Si lo pruebas, verás que se muestra este mensaje de error:

ActiveModel::ForbiddenAttributesError

ActiveModel::ForbiddenAttributesError

Extracted source (around line #6):

```
4
5   def create
6     @article = Article.new(params[:article])
```

Rails incluye varias opciones para ayudarte a desarrollar aplicaciones seguras y ahora mismo estás viendo en directo una de ellas. Esta opción se llama `strong_parameters` y obliga a decirle a Rails exactamente qué parámetros estamos dispuestos a aceptar en nuestros controladores.

Como en este caso sólo nos interesan los atributos `title` y `text`, debes añadir el siguiente método `article_params` y cambiar el código de la acción `create`:

```
def create

  @article = Article.new(article_params)

  @article.save

  redirect_to @article

end
```

```
private

  def article_params

    params.require(:article).permit(:title, :text)

  end
```

El método `permit` es lo que nos permite aceptar tanto el `title` como el `text` en la acción.

Nota

Observa que `def article_params` es un método privado. Esta estrategia impide que un atacante pueda establecer los atributos del modelo manipulando el *hash* que se pasa al modelo. Para más información al respecto, consulta este [artículo sobre los Strong Parameters](#).

5.7. Mostrando artículos

Si ahora vuelves a enviar el formulario, Rails mostrará un mensaje de error indicando que no encuentra la acción `show`. Como esto no es muy útil, vamos a crear la acción `show` antes de continuar.

Como viste en la salida del comando `rake routes`, la ruta de la acción `show` tiene este aspecto:



```
$ bin/rake routes
```

Prefix Verb	URI Pattern	Controller#Action
article GET	/articles/:id(.:format)	articles#show
...		

La sintaxis `:id` de la ruta indica a Rails que esta ruta espera un parámetro llamado `:id`, que en este caso será el atributo `id` del artículo que se quiere ver. Como hicimos anteriormente, es necesario añadir una nueva acción en el archivo `app/controllers/articles_controller.rb` y también hay que crear su vista correspondiente.

```
def show

  @article = Article.find(params[:id])
```

```
end
```

Un par de cosas que debes tener en cuenta: usamos `Article.find` para encontrar el artículo que nos interesa, pasándole el valor `params[:id]` para obtener el parámetro `:id` directamente de la petición. También usamos una variable de instancia (con el prefijo `@`) para guardar una referencia al objeto del artículo. Hacemos eso porque Rails pasa automáticamente todas las variables de instancia a la vista.

Ahora, crea un nuevo archivo `app/view/articles/show.html.erb` con el siguiente contenido:

```
<p>

  <strong>Title:</strong>

  <%= @article.title %>

</p>
```

```
<p>

  <strong>Text:</strong>

  <%= @article.text %>

</p>
```

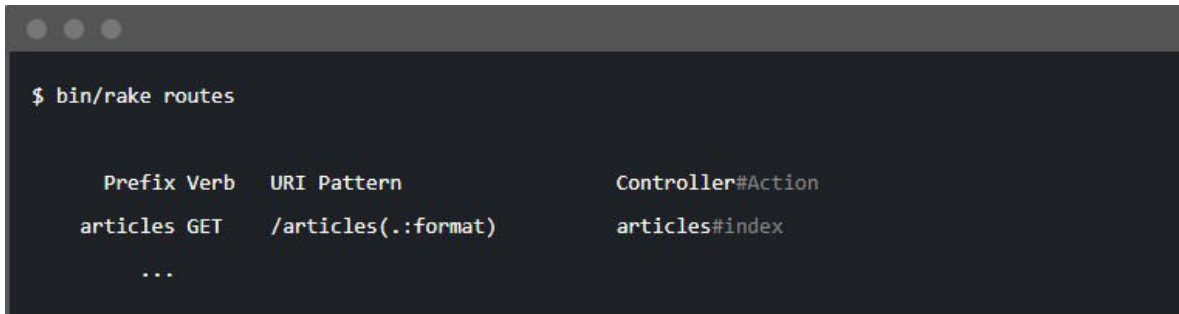
Con estos cambios, ya deberíamos poder crear nuevos artículos. Para ello, visita <http://localhost:3000/posts/new> y pruébalo.

Title: Rails is Awesome!

Text: It really is.

5.8. Listando todos los artículos

Todavía no tenemos una forma de listar todos los artículos disponibles, por lo que vamos a hacerlo ahora mismo. La ruta que corresponde a esta acción según el comando `rake routes` es:



```
$ bin/rake routes
```

Prefix	Verb	URI Pattern	Controller#Action
articles	GET	/articles(.:format)	articles#index
...			

Y la acción `index` correspondiente para esta ruta dentro de `ActionController` del archivo `app/controllers/articles_controller.rb` debería ser:

```
def index

  @articles = Article.all

end
```

Y por último, añade la vista para esta acción en la plantilla localizada en `app/views/articles/index.html.erb`:

```
<h1>Listing articles</h1>

<table>

  <tr>

    <th>Title</th>

    <th>Text</th>

  </tr>

  <% @articles.each do |article| %>

    <tr>

      <td><%= article.title %></td>
```

```

      <td><%= article.text %></td>

    </tr>

  <% end %>

</table>

```

Visita ahora la página <http://localhost:3000/articles> y verás el listado de todos los artículos que has creado.

5.9. Añadiendo enlaces

En estos momentos puedes crear, mostrar y listar artículos. Pero todas estas acciones son independientes y no están conectadas entre sí. Por ello ahora vamos a añadir algunos enlaces que nos permitan navegar entre las páginas.

Abre `app/views/welcome/index.html.erb` y modifícalo como se indica a continuación:

```

<h1>Hello, Rails!</h1>

<%= link_to 'My Blog', controller: 'articles' %>

```

El método `link_to` es uno de los *helpers* que incluye Rails para las vistas. Este método crea un enlace utilizando el texto y la ruta indicados, que en este caso será la ruta `articles`.

Vamos a agregar enlaces a las otras vistas, empezando por añadir el enlace "New Article" en la vista `app/views/articles/index.html.erb` justo por encima de la etiqueta `<table>`:

```

<%= link_to 'New article', new_article_path %>

```

Este enlace te permite acceder directamente al formulario para crear nuevos artículos. Añade también un enlace en `app/views/articles/new.html.erb`, justo debajo del formulario, para volver a la portada del sitio:

```

<%= form_for :article, url: articles_path do |f| %>

  ...

<% end %>

```

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

Finalmente, añade otro enlace a la plantilla `app/views/articles/show.html.erb` para regresar a la acción `index`, de manera que la persona que está viendo un artículo pueda volver fácilmente a la portada del sitio.

```
<p>
```

```
  <strong>Title:</strong>
```

```
  <%= @article.title %>
```

```
</p>
```

```
<p>
```

```
  <strong>Text:</strong>
```

```
  <%= @article.text %>
```

```
</p>
```

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

Truco

Si quieres crear un enlace a una acción dentro del mismo controlador, no es necesario que especifiques la opción `:controller`, ya que Rails utiliza por defecto el controlador actual.

Truco

En el entorno de desarrollo, que es el que se utiliza por defecto, Rails recarga la aplicación entera con cada petición, por lo que no es necesario parar y reiniciar el servidor web con cada cambio.

5.10. Añadiendo validaciones

Ahora mismo, el archivo `app/models/article.rb` del modelo es tan simple como:

```
class Article < ActiveRecord::Base
```

```
end
```

Aunque el archivo está vacío, fíjate que la clase hereda de `ActiveRecord::Base`. El *Active Record* añade de esta manera muchas funcionalidades a tus modelos de Rails. Entre otras, incluye un CRUD básico, validación de datos, búsqueda de información y la posibilidad de relacionar modelos entre sí.

Rails incluye por ejemplo métodos que te permiten validar la información que envías a los modelos. Para definir esta validación, abre el archivo `app/models/article.rb` y añade lo siguiente:

```
class Article < ActiveRecord::Base
```

```
  validates :title, presence: true,
```

```
          length: { minimum: 5 }
```

```
end
```

Estos cambios aseguran que todos los artículos tengan un título de al menos cinco caracteres de longitud. Rails puede validar diferentes características de los modelos, como si incluyen una determinada columna, si su valor es único en la base de datos, su formato, y si existen o no objetos relacionados. Las validaciones se explican con detalle en el artículo [Active Record Validations](#).

Ahora que ya has configurado la validación, cuando invoques el método `@article.save` en un artículo inválido, el resultado será `false`. No obstante, si abres el archivo `app/controllers/articles_controller.rb`, verás que no verificamos el resultado de la llamada a `@article.save` dentro de la acción `create`. Si el método `@article.save` devuelve `false`, deberíamos mostrar de nuevo el formulario al usuario mostrándole los errores que contiene. Para ello, cambia las acciones `new` y `create` en el archivo `app/controllers/articles_controller.rb`:

```
def new
```

```
  @article = Article.new
```

```
end
```

```
def create
```

```
  @article = Article.new(article_params)
```

```

    if @article.save

      redirect_to @article

    else

      render 'new'

    end

  end

end

private

def article_params

  params.require(:article).permit(:title, :text)

end

```

La acción `new` ahora crea una variable de instancia llamada `@article` y el motivo por el que lo hace lo comprenderás muy pronto.

Observa que en la acción `create` utilizamos `render` en vez de `redirect_to` cuando el método `save` devuelve `false`. Se utiliza el método `render` para que el objeto `@article` se pase cuando se renderiza la plantilla. Esta renderización se realiza en la misma petición en la que se envió el formulario, mientras que `redirect_to` le dice al navegador que realice otra petición.

Si recargas la página <http://localhost:3000/articles/new> y tratas de guardar un artículo sin título, Rails te mostrará el mismo formulario vacío, lo cual no es muy útil. Lo que debemos hacer es mostrar al usuario qué errores se han producido. Para ello, modifica el archivo `app/views/articles/new.html.erb` para que compruebe si existen errores:

```

<%= form_for :article, url: articles_path do |f| %>

  <% if @article.errors.any? %>

    <div id="error_explanation">

      <h2><%= pluralize(@article.errors.count, "error") %> prohibited
    
```



```

    this article from being saved:</h2>

    <ul>

    <% @article.errors.full_messages.each do |msg| %>

        <li><%= msg %></li>

    <% end %>

    </ul>

</div>

<% end %>

<p>

    <%= f.label :title %><br>

    <%= f.text_field :title %>

</p>

<p>

    <%= f.label :text %><br>

    <%= f.text_area :text %>

</p>

<p>

    <%= f.submit %>

</p>

<% end %>

<%= link_to 'Back', articles_path %>

```

En este código están sucediendo varias cosas. Primero comprobamos si existen errores con el método `@article.errors.any?`. Si existen, mostramos una lista con todos los errores obtenidos mediante `@article.errors.full_messages`.

El *helper* `pluralize` es uno de los *helpers* incluidos en Rails y que acepta como argumentos un número y una cadena de texto. Si el número pasado es mayor que 1, la cadena de texto se convierte automáticamente en plural..

El motivo por el que añadimos `@article = Article.new` en el controlador `ArticlesController` es porque si no, `@article` tendría un valor `nil` en la vista. Así que al ejecutar `@article.errors.any?` se mostraría un error.

Rails encierra automáticamente con un `<div class="field_with_errors">` a los campos de formulario que contienen errores. Así que sólo tienes que definir los estilos CSS correspondientes para hacer que los mensajes de error destaquen lo suficiente.

Ahora sí verás un mensaje de error cuando trates de guardar un artículo sin título en la página <http://localhost:3000/articles/new>.

New Article

2 errors prohibited this article from being saved:

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

5.11. Actualizando artículos

Hasta ahora nos hemos ocupado de la parte *CR* del acrónimo CRUD. En esta sección nos ocuparemos de la parte *U*, es decir, de actualizar artículos. El primer paso consiste en añadir la acción `edit` a `ArticlesController`.

```
def edit

  @article = Article.find(params[:id])

end
```

La vista contendrá un formulario similar al que hemos usado para crear nuevos artículos. Para ello, crea el archivo `app/views/articles/edit.html.erb` con el siguiente contenido:

```
<h1>Editing article</h1>
```

```
<%= form_for :article, url: article_path(@article), method: :patch do |f| %>

  <% if @article.errors.any? %>

    <div id="error_explanation">

      <h2><%= pluralize(@article.errors.count, "error") %> prohibited

        this article from being saved:</h2>

      <ul>

        <% @article.errors.full_messages.each do |msg| %>

          <li><%= msg %></li>

        <% end %>

      </ul>

    </div>

    <% end %>

    <p>

      <%= f.label :title %><br>

      <%= f.text_field :title %>

    </p>

    <p>

      <%= f.label :text %><br>

      <%= f.text_area :text %>

    </p>

    <p>

      <%= f.submit %>

    </p>

  <% end %>

</form_for>
```

```
</p>
```

```
<% end %>
```

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

En esta ocasión el formulario apunta a la acción `update`, que todavía no hemos creado. La opción `:method => :patch` le dice a Rails que queremos enviar este formulario mediante el método PATCH de HTTP que, según el protocolo REST, es el que se debe utilizar al actualizar recursos.

El primer parámetro de `form_for` puede ser un objeto, por ejemplo `@article`, que se utiliza para rellenar los campos del formulario. Si pasas un símbolo (ejemplo `:article`) cuyo nombre sea idéntico al de una variable de instancia (`@article`) el funcionamiento es el mismo. Esto es precisamente lo que está pasando en este ejemplo. Consulta la [documentación de form_for](#) para conocer más detalles.

A continuación, crea la acción `update` en el archivo `app/controllers/articles_controller.rb`:

```
def update
```

```
  @article = Article.find(params[:id])
```

```
  if @article.update(article_params)
```

```
    redirect_to @article
```

```
  else
```

```
    render 'edit'
```

```
  end
```

```
end
```

```
private
```

```
  def article_params
```

```
params.require(:article).permit(:title, :text)

end
```

El nuevo método `update` se usa al actualizar un registro que ya existe, y acepta como argumento un *hash* que contiene los atributos que deseas actualizar. Como hicimos anteriormente, si hay un error actualizando el artículo queremos volver a mostrar el formulario al usuario. Para ello se reutiliza el método `article_params` definido anteriormente para la acción `create`.

Truco

No es necesario pasar todos los atributos a `update`. Por ejemplo, si ejecutaras `@article.update(title: 'A new title')` Rails solo actualizaría el atributo `title` sin tocar los otros atributos.

Por último, queremos mostrar un enlace a la acción `edit` en la lista de todos los artículos. Así que modifica la vista `app/views/articles/index.html.erb` para añadir un nuevo enlace:

```
<table>

  <tr>

    <th>Title</th>

    <th>Text</th>

    <th colspan="2"></th>

  </tr>

  <% @articles.each do |article| %>

    <tr>

      <td><%= article.title %></td>

      <td><%= article.text %></td>

      <td><%= link_to 'Show', article_path(article) %></td>

      <td><%= link_to 'Edit', edit_article_path(article) %></td>

    </tr>
```

```
<% end %>
```

```
</table>
```

Añade también un enlace en la vista `app/views/articles/show.html.erb`, de manera que haya un enlace `Edit` en la página de cada artículo. Añade lo siguiente al final de la plantilla:

...

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

```
| <%= link_to 'Edit', edit_article_path(@article) %>
```

Y este es el aspecto que tiene la aplicación por el momento:

Listing articles

[New article](#)

Title	Text	
Welcome To Rails	Example	Show Edit
Rails is awesome!	It really is.	Show Edit

5.12. Usando parciales para eliminar las duplicidades en las vistas

La página `edit` se parece mucho a la página `new` porque utilizan el mismo código para mostrar el formulario. Así que vamos a eliminar estas duplicidades mediante los *parciales*. Por convención un *partial* es un archivo de plantilla cuyo nombre empieza por un guión bajo y cuyo contenido se reutiliza en varias plantillas diferentes.

Nota

Puedes aprender más sobre los *parciales* en la guía [Layouts and Rendering in Rails](#).

Crea un nuevo archivo llamado `app/views/articles/_form.html.erb` y que tenga el siguiente contenido:

```
<%= form_for @article do |f| %>

  <% if @article.errors.any? %>

    <div id="error_explanation">

      <h2><%= pluralize(@article.errors.count, "error") %> prohibited
        this article from being saved:</h2>

      <ul>

        <% @article.errors.full_messages.each do |msg| %>

          <li><%= msg %></li>

        <% end %>

      </ul>

    </div>

  <% end %>

<p>

  <%= f.label :title %><br>

  <%= f.text_field :title %>

</p>

<p>

  <%= f.label :text %><br>

  <%= f.text_area :text %>

</p>

<p>
```

```
<%= f.submit %>

</p>

<% end %>
```

Todo este contenido, excepto la declaración `form_for`, es idéntico al contenido que ya teníamos. Como `@article` es un recurso que incluye todas las rutas RESTful y Rails es capaz de inferir qué URI y método se debe utilizar, podemos simplificar al máximo la declaración `form_for`. Consulta la guía [Resource-oriented style](#) para obtener más información sobre este uso de `form_for`.

Ahora actualiza completamente la vista `app/views/articles/new.html.erb` para usar el nuevo parcial:

```
<h1>New article</h1>

<%= render 'form' %>

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

Luego haz lo mismo para la vista `app/views/articles/edit.html.erb`:

```
<h1>Edit article</h1>

<%= render 'form' %>

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

5.13. Borrando artículos

Ahora ya estamos listos para completar la letra "D" del acrónimo CRUD: borrar artículos de la base de datos. Siguiendo la convención REST, la ruta para borrar artículos debería ser la que indica el comando `rake routes`:



```
$ bin/rake routes
```

Prefix	Verb	URI Pattern	Controller#Action
	DELETE	/articles/:id(.:format)	articles#destroy
...			

El método `delete` es el que se debe usar para eliminar recursos de la aplicación. Si esta ruta siguiera el formato `GET` tradicional, los usuarios maliciosos podrían crear enlaces como el siguiente:

```
<a href='http://example.com/articles/1/destroy'>¡Mira este precioso gatito!</a>
```

Para evitarlo se utiliza el método `DELETE` de HTTP mediante la ruta que está asociada con la acción `destroy` del archivo `app/controllers/articles_controller.rb`.

Como esta acción no existe todavía, créala con el siguiente contenido:

```
def destroy

  @article = Article.find(params[:id])

  @article.destroy

  redirect_to articles_path

end
```

Puedes invocar el método `destroy` en cualquier objeto *Active Record* para eliminarlo de la base de datos. Como después del borrado de información se redirige al usuario a la acción `index`, no es necesario crear una vista para esta acción `destroy`.

Finalmente, agrega un enlace a la plantilla de la acción `index` (`app/views/articles/index.html.erb`) para completar todo.

```
<h1>Listing Articles</h1>

<%= link_to 'New article', new_article_path %>

<table>

  <tr>

    <th>Title</th>

    <th>Text</th>

    <th colspan="3"></th>
```

```

</tr>

<% @articles.each do |article| %>

  <tr>

    <td><%= article.title %></td>

    <td><%= article.text %></td>

    <td><%= link_to 'Show', article_path(article) %></td>

    <td><%= link_to 'Edit', edit_article_path(article) %></td>

    <td><%= link_to 'Destroy', article_path(article),
method: :delete, data: { confirm: 'Are you sure?' } %></td>

  </tr>

<% end %>

</table>

```

En esta plantilla se utiliza `link_to` de una manera diferente. El nombre de la ruta se pasa como segundo argumento y después se indican las opciones mediante otro argumento. Las opciones `:method` y `:data-confirm` se utilizan como atributos HTML5 para que cuando se pinche sobre el enlace Rails muestre un mensaje de confirmación al usuario antes de borrar la información.

Este comportamiento es posible gracias al archivo JavaScript llamado `jquery_ujs`, que se incluye automáticamente en el layout de la aplicación (`app/views/layouts/application.html.erb`) que se creó al generar la aplicación Rails. Si no se incluye este archivo, el mensaje de confirmación no se muestra.

Listing Articles

[New article](#)

Title

Text

Rails is awesome! It really is. [Show](#) [Edit](#) [Destroy](#)



¡Felicidades! Ahora ya puedes crear, mostrar, listar, actualizar y borrar artículos.

Nota

Según la filosofía de Rails, es preferible usar objetos que representan a recursos, en vez de crear las rutas a mano. Para obtener más información sobre el enrutamiento, consulta el artículo [Rails Routing from the Outside In](#).

Capítulo 6. Añadiendo otro modelo

A continuación vamos a añadir un segundo modelo a la aplicación. Este nuevo modelo se encargará de gestionar los comentarios de los artículos.

6.1. Generando el modelo

Para generar el nuevo modelo usaremos el mismo generador que se explicó anteriormente para el modelo `Article`. Esta vez se creará un modelo llamado `Comment` que gestionará los comentarios de los artículos. Para crearlo, ejecuta el siguiente comando:

```
$ bin/rails generate model Comment commenter:string body:text article:references
```

Como resultado de este comando se generarán cuatro archivos:

File	Purpose
db/migrate/20140120201010_create_comments.rb	Archivo de migración para crear la tabla <code>comments</code> en la base de datos (en tu caso el nombre del archivo será ligeramente diferente)
app/models/comment.rb	El modelo <code>Comment</code>
test/models/comment_test.rb	Los tests del modelo
test/fixtures/comments.yml	Comentarios de prueba para los tests

Primero echa un vistazo al archivo `app/models/comment.rb`:

```
class Comment < ActiveRecord::Base
```

```
  belongs_to :article
```

```
end
```

Su contenido es muy similar al del modelo `Article` generado anteriormente. La única diferencia es la línea `belongs_to :article`, que configura una relación para *Active Record*. En la próxima sección se explican estas relaciones entre modelos.

Además del modelo, Rails genera un archivo de migración para crear la tabla correspondiente en la base de datos:

```
class CreateComments < ActiveRecord::Migration
```

```
  def change
```

```
    create_table :comments do |t|
```

```
      t.string :commenter
```

```
      t.text :body
```

```
      # this line adds an integer column called `article_id`.
```

```
      t.references :article, index: true
```


```
    t.timestamps
```

```
  end
```

```
end
```

```
end
```

La línea `t.references` crea una columna de tipo *clave foránea* para establecer la relación entre los dos modelos. Además se crea un índice para esta columna. Como ya tenemos todo preparado, ejecuta el siguiente comando:



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

Rails es lo bastante *inteligente* como para ejecutar solamente las migraciones que todavía no se han ejecutado en la base de datos que se está utilizando. Así que el resultado de ejecutar el comando será:

```
== CreateComments: migrating =====
-- create_table(:comments)
-> 0.0115s
== CreateComments: migrated (0.0119s) =====
```

6.2. Asociando modelos

Las asociaciones de *Active Record* permiten declarar las relaciones que existen entre dos modelos. En el caso de los comentarios y los artículos, las relaciones se podrían escribir de esta manera:

- Cada comentario pertenece (en inglés, "*belongs to*") a un artículo.
- Un artículo puede tener muchos (en inglés, "*have many*") comentarios.

Si te fijas un poco, la sintaxis que utiliza Rails es prácticamente la misma que como se describen las relaciones en inglés. Recuerda la línea del modelo `Comment` (archivo `app/models/comment.rb`) que establece la relación con `Article`:

```
class Comment < ActiveRecord::Base

  belongs_to :article

end
```

Ahora edita el archivo `app/models/article.rb` para definir el otro extremo de la relación:

```
class Article < ActiveRecord::Base

  has_many :comments

  validates :title, presence: true,

              length: { minimum: 5 }

end
```

Gracias a estas dos declaraciones (`belongs_to` y `has_many`), Rails puede hacer casi todo el trabajo automáticamente. Si por ejemplo tienes una variable de instancia llamada `@article` que contiene un artículo, puedes obtener todos sus comentarios mediante la instrucción `@article.comments`.

Nota

Consulta el artículo [Active Record Associations](#) para obtener más información sobre las asociaciones.

6.3. Añadiendo una ruta para los comentarios

En primer lugar debemos añadir una nueva ruta para que Rails sepa dónde queremos navegar para ver los comentarios. Para ello, abre el archivo `config/routes.rb` y haz que tenga el siguiente contenido:

```
resources :articles do
  resources :comments
end
```

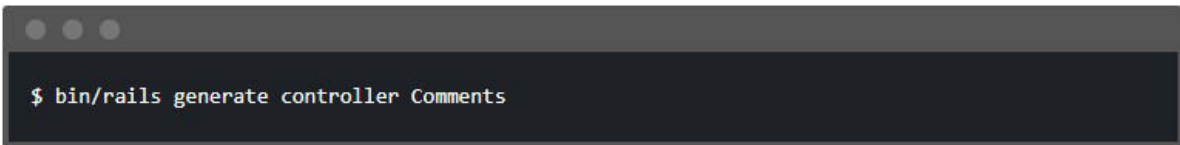
Esta configuración crea la ruta `comments` dentro de la ruta `articles` que definimos anteriormente. Esta es otra forma de establecer la relación entre los dos modelos.

Nota

Para obtener más información sobre el enrutamiento, consulta la guía [Routing Guide](#).

6.4. Generando un controlador

El modelo ya está creado así que ahora podemos dedicarnos a su controlador asociado. De nuevo utilizaremos el comando que genera controladores:



```
$ bin/rails generate controller Comments
```

Como resultado de este comando se generan seis archivos y un directorio vacío:

Archivo/Directorio	Propósito
app/controllers/comments_controller.rb	El controlador Comments
app/views/comments/	Directorio donde guardar las vistas del controlador
test/controllers/comments_controller_test.rb	El test funcional del controlador
app/helpers/comments_helper.rb	El <i>helper</i> para las vistas relacionadas con los comentarios
test/helpers/comments_helper_test.rb	Test unitario para el <i>helper</i>
app/assets/javascripts/comment.js.coffee	Archivo CoffeeScript para las vistas del controlador
app/assets/stylesheets/comment.css.scss	Hoja de estilos CSS para las vistas del controlador

Como sucede en cualquier blog de Internet, los usuarios podrán añadir comentarios en los artículos y después de hacerlo, se les redirigirá a la página de ese mismo artículo para que puedan ver su comentario publicado. Por eso el controlador `CommentsController` deberá incluir un método para crear comentarios y para borrar todos los comentarios de *spam* que lleguen.

Así que en primer lugar vamos a modificar la plantilla que muestra los artículos (archivo `app/views/articles/show.html.erb`) para que deje crear nuevos comentarios:

```
<p>
  <strong>Title:</strong>

  <%= @article.title %>
</p>
```



```

<p>

  <strong>Text:</strong>

  <%= @article.text %>

</p>

<h2>Add a comment:</h2>

<%= form_for([@article, @article.comments.build]) do |f| %>

  <p>

    <%= f.label :commenter %><br>

    <%= f.text_field :commenter %>

  </p>

  <p>

    <%= f.label :body %><br>

    <%= f.text_area :body %>

  </p>

  <p>

    <%= f.submit %>

  </p>

<% end %>

<%= link_to 'Back', articles_path %>

| <%= link_to 'Edit', edit_article_path(@article) %>

```

Este código añade un formulario en la página `show` de los artículos para poder crear nuevos comentarios llamando a la acción `create` del controlador `CommentsController`. En este caso el método `form_for` utiliza un array que crea rutas anidadas de tipo `/articles/1/comments`.

El siguiente paso consiste en crear la acción `create` en el archivo `app/controllers/comments_controller.rb`:

```
class CommentsController < ApplicationController

  def create

    @article = Article.find(params[:article_id])

    @comment = @article.comments.create(comment_params)

    redirect_to article_path(@article)

  end

  private

  def comment_params

    params.require(:comment).permit(:commenter, :body)

  end

end
```

Este controlador es un poco más complejo del que creamos para los artículos. Esta es una de las consecuencias de anidar relaciones. Cada petición relacionada con un comentario debe contener una referencia al artículo con el que está relacionado. Por eso tenemos que buscar primero el modelo `Article` que está relacionado con el comentario.

Además, la acción utiliza algunos de los métodos disponibles para las relaciones. Así por ejemplo se utiliza el método `create` sobre `@article.comments` para crear y guardar un comentario. Esto hace que el comentario esté automáticamente relacionado con este artículo específico.

Después de crear el nuevo comentario, se redirige al usuario de nuevo a la página que muestra el artículo original mediante el *helper* `article_path(@article)`. Como acabamos de ver, este *helper* llama a la acción `show` del controlador `ArticlesController`, que a su vez renderiza la plantilla `show.html.erb`. Como esta es la plantilla en la que se debe mostrar el nuevo comentario, añade lo siguiente en el archivo `app/views/articles/show.html.erb`:

```
<p>

  <strong>Title:</strong>

  <%= @article.title %>

</p>
```

```
<p>

  <strong>Text:</strong>

  <%= @article.text %>

</p>
```

```
<h2>Comments</h2>

<% @article.comments.each do |comment| %>

  <p>

    <strong>Commenter:</strong>

    <%= comment.commenter %>

  </p>
```

```
<p>

  <strong>Comment:</strong>

  <%= comment.body %>

</p>
```

```
<% end %>
```

```
<h2>Add a comment:</h2>

<%= form_for([@article, @article.comments.build]) do |f| %>
```

```

<p>

  <%= f.label :commenter %><br>

  <%= f.text_field :commenter %>

</p>

<p>

  <%= f.label :body %><br>

  <%= f.text_area :body %>

</p>

<p>

  <%= f.submit %>

</p>

<% end %>

<%= link_to 'Edit Article', edit_article_path(@article) %> |
<%= link_to 'Back to Articles', articles_path %>

```

Ahora ya puedes añadir artículos y comentarios en el blog y cada contenido se muestra en el lugar adecuado.

Title: Rails is awesome!

Text: It really is.

Comments

Commenter: A fellow dev

Comment: I agree!!!

Add a comment:

Commenter

Body

Create Comment

[Back](#) | [Edit](#)

Capítulo 7. Refactorización

La aplicación ya permite crear artículos y comentarios, pero si echas un vistazo al archivo `app/views/articles/show.html.erb` verás que la plantilla es muy larga y su código es un poco caótico. Para limpiar el código podemos hacer uso de los *parciales*.

7.1. Renderizando colecciones de parciales

En primer lugar vamos a crear un parcial para mostrar todos los comentarios de un artículo. Crea el archivo `app/views/comments/_comment.html.erb` y añade el siguiente contenido:

```
<p>

  <strong>Commenter:</strong>

  <%= comment.commenter %>

</p>
```

```
<p>

  <strong>Comment:</strong>

  <%= comment.body %>

</p>
```

Then you can change `app/views/articles/show.html.erb` to look like the following:

Ahora puedes simplificar el archivo `app/views/articles/show.html.erb` de la siguiente manera:

```
<p>

  <strong>Title:</strong>

  <%= @article.title %>

</p>
```

```

<p>

  <strong>Text:</strong>

  <%= @article.text %>

</p>

<h2>Comments</h2>

<%= render @article.comments %>

<h2>Add a comment:</h2>

<%= form_for([@article, @article.comments.build]) do |f| %>

  <p>

    <%= f.label :commenter %><br>

    <%= f.text_field :commenter %>

  </p>

  <p>

    <%= f.label :body %><br>

    <%= f.text_area :body %>

  </p>

  <p>

    <%= f.submit %>

  </p>

<% end %>

<%= link_to 'Edit Article', edit_article_path(@article) %> |

<%= link_to 'Back to Articles', articles_path %>

```

Ahora el *partial* definido en `app/views/comments/_comment.html.erb` se utiliza para renderizar cada comentario de la colección `@article.comments`. Como el método itera sobre `@article.comments`, asigna cada comentario a una variable local con el mismo nombre que el *partial* (en este caso, `comment`).

7.2. Renderizando un formulario parcial

El siguiente paso consiste en extraer de la plantilla la sección para crear nuevos comentarios. Crea el archivo `app/views/comments/_form.html.erb` y añade el siguiente contenido:

```
<%= form_for([@article, @article.comments.build]) do |f| %>

  <p>

    <%= f.label :commenter %><br>

    <%= f.text_field :commenter %>

  </p>

  <p>

    <%= f.label :body %><br>

    <%= f.text_area :body %>

  </p>

  <p>

    <%= f.submit %>

  </p>

<% end %>
```

Ahora ya puedes refactorizar la plantilla `app/views/articles/show.html.erb` de la siguiente manera:

```
<p>

  <strong>Title:</strong>

  <%= @article.title %>
```



```
</p>
```

```
<p>
```

```
  <strong>Text:</strong>
```

```
  <%= @article.text %>
```

```
</p>
```

```
<h2>Comments</h2>
```

```
<%= render @article.comments %>
```

```
<h2>Add a comment:</h2>
```

```
<%= render "comments/form" %>
```

```
<%= link_to 'Edit Article', edit_article_path(@article) %> |
```

```
<%= link_to 'Back to Articles', articles_path %>
```

El segundo `render` simplemente indica la plantilla *parcial* que se quiere renderizar. Rails es capaz de interpretar la sintaxis `comments/form` de manera correcta, así que se renderiza el *parcial* `_form.html.erb` del directorio `app/views/comments`.

El objeto `@article` está disponible en cualquier *parcial* renderizado dentro de la vista, ya que se ha definido como una variable de instancia.

Capítulo 8. Borrando comentarios

Otra de las funcionalidades básicas de cualquier blog es la posibilidad de borrar los comentarios de *spam*. Para ello, vamos a añadir un enlace en la vista y una acción llamada `destroy` en el controlador `CommentsController`.

En primer lugar, añade el enlace para borrar comentarios en la plantilla del *partial* `app/views/comments/_comment.html.erb`:

```
<p>

  <strong>Commenter:</strong>

  <%= comment.commenter %>

</p>

<p>

  <strong>Comment:</strong>

  <%= comment.body %>

</p>

<p>

  <%= link_to 'Destroy Comment', [comment.article, comment],
            method: :delete,
            data: { confirm: 'Are you sure?' } %>

</p>
```

Al pinchar el enlace "*Destroy Comment*", se envía una petición HTTP de tipo `DELETE` a la URL `/articles/:article_id/comments/:id` que será respondida por `CommentsController`. Así que vamos a crear la acción `destroy` en el controlador (archivo `app/controllers/comments_controller.rb`):

```
class CommentsController < ApplicationController
```

```

def create

  @article = Article.find(params[:article_id])

  @comment = @article.comments.create(comment_params)

  redirect_to article_path(@article)

end


def destroy

  @article = Article.find(params[:article_id])

  @comment = @article.comments.find(params[:id])

  @comment.destroy

  redirect_to article_path(@article)

end


private

def comment_params

  params.require(:comment).permit(:commenter, :body)

end

end

```

La acción `destroy` busca el artículo que estamos leyendo, localiza el comentario dentro de la colección `@article.comments`, lo borra de la base de datos y vuelve a mostrar la página con el artículo original.

8.1. Borrando los objetos asociados

Si borras un artículo, todos sus comentarios también deberían borrarse para que no ocupen un espacio innecesario en la base de datos. Rails permite configurar este comportamiento mediante la opción `dependent` de la asociación entre modelos.

Para ello, modifica el modelo `Article` cambiando el contenido del archivo `app/models/article.rb` por lo siguiente:

```
class Article < ActiveRecord::Base

  has_many :comments, dependent: :destroy

  validates :title, presence: true,

              length: { minimum: 5 }

end
```

Capítulo 9. Seguridad

9.1. Autenticación básica

Si publicaras ahora tu aplicación Rails, cualquier persona podría añadir, editar y borrar artículos y comentarios. Obviamente Rails incluye algunas opciones de seguridad para evitar esto. Una de ellas es la autenticación basada en HTTP.

La clave consiste en proteger el acceso a varias de las acciones definidas en el controlador `ArticlesController`. Si el usuario no está autenticado, no podrá acceder a esas acciones. Para ello utilizaremos el método `http_basic_authenticate_with` de Rails.

La configuración de la autenticación en este caso consiste en indicar al principio del controlador `ArticlesController` que queremos proteger todas las acciones salvo `index` y `show`:

```
class ArticlesController < ApplicationController

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

  def index

    @articles = Article.all

  end

  # ...
```

Otra restricción adicional consiste en evitar que los usuarios puedan borrar comentarios. Para ello, añade lo siguiente en `CommentsController` (archivo `app/controllers/comments_controller.rb`):

```
class CommentsController < ApplicationController

  http_basic_authenticate_with name: "dhh", password: "secret", only: :destroy
```

```

def create

  @article = Article.find(params[:article_id])

  ...

end

# ...

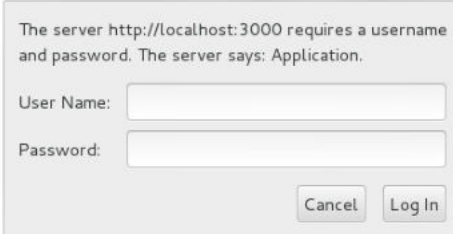
```

Si ahora tratas de crear un nuevo artículo, verás la siguiente ventana en la que el navegador te solicita un usuario y contraseña.

Listing Articles

[New article](#)

Title	Text
Rails is awesome! It really is.	Show Edit Destroy



Las aplicaciones Rails pueden utilizar obviamente otros métodos de autenticación alternativos. Dos de las alternativas más populares se llaman [Devise](#) y [Authlogic](#).

9.2. Otros comentarios sobre la seguridad

La seguridad, sobre todo cuando hablamos de aplicaciones web, es un tema muy complejo. Por eso puedes consultar la guía [Ruby on Rails Security Guide](#) para obtener más información al respecto.

Capítulo 10. Siguientes pasos

Después de crear tu primera aplicación Rails de prueba, ya deberías ser capaz de empezar a *jugar* con ella para probar nuevas cosas por tu cuenta. En cualquier caso, no estás solo en este proceso de aprendizaje. Como seguramente tarde o temprano necesitarás ayuda, puedes echar mano a estos recursos:

- Las [Guías de Ruby on Rails en inglés](#).
- El [Tutorial de Rails en inglés](#)
- La [lista de correo en inglés](#).
- El canal #rubyonrails de irc.freenode.net

Rails también incluye una completa ayuda que puedes consultar mediante la utilidad rake de la línea de comandos:

- Ejecuta `rake doc:guides` para obtener una copia completa de las guías de Rails en el directorio `doc/guides` de tu aplicación. Después sólo tienes que abrir el archivo `doc/guides/index.html` en tu navegador favorito.
- Ejecuta `rake doc:rails` para obtener una copia completa de la documentación de la API de Rails en el directorio `doc/api` de la aplicación. Después, abre el archivo `doc/api/index.html` con tu navegador preferido.

Nota:

Para poder generar las guías de Rails localmente mediante el comando `doc:guides`, debes instalar primero la gema RedCloth. Añádela al archivo Gemfile y ejecuta `bundle install` para instalarla.