UNIVERSIDAD NACIONAL AUTÓNOMA

DE MÉXICO

FACULTAD DE CIENCIAS

MATERIAL DE APOYO PARA EL LABORATORIO DE INGENIERÍA DE SOFTWARE USANDO EL MARCO DE TRABAJO RUBY ON RAILS

ACTIVIDAD DE APOYO A LA DOCENCIA

QUE PARA OBTENER EL TÍTULO DE:

LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

P R E S E N T A :

ALFREDO PAVEL CEDILLO PÉREZ

DIRECTOR DE TESIS:

M. EN C. MARÍA GUADALUPE ELENA IBARGÜENGOITIA GONZÁLEZ

2015

Universidad Nacional Autónoma de México
UNAM
Biblioteca Central
Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Índice de contenido

HOJA DE CONTACTO.

Nombre del Alumno(a): Alfredo Pavel Cedillo Pérez

Carrera: Ciencias de la Computación

Número de Cuenta: 304053320

Modalidad: Actividad de Apoyo a la Docencia

Correo Electrónico: alfredo.pcp@ciencias.unam.mx

Grado y Nombre del Propietario: Dra. Hanna Jadwiga Oktaba

Correo Electrónico: hanna.oktaba@ciencias.unam.mx

Grado y Nombre del Propietario: M. en I. Miguel Ehécatl Morales Trujillo

Correo Electrónico: migmor@ciencias.unam.mx

Grado y Nombre del Propietario: M en C. María Guadalupe Elena Ibargüengoitia González

Correo Electrónico: gig@ciencias.unam.mx

Grado y Nombre del Propietario: M. en I. Yasmine Macedo Reza

Correo Electrónico: yasmine28@gmail.com

Grado y Nombre del Propietario: M. en I. Gerardo Avilés Rosas

Correo Electrónico: gar@ciencias.unam.mx

Dedicatoria y Agradecimientos.

.

Quiero dedicar y agradecer esta tesis a todas las personas que me ayudaron de manera directa e indirecta en terminar este trabajo. A mis padres, Alfredo y Reyna por ofrecerme siempre su apoyo incondicional y por creer en mi en todo este tiempo que demore en terminarla. A mis hermanas Karla y Katia que de igual forma me inspiraron para seguir adelante y no dejarlo de lado. A mi hermano Yerik, que aunque es muy pequeño, es uno de los pilares que me mantienen en el camino correcto. A mi primo Andrés y a mis tias Cecilia y Estela les dedico este trabajo porque me han enseñado el valor de lo que es la unión familiar. A toda la familia (que es mucha y no me cabrían los nombres)por parte de mi mamá, en especial a mi abuelita Gabriela, les agradezco su confianza y su apoyo, deseándoles mucho éxito como me lo desearon a mi.

Dedico y agradezco también este trabajo a mi novia Elizabeth que siempre estuvo al pendiente de mi progreso, y que se tomo la molestia de leer y darme comentarios constructivos. Pocas personas como tú se encuentran en este mundo tan surrealista,

Agradezco a todos mis amigos que he conocido en lo que va de mi vida y que he conservado. A mis hermanos Alfredo, Josué, Alan, Alfonso y Christian que me muestran el valor de la amistad de forma muy peculiar. Hermanos David y Miguel con los que pasado muchos momentos de aprendizaje y de reflexión, de buena vibra y de risas. A Carlos un hermano con el que he reflexionado desde varios puntos de vista la vida cotidiana, y que siempre tiene una plática interesante y a mi hermano Alan al que conozco desde la secundaria y ha sido mi más viejo amigo y con el que fui aprendiendo a tocar música. A mis hermanas Beatriz y Laura que estimo mucho y pese a las cancelaciones y no vernos en lapsos de tiempo largos, siempre están ahí con un consejo o un buen chiste. A mi otra hermana Sandra, que es una persona muy interesante y muy culta, proporcional a su sentido del humor. A mi compadre Mario y a la comadre Yasmine, las dos personas que me han enseñado bastante en el ámbito laboral, social y por lo que valía la pena ir a trabajar.

A mi asesora la Mtra. Guadalupe (Lupita) que fue muy paciente y que no perdió la esperanza en mi, gracias por tu tiempo y por guiarme en todo este camino de aprendizaje y formación. A todo mi jurado (Dra. Hanna, Mtro. Miguel, Mtra. Yasmine y el Mtro. Gerardo) que revisó mi trabajo y me apoyo

en la corrección de mis errores gramaticales y mis errores lógicos. Muchas gracias por su ayuda y que la vida les retorne sólo cosas buenas y éxito.

Por último, va una dedicatoria va una dedicatoria y agradecimiento a aquella persona que lea este trabajo, de corazón.

“— Maestro, mi mente está confusa. Aprendemos a fortalecer nuestro cuerpo y, sin embargo, se nos enseña que debemos respetar a todos aquellos contra quienes podríamos emplear nuestra fuerza.
— Cuando tu vida o la de un inocente sea amenazada, entonces sabrás defenderla.
— Y, estando mejor preparado que los demás para hacerlo, ¿no debería enfrentarme al enemigo y luchar?
— Ignora a quien te insulte. Controla el hervor de tu sangre. Huye del que te ha ofendido.
— ¿No es esa la actitud de un cobarde?
— El jabalí salvaje huye del tigre. La naturaleza ha provisto a ambos con armas mortales y ambos saben que pueden causar la muerte del otro. Al huir, el jabalí salva su propia vida y también la del tigre: eso no es cobardía, es amor a la vida”.

Maestro Po (Kung Fu, La Leyenda).

"Si las puertas de la percepción se depurasen, todo aparecería a los hombres como realmente es: infinito. Pues el hombre se ha encerrado en sí mismo hasta ver todas las cosas a través de las estrechas rendijas de su caverna."

William Blake (Las bodas del cielo y el infierno).

Capítulo 1: Introducción

El curso de Ingeniería de Software de la carrera de Ciencias de la Computación de la Facultad de Ciencias es una materia obligatoria correspondiente al séptimo semestre (plan 1994) y al sexto semestre (plan 2013), generalmente sigue un método de desarrollo de software en el cual, se genera una aplicación Web. Algunos alumnos prefieren realizar esta aplicación usando *Ruby on Rails*. La materia ya cuenta con material de apoyo para hacer la aplicación con tecnología Java, falta el material de apoyo para los que quieren hacerlo con *Rails*.

El objetivo de este trabajo es desarrollar una aplicación Web en *Ruby on Rails* paso a paso, de forma que sirva de apoyo a dichos alumnos, será necesario el adquirir y/o expandir sus conocimientos y habilidades necesarias para el desarrollo de software para la plataforma web.

La aplicación que se desarrollará de apoyo es un *microblogging*[1] que permita la interacción entre usuarios a través de mensajes (también llamados *post*). Además de *Ruby on Rails*, el conjunto de herramientas a utilizar, incluye control de versiones, *HTML/CSS*, bases de datos, creación y modificación de usuarios y mensajes; seguridad básica y despliegue de la aplicación en un servidor. Se usará como controlador de versiones, el software libre *Git*.

El desarrollo de la aplicación parte desde cero, es decir, se genera un esqueleto de la aplicación y poco a poco se agregan las funciones necesarias que cumplan con lo siguiente:

- Creación de un esqueleto preliminar del sistema: es decir, el alumno podrá ver cómo es que *Rails* genera a partir de unos cuantos comandos, el esqueleto funcional y básico de nuestro sistema.
- Uso de páginas estáticas y dinámicas.
- Manejo de hojas de estilo: el sistema debe contar con una interfaz no demasiada básica, es por eso que aquí se darán las bases del uso de estilos CSS[2].
- Generación de modelos, vistas y controladores para el sistema usando *Ruby on Rails*.
- Registro, actualización, muestra y eliminación de usuarios y mensajes.
- Autenticación de usuarios.
- Seguridad básica.

[1] *Microblogging* es una forma de *blogging* (un blog es un sitio informativo que se publica en Internet y consiste en redactar y subir, entradas o mensajes), que permite a los usuarios escribir mensajes cortos de texto y publicarlos para que sean vistos por cualquiera o por un grupo restringido del usuario.

[2] Hojas de Estilo en Cascada.

Esta aplicación podrá ser usada como una guía de consulta en los temas relacionados en el ambiente de desarrollo empleado o bien, en el caso que se requiera, como ejemplo práctico para entender un problema y una solución al mismo. A continuación se explica lo que contiene cada capítulo.

Empezando con el capítulo 2 se dan los pasos necesarios para que alumno configure el ambiente de desarrollo, instalando los paquetes necesarios y las herramientas correctamente configuradas.

En el capítulo 3 se tiene como objetivo que el alumno se familiarice con el lenguaje de programación *Ruby.*

En el capítulo 4 se empieza a desarrollar una aplicación con *Ruby on Rails,* se crea una aplicación que tiene dos parámetros: usuarios y los mensajes de usuarios.

Para el capítulo 5 se crea una aplicación web para probar y añadir páginas estáticas. Después, en el capítulo 6 se enfoca en el uso de hojas de estilo para dar a las vistas, un diseño más profesional.

Siguiendo con el desarrollo de la aplicación, en el capítulo 7 se muestra la adición de usuarios a nuestra aplicación, para ellos diseñaremos el modelo de usuarios y añadiremos las funciones de creación, actualización y búsqueda de usuarios.

Pasando al capítulo 8 donde se empieza a desarrollar el registro de usuarios a través de una interfaz web.

Para el capítulo 9 se muestra la implementación que los usuarios puedan iniciar y cerrar sesión. En el capítulo 10, se agregan las acciones *REST* que faltan para nuestro recurso Usuarios, es decir, las acciones: editar (*edit*), actualizar (*update*), indexar (*index*) y destruir (*destroy*). Finalmente con el capítulo 11 se abarca donde lo relacionado a los mensajes de usuarios. Justo aquí es donde empezamos el modelo de mensajes para posteriormente asociar éste y el modelo de usuarios para tener la relación mensaje-usuario.

Capítulo 2: Configuración de Ambiente

2.1. Ruby

La propia página en español de *Ruby* lo define como "*un lenguaje de programación dinámico y de código abierto enfocado en la simplicidad y productividad*[3]*"*. *Ruby* fue creado por *Yukuhiro "Matz" Matsumoto*, este lenguaje se diseñó con partes de los lenguajes favoritos de Yukuhiro: *Perl, Smalltalk, Eiffel, Ada* y *Lisp*. Como dato curioso, la plataforma social *Twitter* en su primera etapa en la que conoció al mundo, fue desarrollada con *Ruby on Rails*[4].

Algunas de las ventajas que sobresalen de *Ruby* son:

- Es un lenguaje interpretado, es decir, que el código de nuestros programas no necesita ser pre-procesado a través de un *compilador,* pero si necesita pasar por un *intérprete* que se encarga de traducir las instrucciones con semántica "humana" a código "máquina". Esto nos brinda independencia sobre la máquina y el sistema operativo, basta con que exista un intérprete de *Ruby* en él.

- *Ruby* es un lenguaje *orientado a objetos* permitiendo a los usuarios manipular estructuras de datos basadas en *objetos*.

- Es fácil de usar, claro y simple.

- *Ruby* se enfoca en no confundir al programador, tiene una sintaxis muy similar al inglés.

Desventajas de *Ruby*:

- La portabilidad. El problema es que en la actualidad, la mayoría de los *lenguajes compilados* existen para casi todas las plataformas, no pasa lo mismo con las *máquinas virtuales* o *frameworks*.

- Velocidad. Al ser un lenguaje interpretado, la velocidad se ve sumamente afectada. El recolector de basura (*Garbage Collector*) consume demasiada memoria *RAM* y aloja los objetos creados en nuestra aplicación en memoria, saturándola.

Existen tres formas de instalar *Ruby* en nuestro sistema, cada una sugiere un nivel de conocimiento para su ejecución:

- Compilar el código fuente (enfocado hacia los desarrolladores de software).

[3] https://www.ruby-lajng.org/es/

[4] Para conocer más aplicaciones conocidas y desarrolladas con el *framework*, consultar: http://www.developerdrive.com/2011/09/20-best-sites-built-with-ruby-on-rails/

- Instalar *Ruby* a través de terceros (enfocado a usuarios que requieran una instalación rápida).

- A través de algún sistema de gestión de paquetes que soporten *Ruby*.

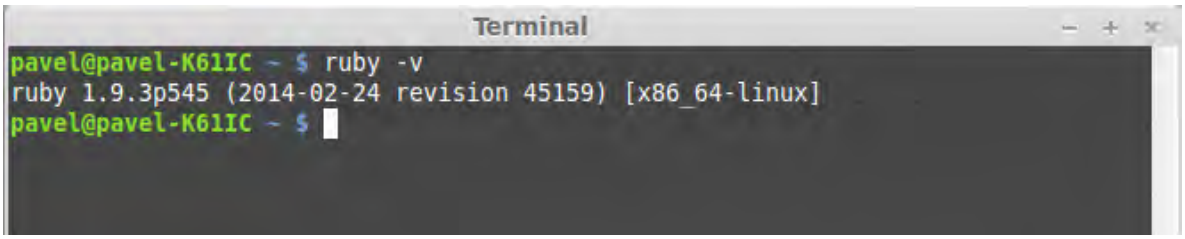
Nos enfocaremos en la instalación a través de terceros, usaremos un intermediario llamado *Ruby Manager Version*[5]. Como su nombre lo indica, es un manejador de versiones para *Ruby,* nos permite instalar distintas versiones del lenguaje, esto con fines propios de cada desarrollador, la razón más común depende de los recursos con los que contamos a la mano a la hora de programar nuestras aplicaciones.

Otro de los motivos al usar software de terceros, es que en el curso de *Ingeniería de Software*, los alumnos cuentan con un periodo determinado de entrega. Debemos de cumplir los tiempos y la configuración toma (en algunos casos) bastante de ese tiempo.

2.1.1. Instalación de Ruby (RVM)

Para empezar podemos verificar si *Ruby* está instalado en nuestro sistema operativo Linux tecleando el siguiente comando:

```
$ ruby -v
```



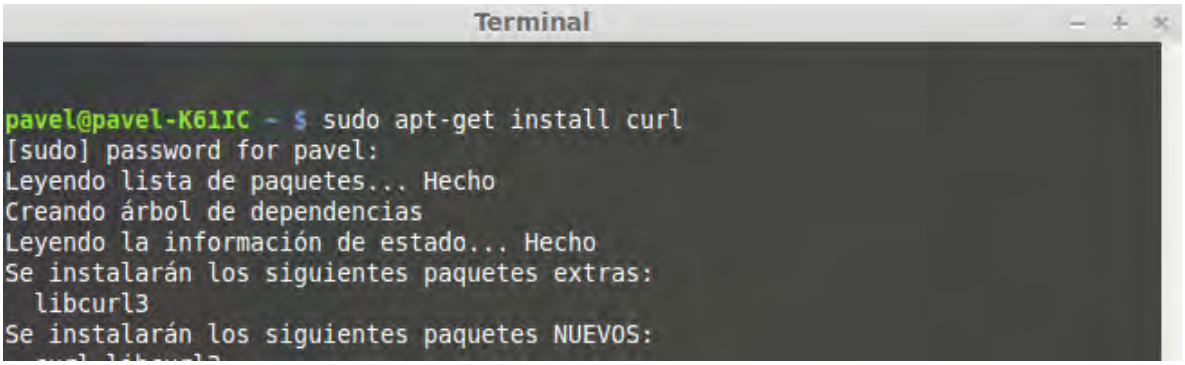
En caso contrario vamos a instalar *Ruby* en nuestro sistema desde cero. Primero procederemos con la instalación en *Linux*[6].

Instalamos *curl*[7] (una herramienta complementaria para un intérprete de comandos, su función es transferir datos con una sintaxis URL a través de distintos protocolos) con el siguiente comando:

```
$ sudo apt-get install curl
```

[5] Página oficial de RVM (en inglés): https://rvm.io/
[6] Para otras distribuciones verificar la web oficial (en inglés): https://www.ruby-lang.org/en/

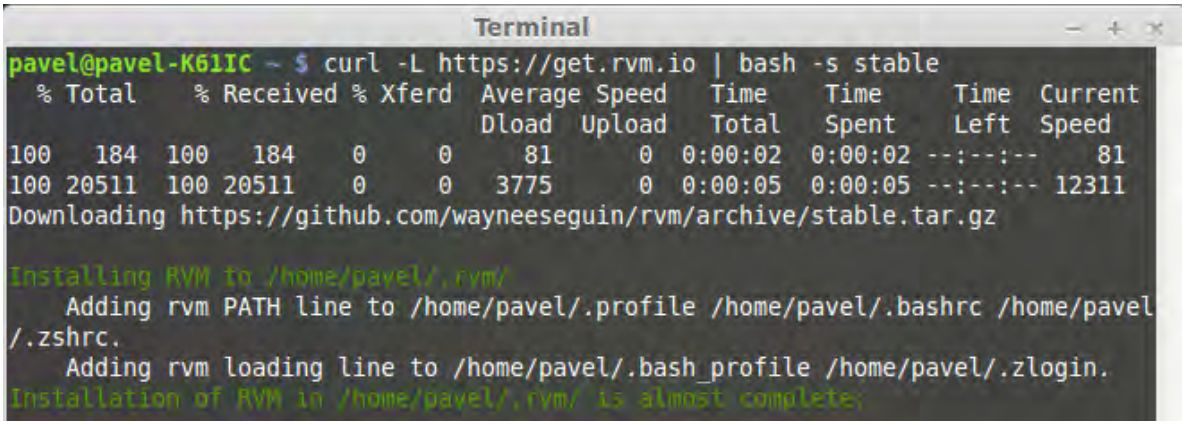
[7] Para más información: http://curl.haxx.se/



Ahora que tenemos instalado *curl* ejecutamos:

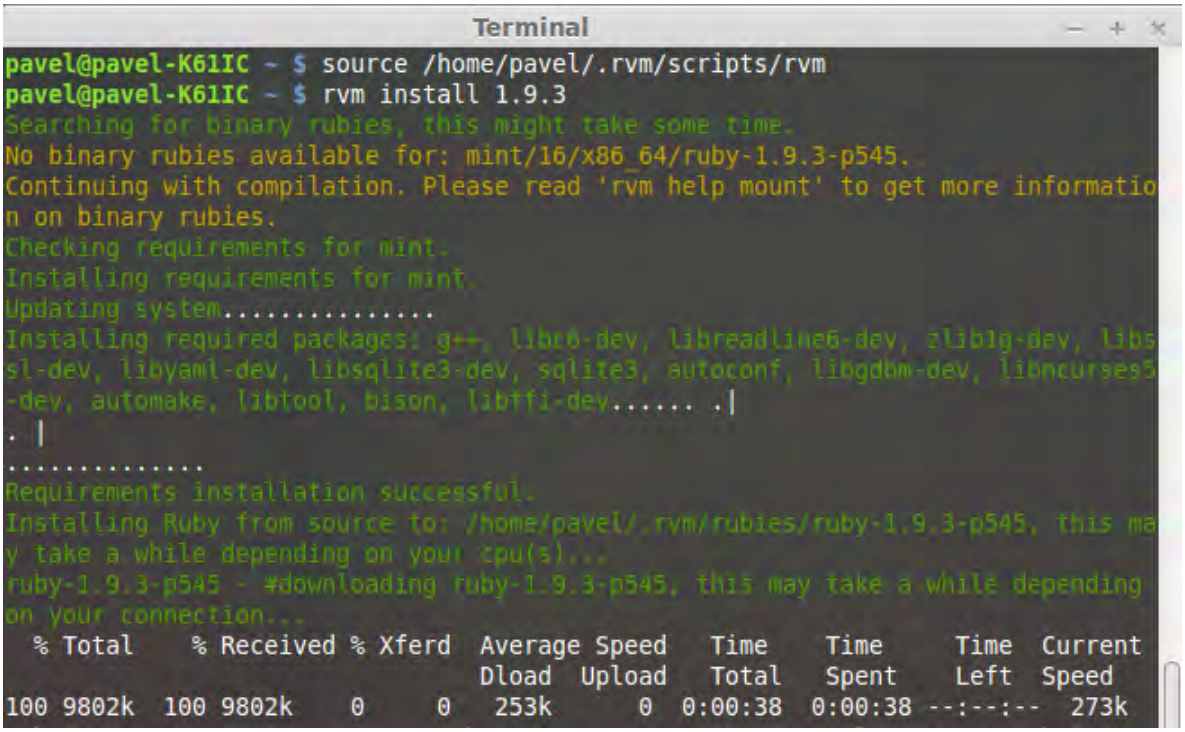
```
$ curl -L https://get.rvm.io | bash -s stable
```

Lo anterior se hace para instalar el RVM en nuestro sistema.



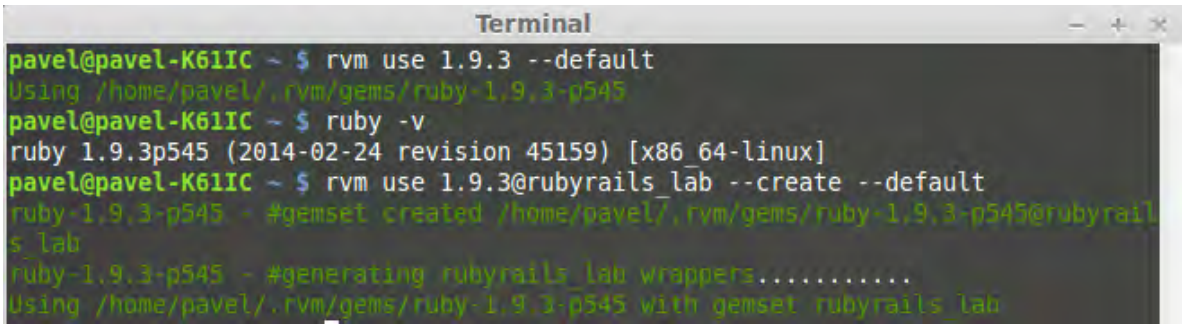
Ya instalado nuestro *RVM*, lo cargamos y después instalamos *Ruby 1.9.3.*, finalmente agregamos esta última versión como default al cargar una nueva terminal y por último verificamos que *Ruby* está en nuestro sistema*:*

```
$ source /home/USUARIO/.rvm/scripts/rvm

$ rvm install 1.9.3

$ rvm use 1.9.3 --default
```



Listo, ya tenemos *Ruby* en nuestro sistema. Ahora debemos de especificar al *RVM* que set de *gemas* va a usar nuestra versión. Las gemas (que se explican más adelante con detalle) son paquete auto-contenidos en *Ruby* creadas por terceros para añadir nuevas funciones al mismo.*Al* tener distintas versiones, pueden existir conflictos, para eso es que tenemos que especificar el set de gemas para la versión de *Ruby* que estemos o vayamos a utilizar. Para nuestra versión (1.9.3) debemos crear un set llamado *rubyrails_lab*:

```
$ rvm -v

$ rvm use 1.9.3@rubyrails_lab --create --default
```

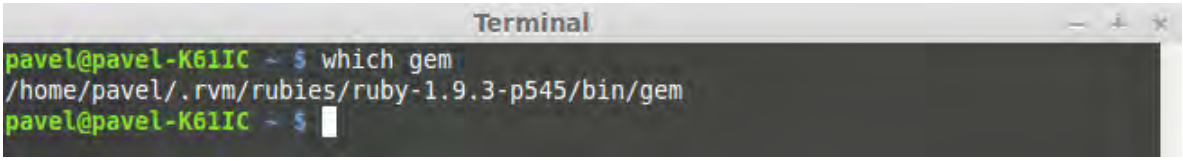


2.1.2. Instalando RubyGems

Citando la definición empleada en la página oficial de *Ruby*: *"RubyGems* es la herramienta preferida por la comunidad para distribuir código. Por lo general, las gemas tienen links hacia la documentación e información sobre los desarrolladores, así que es un buen punto de partida para comenzar a explorar el mundo Ruby".

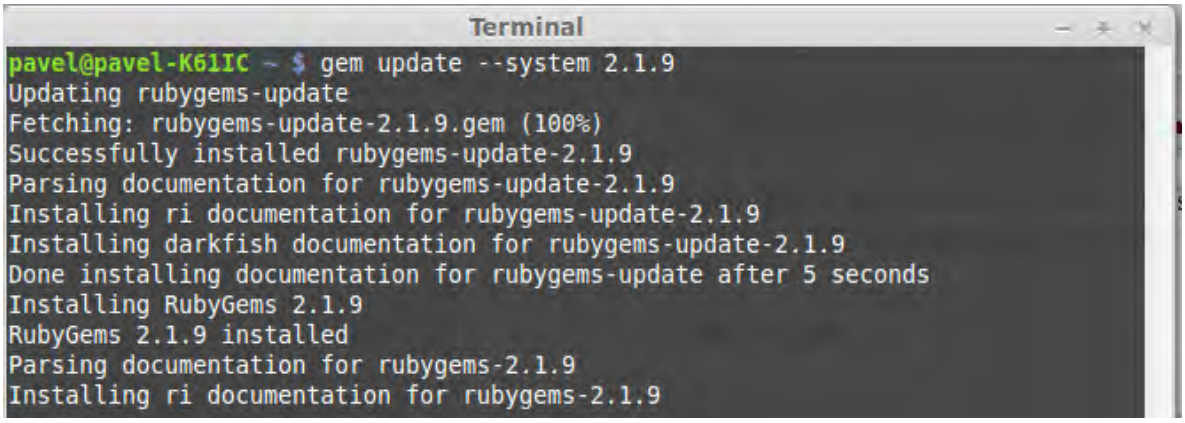
Una gema puede ser plugins o fragmentos de código que pueden ser añadidos a nuestros proyectos de *Ruby on* Rails. El listado de gemas disponibles se encuentra en *RubyForge* (https://rubygems.org/). Al haber instalado *RVM*, ya tenemos *RubyGems*, para verificarlo:

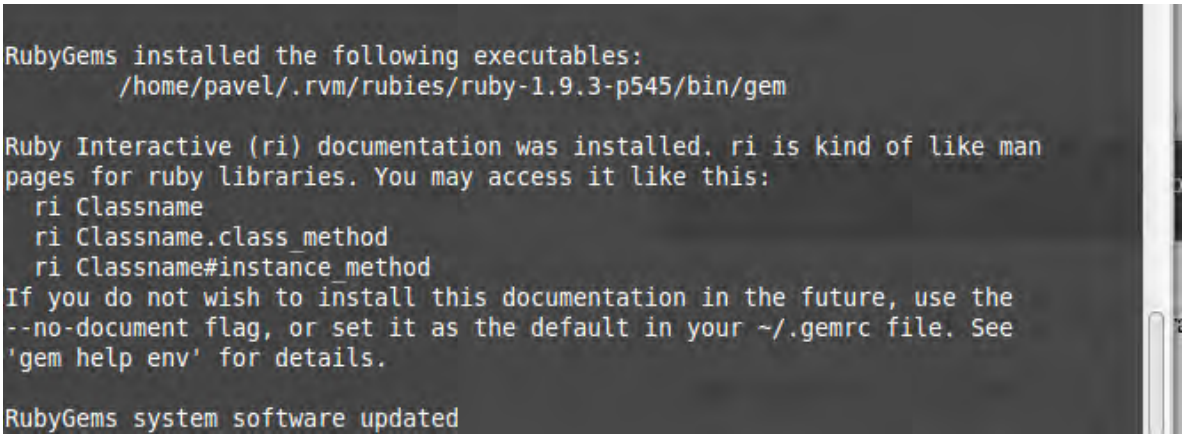
```
$ which gem
```



Debemos tener al menos la versión 2.1.9., para actualizar *RubyGems*:

```
$ gem update --system 2.1.9
```





2.2. Rails

Rails es un *framework* (un conjunto de lenguajes, tecnologías y/o bibliotecas que se enfocan a la reutilización de componentes de sofware para el rápido desarrollo de aplicaciones) de código libre para *Ruby* creado en el año 2003 por *David Heinemeier Hansson*[8] *(*desde entonces *Rails core team*[9] y terceros han optimizado *Rails).* Siguiendo la arquitectura *Modelo-Vista-Controlador*, siempre existe un lugar para cada fragmento de código y estos fragmentos pueden interactuar de una forma estándar. El *framework* al crear nuevas funcionalidades en código, también crea *tests* para la comprobación de éstas.

Rails se creó pensando en crear aplicaciones web de manera rápida y controlada por el programador, con *Rails* una aplicación puede llegar a ser diez veces más rápida de desarrollar, esto gracias a *Ruby* y en parte a dos principios:

- *Convención sobre configuración (Convention over configuration):* que nos dice que siguiendo las convenciones en *Rails* escribiremos menos código para nuestra aplicación, por otro, lado si la configuración que necesitamos no es convencional, entonces la definimos.

- *No te repitas (Don't repeat yourself)*: no te repitas, las definiciones deben hacerse sólo una vez, como mencionamos anteriormente, todo nuestro código interactúa y no necesitamos puentes.

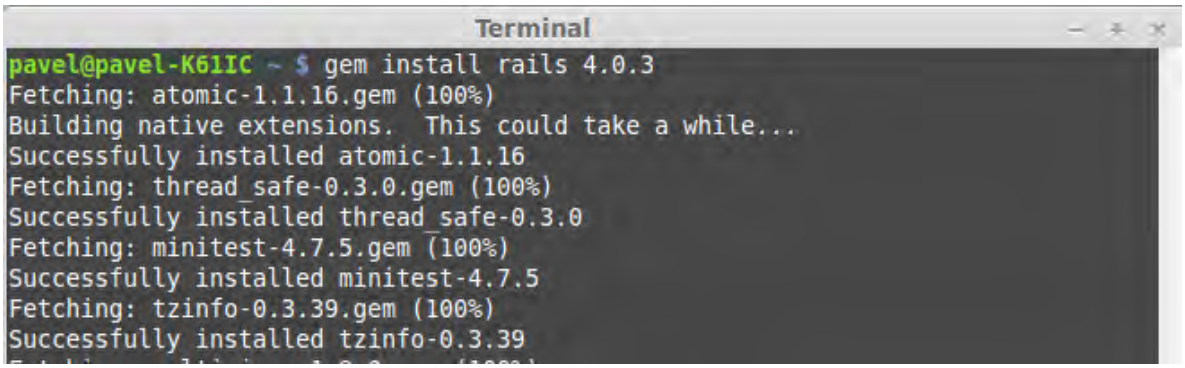
2.2.1. Instalando Rails

Resulta sumamente sencillo instalar *Rails*, teniendo instalado *RubyGems* sólo debemos ejecutar este comando en nuestra consola y luego verificamos la instalación:

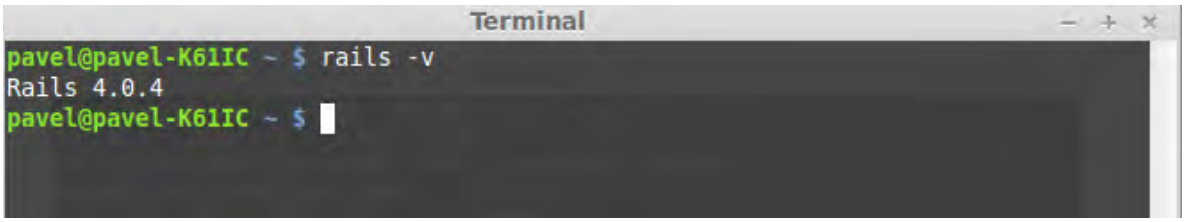
```
$ gem install rails --version 4.0.3
```

[8] Página personal de David Heinemeier: http://david.heinemeierhansson.com/

[9] Página del *Rails core team*: http://rubyonrails.org/core




```
$ rails -v
```



2.3. Git

En el laboratorio de *Ingeniería de Software* es muy importante que se maneje un control de versiones óptimo, eficiente y sencillo. Siempre que hacemos cambios a nuestros archivos y/o añadimos nuevos, como pueden ser: fragmentos de código, programas, recursos web, archivos de terceros, debemos de ser capaces de guardar dichos cambios y/o añadidos y a su vez poder deshacerlos, debemos también poder "regresar" a versiones anteriores de nuestros archivos, para este fin se creó *Git.*

Git es un software de control de versiones distribuido que fue diseñado por *Linus Torvalds,* su principal objetivo es permitir mantener una cantidad considerable de código a una cantidad considerable de programadores de manera eficiente. Podemos hacer notar tres características principales:

- *Git* no almacena los archivos originales y por tanto no conserva una lista de los cambios que se haya realizado a estos archivos cada nueva versión. *Git* lo que hace es guardar una "instantánea" del estado de cada archivo en un momento específico. Si nada cambia no crea una nueva copia de cada archivo, sólo crea un referencia de los archivos originales.

- La eficiencia. Basándose en que cada programador guarda una copia completa del repositorio de manera local, *Git* no se ve afectado por la red (como pasa con otros controladores de versiones), por ende su velocidad no se ve afectada.

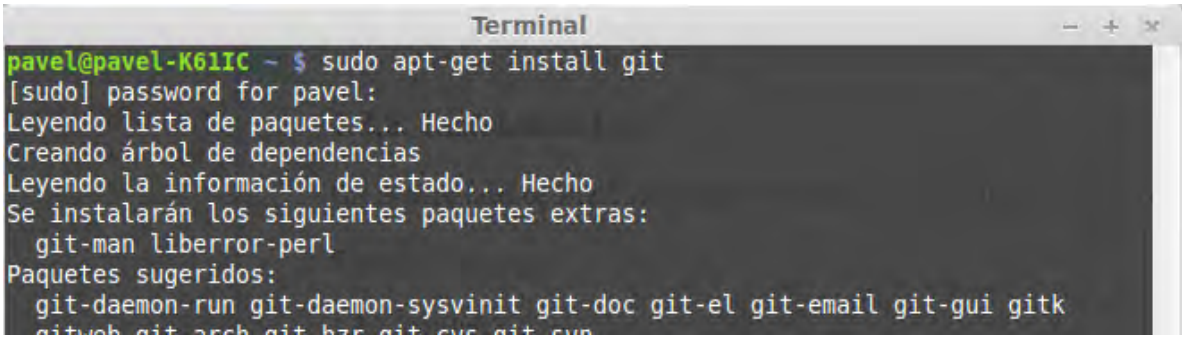
- *Git* tiene integridad, todo se verifica por una suma de comprobación antes de ser guardado y se verifica a partir del mismo momento mediante esta suma.

También cuenta con el manejo de ramas, una ayuda que nos simplifica el programar en equipo y sobretodo si no se tiene la presencia de todos los miembros del mismo. Ahora procederemos a instalarlo[10].

2.3.1. Instalación y uso de Git

Git es muy fácil de instalar en nuestro sistema *Linux*[11] (*Mint),* tecleamos:

```
$ apt-get install git
```



El uso de *Git* resulta ser sencillo y a la vez muy robusto, los comandos que usaremos son los más básicos pero resultan suficientes para llevar a cabo nuestro control de versiones. A continuación una pequeña introducción al uso de *Git*, una vez instalado debemos de añadir el nombre y correo de la persona que usa el repositorio:

```
$ git config --global user.name “Tu nombre”

$ git config --global user.email nuestro.email@ejemplo.com
```

Un comando que estaremos usando a menudo es `checkout` para no escribir todo el comando podemos usar un *alias* (en este caso sería “*co”*), se hace con el siguiente comando:

```
$ git config --global alias.co checkout
```

Ahora sí, para iniciar nuestro repositorio, debemos situarnos en la directorio raíz de nuestra aplicación y ejecutar el siguiente comando:

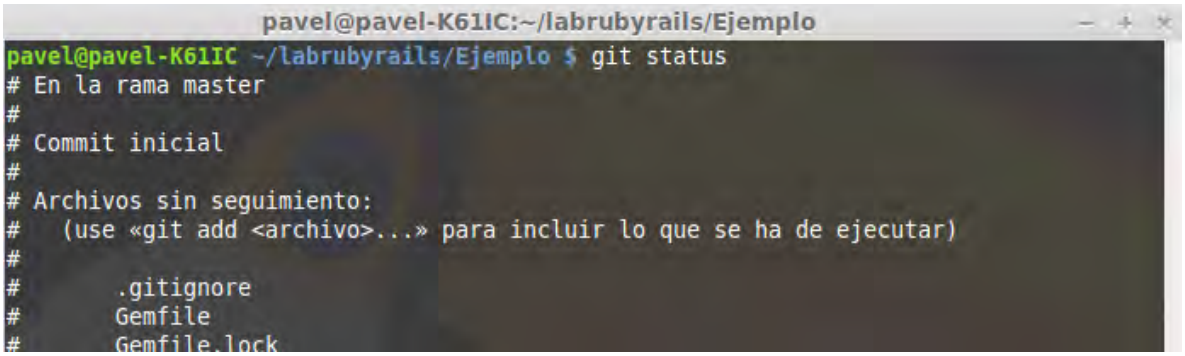
[10] Para probar *Git* de manera online, consultar el siguiente enlace: http://try.github.io/levels/1/challenges/1

[11] Otras distribuciones, consultar: http://git-scm.com/book/en/Getting-Started-Installing-Git

```
$ git init
```

Para verificar el estado de *Git* y qué archivos se añadieron o faltan añadir usamos:

```
$ git status
```



Git puede llegar a tener seguimiento de todos los archivos (si así lo especificamos, por el momento sólo inicia el repositorio), pero no necesariamente todos estos archivos nos son útiles. Para especificar qué archivos no necesitamos rastrear, usamos un archivo propio de *Git* llamado *gitignore*. Por default, *Rails* crea un archivo .**gitignore** al iniciarlo en una nueva aplicación (Figura 2.1).

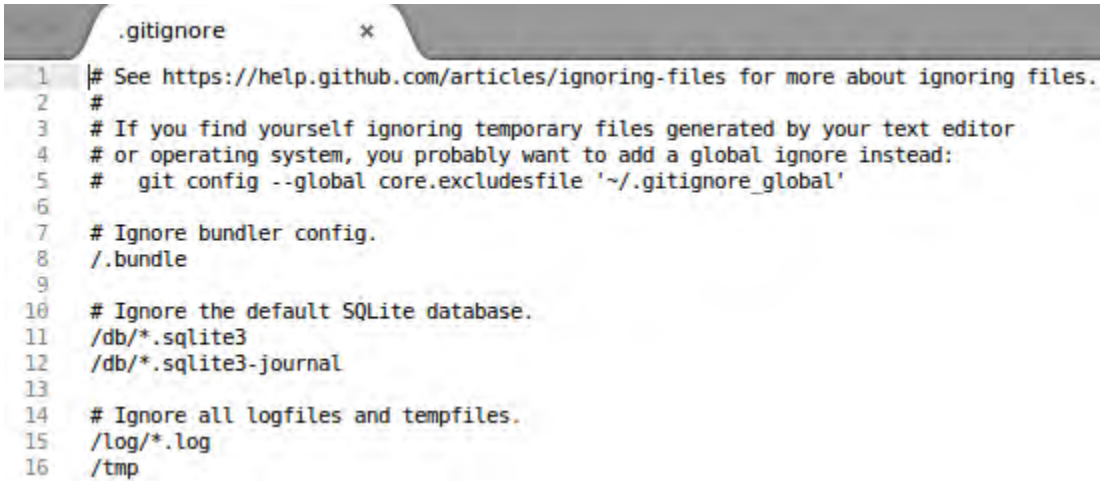


Figura 2.1. Archivo *gitignore*.

Cómo podemos observar, los comentarios llevan precedido el símbolo "#" (*hash*), estos comentarios nos dan una idea de cómo añadir los archivos o directorios a nuestro archivo *gitignore*, maneja comodines con un asterisco (*) y las rutas con un slash (/), notamos que los archivos ignorados son en parte logs de *Rails* y archivos del manejador de la base de datos así como archivos temporales.

2.3.2. Comandos Add y Commit

A continuación veremos los comandos que más usaremos con *Git*, serán los comandos responsables de añadir y de confirmar archivos así como los cambios en los mismos.

Add

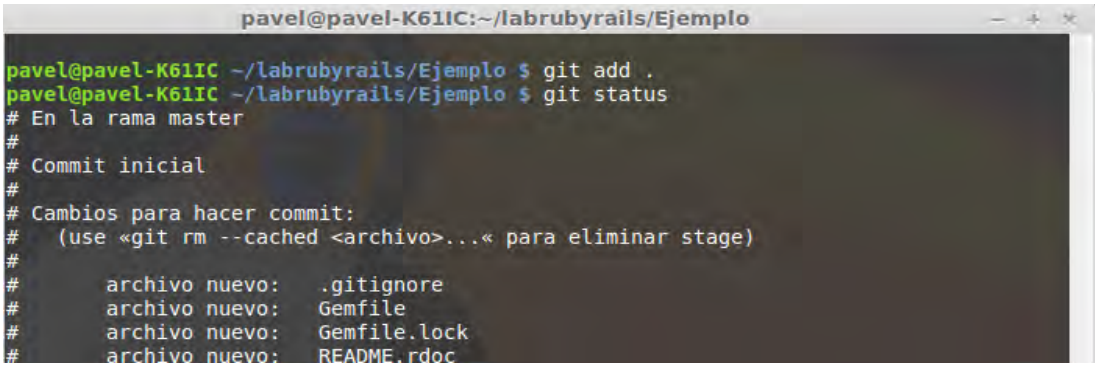
Ya una vez iniciado *Git* en nuestro directorio de trabajo y especificando qué archivos debe ignorar, damos paso a añadir los archivos que nos importa llevar un seguimiento si existe un cambio en ellos, para tal ejecutamos:

```
$ git add .
```

El punto (en *Linux*) representa el directorio sobre el que estamos, en este caso, el directorio de nuestra aplicación. *Git* hace este proceso de forma recursiva y añade todos los archivos (y cambios, de haberlos) de los subdirectorios. Podemos añadir archivos específicos también.

```
$ git add nombre-archivo
```

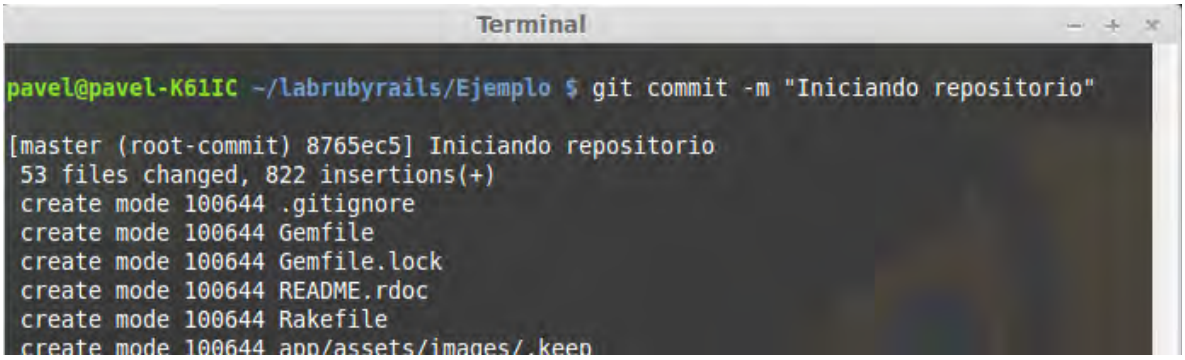
Todos los archivos añadidos, se van al área de "preparado", la cual contiene los cambios a archivos existentes o nuevos archivos que se añadan a nuestro proyecto; para ver los archivos que se encuentran en esta área podemos usar el comando `status:`



Commit

Una vez usado el comando `add`, *Git* nos hace notar: "Cambios para hacer *commit*", esto quiere decir que los archivos se añadieron al área de preparación pero hace falta confirmar los cambios. Para confirmarlos, se usa el comando `commit`.

```
$ git commit -m "Iniciando repositorio"
```

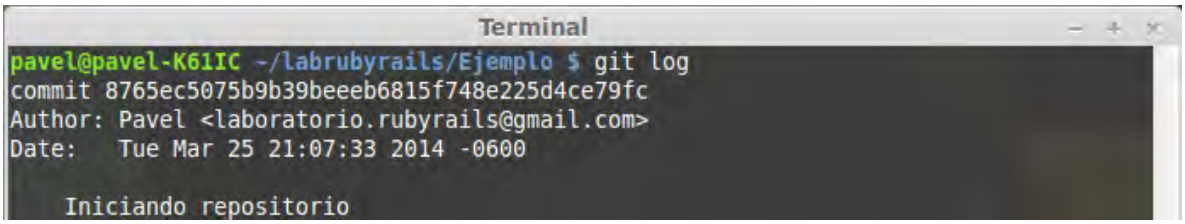


La bandera `-m` nos permite escribir un mensaje para los cambios efectuados, si no usamos dicha bandera, *Git* abrirá nuestro editor de texto por defecto para añadir el mensaje de confirmación. Por otro lado si queremos añadir cambios y confirmarlos casi al mismo tiempo podemos usar el siguiente comando (la bandera -am hace que el *commit* se aplique a los archivos con seguimiento pero no añade archivos nuevos en el directorio a *Git*):

```
$ git commit -am "Iniciando repositorio"
```

El mensaje que utilizamos para confirmar nos permite identificar los cambios que hayamos hecho en nuestro proyecto, para poder visualizarlos, escribimos en terminal:

```
$ git log
```



Por el momento se muestra un *commit* muy pobre en información, pero a medida que vayamos haciendo cambios en nuestro proyecto nos dará más detalle de todos éstos.

2.3.3. Deshaciendo cambios

Como cualquier humano, podemos cometer fallos a la hora de usar los comandos para añadir y confirmar pero *Git* nos da la opción de poder deshacerlos en nuestro proyecto. Esto es muy útil, más cuando nos llegamos a distraer o nos confundimos sobre qué modificamos.

Modificación de la última confirmación

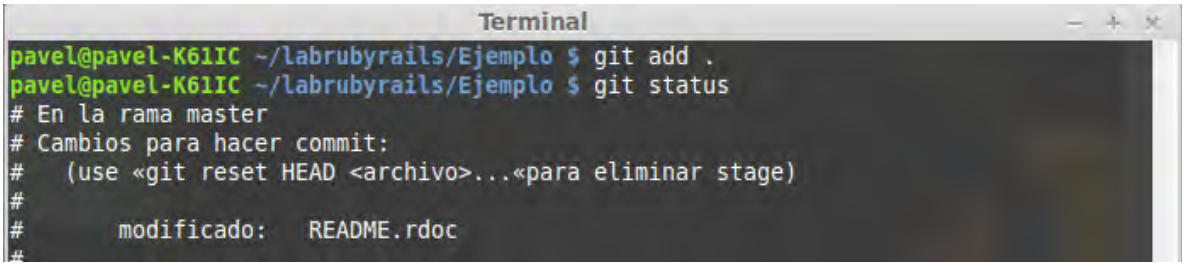
Si por alguna razón confirmamos los cambios sin haber revisado antes qué hicimos o simplemente el mensaje de confirmación no era el que deseábamos, podemos hacer un *commit* usando la bandera `--amend`:

```
$ git commit --amend
```

Deshacer cambios del área de preparación

Si deseamos sacar algún archivo del área de preparación, usamos un comando que el mismo *Git* nos sugiere después de usar el comando `add`:

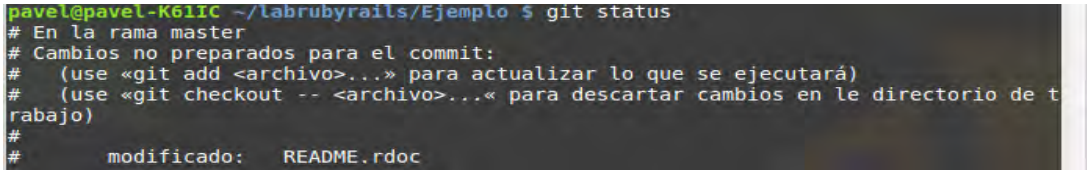
```
$ git reset HEAD <archivo>
```



Deshaciendo modificaciones en nuestros archivos

Ahora, si lo que queremos es revertir las modificaciones que hayamos hecho a algún archivo de nuestro proyecto, como por ejemplo el código fuente de un programa, podemos usar este otro comando de *Git*:

```
$ git checkout -- <archivo>
```



2.3.4. Repositorios remotos

Cuando trabajamos en equipo o simplemente nuestro lugar de trabajo no es constante y quizá en su momento, no contamos con el mismo equipo, podemos usar los repositorios remotos. Los repositorios remotos son versiones de nuestro proyecto pero alojadas en internet. Nos son útiles también cuando trabajamos en equipo, mandamos (comando `push`) y recibimos (comando `pull`) información de los cambios realizados a la aplicación..

Mostrando nuestros repositorios

Para saber qué repositorios tenemos en *Git*, se usa el comando `git remote`. Si añadimos la bandera `-v` podemos también ver la *URL* que tiene el repositorio remoto.

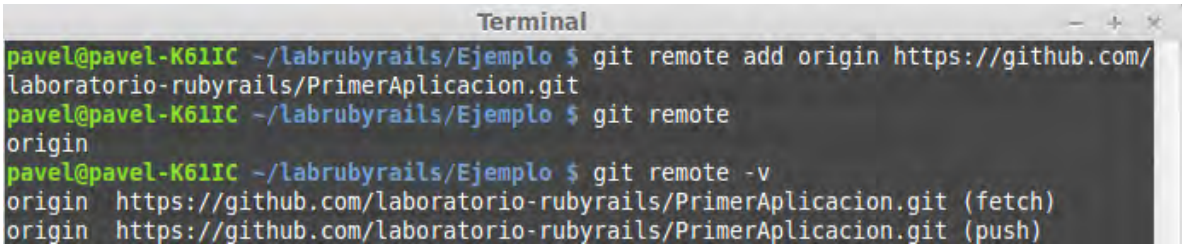
```
$ git remote
$ git remote -v
```

Si contamos con un repositorio, lo más seguro es que se llame *origin*, que es el nombre que usa *Git* como predeterminado para un repositorio, en nuestro caso aún no tenemos repositorios remotos pero en la siguiente sección veremos cómo agregarlos.

Añadir repositorios remotos

Para añadir un repositorio remoto se usa el comando `git remote add <nombre> <url>`, en nuestro caso usaremos un repositorio en línea llamado *Github.* Cuando nos registramos y creamos un nuevo repositorio (Figura 2.2), *Github* nos brinda una *URL* que podemos usar con el comando previo (esto es sólo un ejemplo, existen distintos repositorios en línea[12]):

```
$ git remote add <nombre> <url>
```



[12] Podemos consultar alternativas a *Github* en: http://www.genbeta.com/herramientas/cuatro-alternativas-a-sourceforge

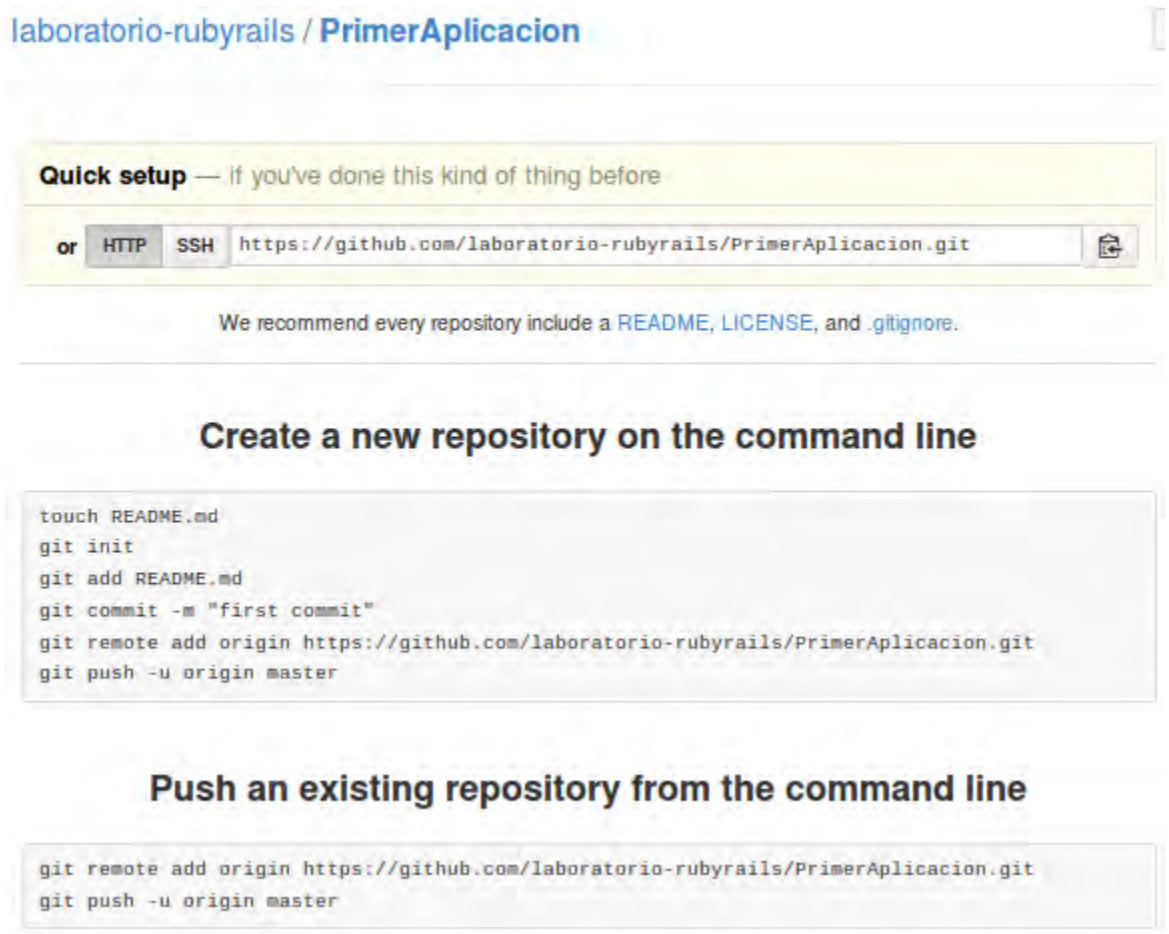


Figura 2.2. Repositorios remotos en *Github*.

Enviando nuestros repositorios

Cuando nuestro proyecto esté listo para ser enviado a nuestro repositorio remoto, usaremos el comando: `git push <nombre-remoto> <nombre-rama>`. Más adelante veremos el tema de las ramas en *Git* sólo basta por ahora que la rama *master* es la rama principal. En este caso enviaremos nuestro primer proyecto, con la rama *master* a nuestro repositorio *origin:*

```
$ git push origin master
```

```
pavel@pavel-K61IC ~/labrubyrails/Ejemplo $ git push origin master
Username for 'https://github.com': laboratorio-rubyrails
Password for 'https://laboratorio-rubyrails@github.com':
Counting objects: 60, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (49/49), done.
Writing objects: 100% (60/60), 14.31 KiB | 0 bytes/s, done.
Total 60 (delta 2), reused 0 (delta 0)
To https://github.com/laboratorio-rubyrails/PrimerAplicacion.git
 * [new branch]      master -> master
```

Eliminando repositorios

Para eliminar repositorios en *Git* ejecutamos el siguiente comando:

```
$ git remote rm <nombre-remoto>
```

Ramas.

El uso de ramas o *branching* es de las características más importantes de *Git* frente a otros sistemas de control de versiones. Las ramas nos ayudan a que el desarrollo del proyecto sea ágil cuando consta de distintos módulos (parte gráfica de la aplicación, sesiones, login, etcétera) o de un equipo de varios miembros.

2.3.5. Operaciones con ramas

Crear ramas

Para crear una rama ejecutamos el siguiente comando:

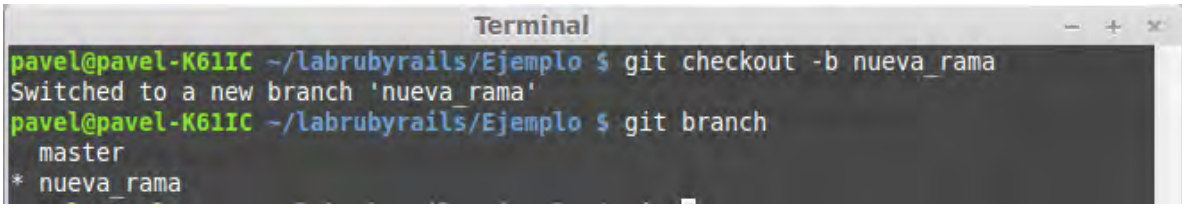
```
$ git branch <nueva-rama>
```

Para cambiar de la rama principal *master* a una rama nueva:

```
$ git checkout <nueva-rama>
```

O simplemente usar el siguiente atajo para crear y cambiar de rama:

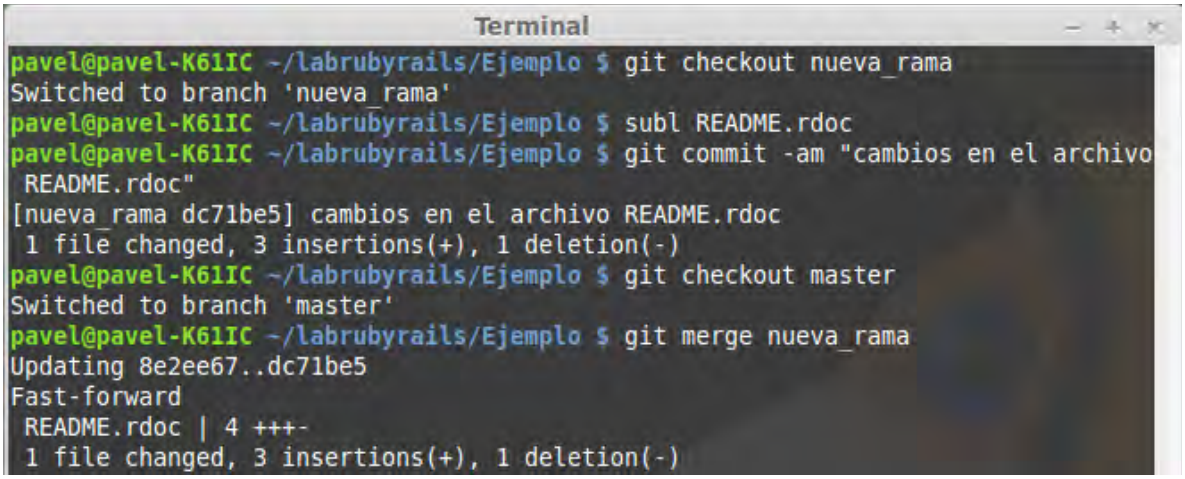
```
$ git checkout -b <nueva_rama>
```



Unir cambios (merge)

Una vez hecho cambios en archivos del proyecto, podemos añadirlos a los archivos de la rama principal *master*, para hacer esto se usa el comando `merge`, primeramente debemos de cambiarnos a la rama *master* y luego hacer el *`merge`*.

```
$ git checkout master

$ git merge <nueva_rama>
```



Eliminar rama

Para eliminar una rama, simplemente debemos de ejecutar los siguiente:

```
$ git branch -d <nueva_rama>
```

2.4. GitHub

Github es un repositorio en línea para proyectos creados por la comunidad de desarrolladores. Es el complemento perfecto para *Git* y para nuestro laboratorio, podemos subir nuestro repositorio local a *Github* a través de los repositorios remotos que maneja *Git*. Existen dos versiones de *Github*, una de pago y una gratis, para los fines de nuestro laboratorio, con la versión gratis será más que suficiente.

Sólo debemos registrarnos en *www.github.com* y añadir el repositorio remoto a *Git* como hemos visto previamente.

2.5. Base de Datos

Para nuestra aplicación se usará *SQLite* como manejador de las bases de datos, esto es porque *SQLite* cuenta con algunas ventajas pero sólo para cuestiones de desarrollo y pruebas de una aplicación, más no de producción. En cualquier caso el alumno puede trabajar con algún otro manejador de bases de datos (*MySQL*, *PostgreSQL*).

Las ventajas que ofrece *SQLite* son:

- Estable: *SQLite* es compatible con conjunto *ACID*[13] (Atomicidad, Consistencia, Aislamiento y Durabilidad).

- Tamaño: *SQLite* cuenta con una poca memoria y sólo una biblioteca es necesaria para acceder nuestra base de datos.

- Portabilidad: se puede ejecutar en cualquier sistema operativo y las bases de datos pueden ser portadas sin administración o configuración extra.

- Costo: *SQLite* al ser de dominio público, lo puede utilizar cualquier persona para cualquier propósito sin costo.

Más adelante, cuando hagamos nuestra primer aplicación de ejemplo, se realizará la configuración de los manejadores de bases de datos *MySQL* y *PostgreSQL* esto es así porque para poder hacer dichas configuraciones debemos crear primero un proyecto y una vez creado ya podemos empezar a modificar los archivos correspondientes, que en este caso el primero sería el de la base de datos.

[13] Si se requiere más de información: http://es.wikipedia.org/wiki/ACID

2.6. Instalación en Windows

La instalación del software anterior se instala de manera sencilla en lo que respecta a *Windows*. Debemos descargar el paquete llamado *RailsInstaller* de la liga http://railsinstaller.org/en, el software que contiene el paquete es el siguiente[14]:

- *Ruby 1.9.3*
- *Rails 3.2*
- *Bundler.*
- *Git.*
- *SQLite.*
- *TinyTDS.*
- *SQL Server Support.*
- *DevKit.*

2.7. Conociendo Ruby on Rails

Ahora pasamos a la acción, crearemos nuestra primer aplicación la cual será muy sencilla, de hecho no modificaremos nada de lo que *Ruby on Rails* crea en el directorio de trabajo. Para empezar crearemos un directorio especifico para nuestros proyectos en rails, luego usaremos el comando `rails new` que genera un esqueleto en nuestro directorio, que consta de varios archivos y directorios (así como el archivo ***gitignore*** del que hablamos previamente).

```
$ mkdir lab_rails

$ cd lab_rails

$ rails new primer_aplicacion
```

[14] El contenido de dicho paquete puede variar para el momento en que se lea este documento.

```
Terminal
pavel@pavel-K61IC ~ $ mkdir lab_rails
pavel@pavel-K61IC ~ $ cd lab_rails/
pavel@pavel-K61IC ~/lab_rails $ rails new primer_aplicacion
      create
      create  README.rdoc
      create  Rakefile
      create  config.ru
      create  .gitignore
      create  Gemfile
```

Obviamos que *Rails* crea más archivos debajo de lo que muestra la imagen anterior (Figura 2.3). Damos pie a explicar la mayoría de ellos:

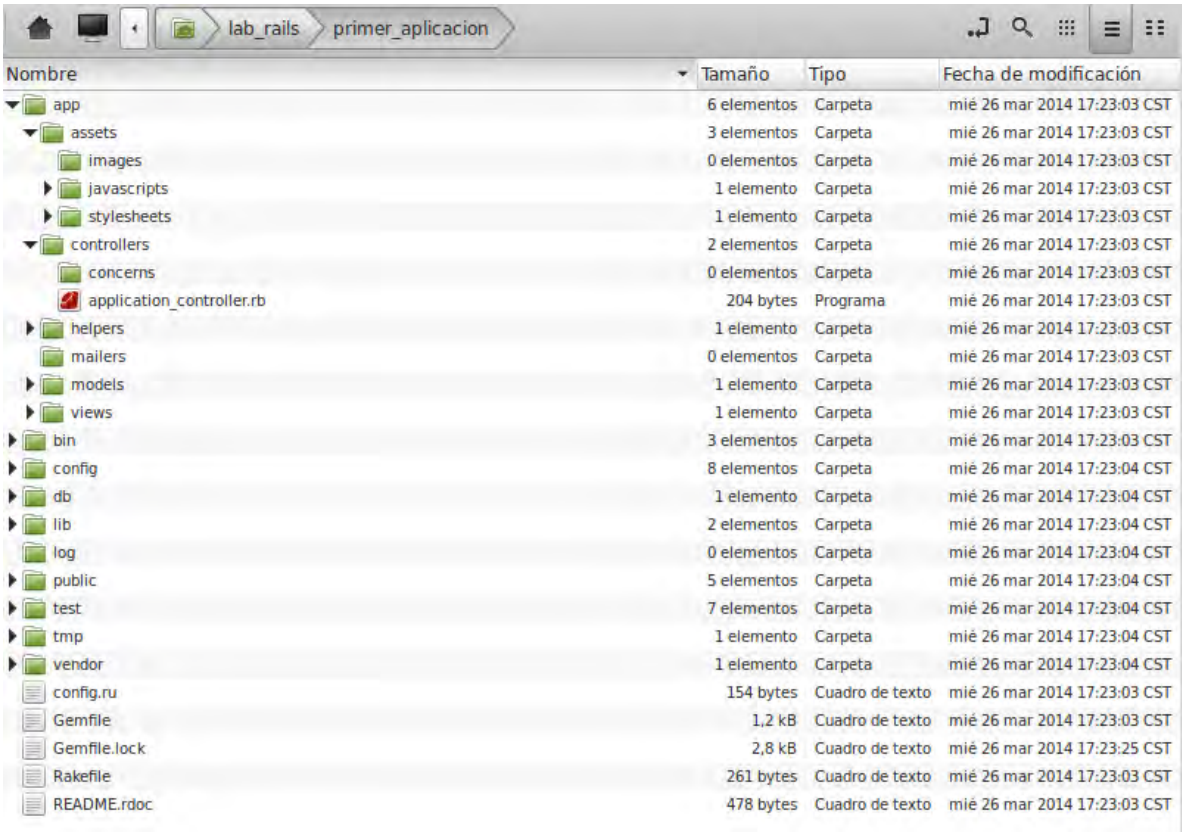


Figura 2.3. Directorios del nuevo proyecto.

Tabla 2.1. Archivos y directorios de *Ruby on Rails*.

Directorios/Archivos	**Función**
app	Este directorio es el núcleo de nuestra aplicación. Los subdirectorios contienen principalmente vistas, controladores y los modelos (lógica).
app/assets	Contiene los recursos de la aplicación (hojas de estilo CSS, archivos *Javascript* e imágenes).
app/controller	Aquí se encuentra todas las clases de la aplicación (controllers).
app/models	Contiene las clases que interactúan con los datos almacenados en una base de datos.
app/views	Contiene todas las vistas (parte visual e interactiva de la aplicación al usuario) de nuestra aplicación,
app/view/layouts	Aquí se guardan las plantillas que son parte de esquemas a utilizar en las vistas de nuestra aplicación.
app/helpers	Clases "*helper*" que complementan a clases de tipo *model*, *view* y *controller*.
bin/	Contiene archivos binarios que son ejecutables.
config/	Dentro están los archivos de configuración de la aplicación.
db/	Contiene archivos referentes a la base de datos.
doc/	Documentación.
lib/	Bibliotecas para la aplicación.
lib/assets	Contiene recursos de bibliotecas externas (hojas de estilo CSS, archivos *Javascript* e imágenes).

Directorios/Archivos	Función
log/	Dentro los archivos *log* de errores.
public/	Aquí se encuentra todo lo accesible a un nivel público, por lo general, páginas de error.
bin/rails	Desde aquí podemos generar código, abrir sesiones o simplemente iniciar el servidor local.
test/	Pruebas de la aplicación.
tmp/	Cómo intuimos, archivos temporales de la aplicación.
vendor/	Directorio que contiene las gemas, plugins o recursos de terceros.
README.rdoc	Archivo que contiene los detalles de nuestra aplicación.
Rakefile	Un archivo similar al *Makefile*[15] de *Unix*, ayuda a construir, empaquetar y para poder probar el código
Gemfile	Este archivo contiene las gemas requeridas por la aplicación.
Gemfile.lock	Archivo que contiene una lista de las gemas usadas y que permite que al clonar la aplicación, se mantenga la misma versión de cada gema.
.gitignore	Como vimos anteriormente, este archivo indica a *Git*, los archivos que debe ignorar en nuestro control de versiones.

Ya que se generó el esqueleto de nuestra aplicación, podemos añadir por ejemplo, el directorio doc/ (línea 14, Figura 2.4) al archivo *gitignore:*

[15] Más información sobre el archivo *Makefile*: http://es.wikipedia.org/wiki/Make

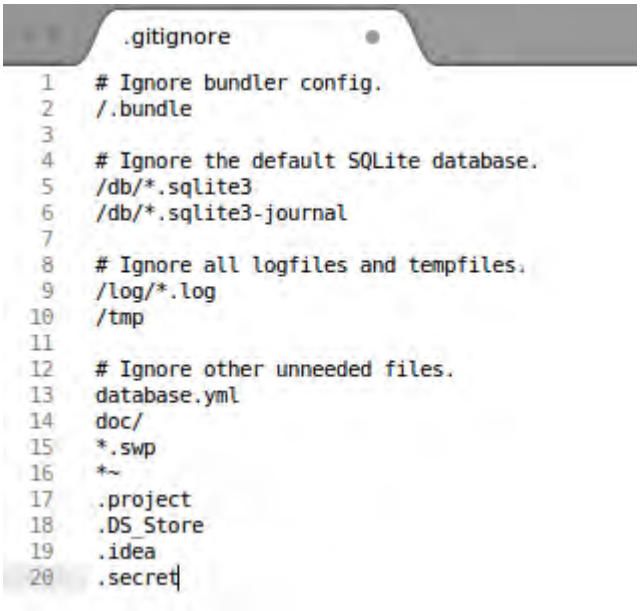


Figura 2.4. Archivo *gitignore* aumentado.

Ahora lo que sigue es especificar las *gemas* necesarias. Estas *gemas* podemos encontrarlas en el archivo *Gemfile*. Para los fines de nuestro curso deberemos modificar este archivo con las versiones especificas para cada una[16]:

- *gem 'uglifier', '>= 2.2.1'*
- *'coffee-rails', '~> 4.0.1'*
- *gem 'jquery-rails', '3.0.4'*
- *gem 'turbolinks', '1.1.1'*
- *gem 'sdoc', '0.3.20', require: false*

Añadimos también a la *gema sqlite3* a un entorno de desarrollo, líneas 7-9, Figura 2.5 (si cambiamos de manejador de bases de datos con los pasos que se explican más adelante, la gema que añadimos a desarrollo será la correspondiente al manejador que usemos), esto es para evitar futuros conflictos con aplicaciones que tengan acceso a la base de datos de nuestro proyecto.

Pueden generarse algunas dudas sobre la sintaxis del archivo *Gemfile*, por aclarar dos en especial, añadimos "*>=*" para especificar a *Bundler* que necesitamos la versión que sea mayor o igual a la versión especificada, por ejemplo, la gema *uglifier* que debemos tener

[16] Las versiones de las *gemas* descritas corresponden a la fecha en la que se escribe este documento puede que se hayan actualizado a una versión más estable para cuando el alumno lea este documento. Se recomienda verificar cada *gema*.

debe ser la versión *2.2.1* o mayor (*3.0*, por citar alguna versión superior) pero no menor. Si optamos por usar *"~>"*, *Bundler* instalará una versión de la gema con cambios menores pero no cambios mayores, por ejemplo, citando de nuevo la gema *uglifier, Bundler* instalará una versión *2.2.2* o *2.2.3*, pero no instalará la versión *3.0*.

Si añadimos la bandera `--without production` a *Bundler*, anulamos la instalación local de cualquier *gema* de producción.

```
1  source 'https://rubygems.org'
2
3  # Bundle edge Rails instead: gem 'rails', github: 'rails/rail
4  gem 'rails', '4.0.4'
5
6  # Use sqlite3 as the database for Active Record
7  group :development do
8    gem 'sqlite3'
9  end
10 # Use SCSS for stylesheets
11 gem 'sass-rails', '~> 4.0.2'
12
13 # Use Uglifier as compressor for JavaScript assets
14 gem 'uglifier', '>= 2.2.1'
15
16 # Use CoffeeScript for .js.coffee assets and views
17 gem 'coffee-rails', '~> 4.0.1'
18
19 # See https://github.com/sstephenson/execjs#readme for more s
20 # gem 'therubyracer', platforms: :ruby
21
22 # Use jquery as the JavaScript library
23 gem 'jquery-rails', '3.0.4'
24
25 # Turbolinks makes following links in your web application fa
26 gem 'turbolinks', '1.1.1'
27
28 # Build JSON APIs with ease. Read more: https://github.com/ra
29 gem 'jbuilder', '~> 1.2'
30
31 group :doc do
32   # bundle exec rake doc:rails generates the API under doc/ap
33   gem 'sdoc', '0.3.20', require: false
34 end
```

Figura 2.5. *Gemfile* modificado.

Si lo que queremos es cambiar el manejador de la base de datos *SQLite* por algún otro como *MySQL* (para *PostgreSQL* sería análogo) debemos tener en un archivo:

- **database.yml**: todo lo referente a la configuración de la base de datos se almacena en este archivo. Podemos notar que *Rails* maneja tres tipos de bases de datos: desarollo, pruebas y producción (Figura 2.6).

database.yml

```
# SQLite version 3.x
#   gem install sqlite3
#
#   Ensure the SQLite 3 gem is defined in your Gemfile
#   gem 'sqlite3'
development:
  adapter: sqlite3
  database: db/development.sqlite3
  pool: 5
  timeout: 5000

# Warning: The database defined as "test" will be erased and
# re-generated from your development database when you run "rake".
# Do not set this db to the same as development or production.
test:
  adapter: sqlite3
  database: db/test.sqlite3
  pool: 5
  timeout: 5000

production:
  adapter: sqlite3
  database: db/production.sqlite3
  pool: 5
  timeout: 5000
```

Figura 2.6. **database.yml**.

Cómo observamos, aparecen las tres bases de datos (en inglés, *development, test, production*), así como su respectiva configuración. Lo que debemos hacer entonces es modificar la información del archivo **database.yml** por la información correspondiente al manejador que deseemos utilizar.

MySQL.

No se dará detalle de la instalación de este manejador (ni de *PostgreSQL*), ya que el alumno debió haber llevado previo a este curso, la materia de *Fundamentos de Bases de Datos* o en otro caso, haber visto tal instalación en la ayudantía del curso.

Debemos añadir la gema correspondiente del manejador en nuestro archivo *Gemfile*, en este caso la *gema* "*mysql2*[17]" . Luego, la instalamos con *Bundler*:

```
$ gem 'mysql2'

$ bundle install
```

[17] La documentación referente a *mysql2* se encuentra en: http://rubydoc.info/gems/mysql2/0.3.15/frames

```
# Use sqlite3 as the database for Active Record
#group :development do
#  gem 'sqlite3'
#end

group :development do
  gem 'sqlite3'
end
```

Ahora, nos conectamos al manejador y creamos las tres bases de datos que *Rails* utiliza (*development, test* y *production*):

```
mysql> CREATE DATABASE lab_development;
Query OK, 1 row affected (0.00 sec)

mysql> CREATE DATABASE lab_test;
Query OK, 1 row affected (0.00 sec)

mysql> CREATE DATABASE lab_production;
Query OK, 1 row affected (0.00 sec)
```

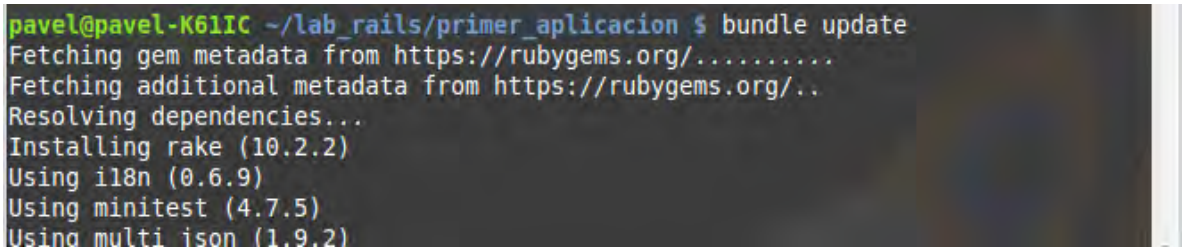
Por último modificamos el archivo **database.yml** con la información de nuestras respectivas bases recién creadas:

database.yml

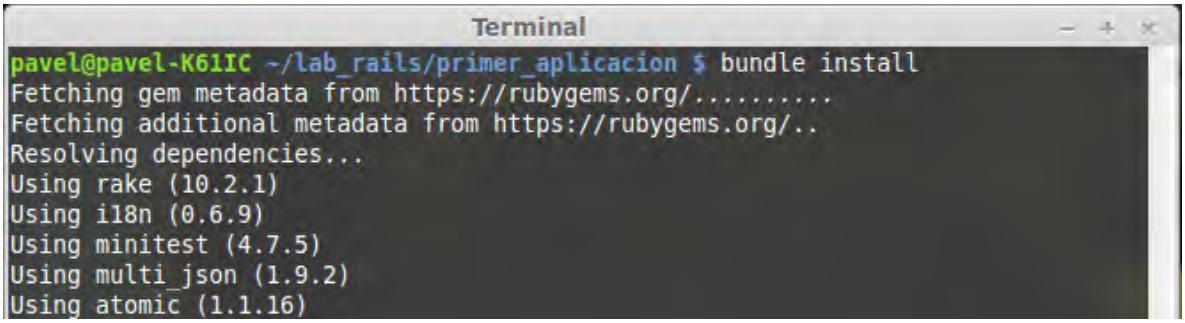
```
10    pool: 5
11    username: root
12    password: root
13    host: 127.0.0.1
14    timeout: 5000
15
16  # Warning: The database defi
17  # re-generated from your dev
18  # Do not set this db to the
19  test:
20    adapter: mysql2
21    encoding: utf8
22    database: lab_test
23    pool: 5
24    username: root
25    password: root
26    host: 127.0.0.1
27    timeout: 5000
28
29  production:
30    adapter: mysql2
31    encoding: utf8
32    database: lab_production
33    pool: 5
34    username: root
35    password: root
36    host: 127.0.0.1
37    timeout: 5000
```

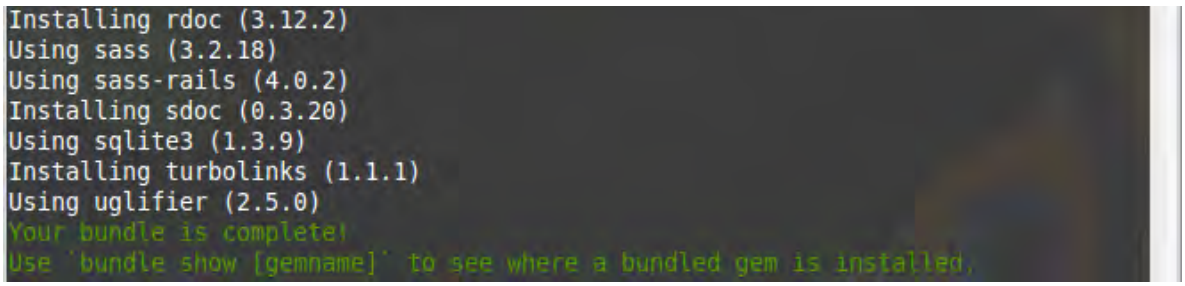
Ya teniendo el archivo *Gemfile* modificado y el archivo **database.yml** con el manejador que nos guste, debemos instalar las *gemas*. *Bundler*[18] será el encargado de instalarlas, para ejecutarlo se usan los comandos `bundle, update` y `bundle install`[19]:

```
$ bundler update
```




```
$ bundle install
```





Si todo sale bien podemos ya probar nuestra aplicación haciendo uso del comando `rails server`, que dará inicio a un servidor local (podemos también usar el atajo, `rails s`). Veremos nuestra aplicación en la dirección que muestra la terminal: *localhost:3000* (Figura 2.7).

[18] *Bundler* es un gestor de gemas para aplicaciones en *Rails*. El archivo *Gemfile* nos muestra a través de las gemas que usaremos, una parte de la sintaxis de *Bundler*.

[19] Si no se especifica una versión en el archivo *Gemfile*, *Bundler* instalará la última versión de dicha gema.

```
$ rails server

$ rails s
```

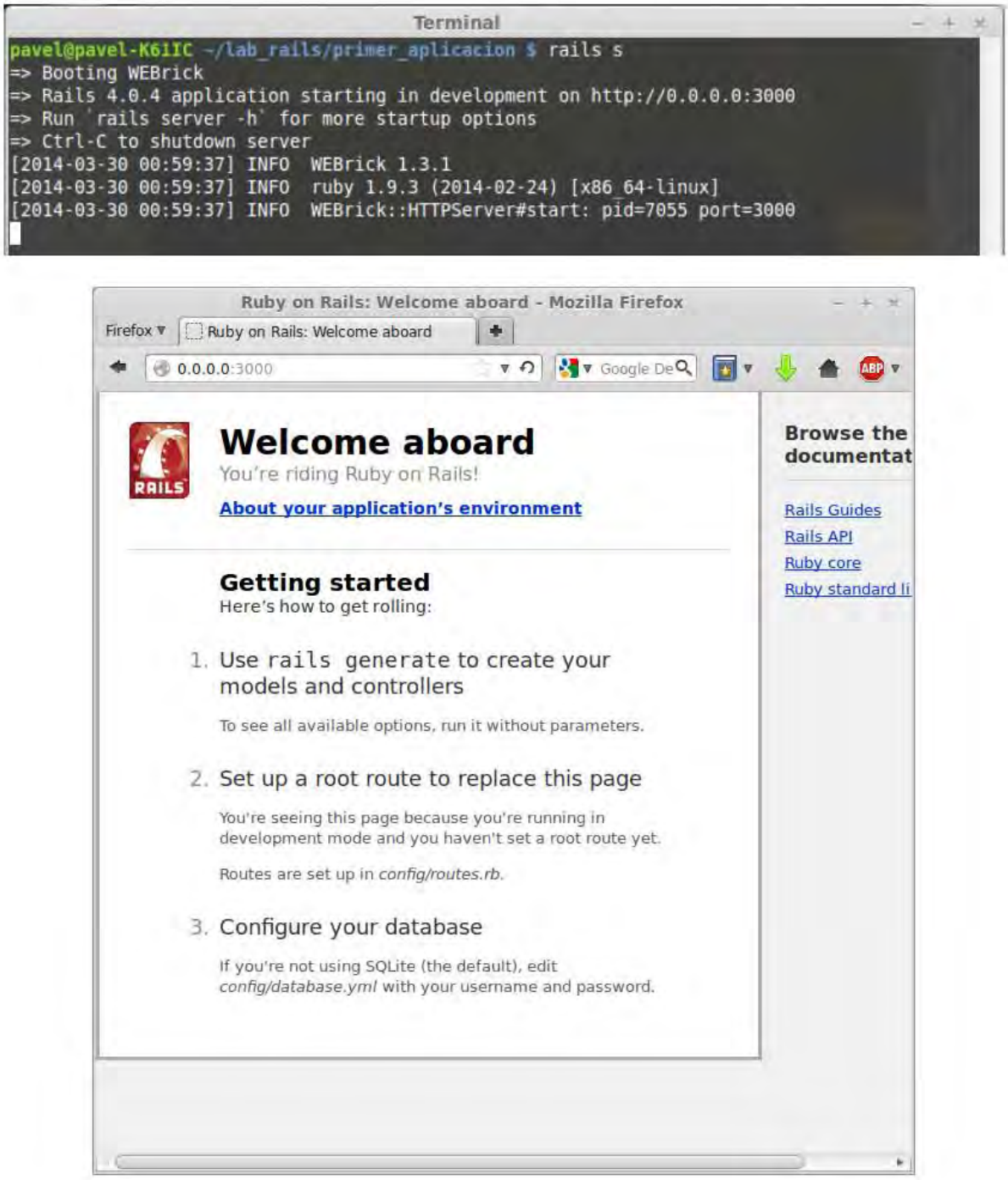


Figura 2.7. Página principal de nuestra aplicación.

Si tenemos problemas al iniciar el servidor, podemos resolverlos instalando los siguientes paquetes.

```
$ sudo apt-get install libxslt-dev libxml2-dev libsqlite3-dev
```

Si el problema persiste y la terminal nos dice algo sobre *"ExecJS and could not find a JavaScript runtime"*, ejecutamos lo siguiente:

```
$ sudo apt-get install node.js
```

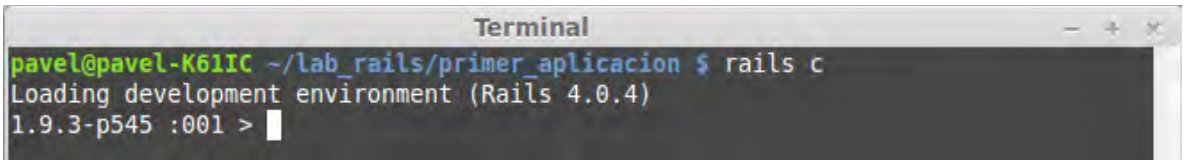
Lo anterior es porque necesitamos un interprete de *Javascript*, que en este caso será NodeJS.

Capítulo 3: Conociendo Ruby

3. Desarrollo

Hemos visto como se configura el ambiente para trabajar con *Ruby,* ahora toca turno a trabajar con él. Para empezar a usar *Ruby*, haremos uso del *Interactive Ruby Shell* (*IRB*) que al ser la base de la consola en *Rails*, vendrá con todas las funciones de *Ruby*. Para iniciar abrimos una terminal y tecleamos el comando `rails console` (o el atajo `rails c`), al hacerlo la consola de *Rails* se abrirá y estaremos listos para jugar con *Ruby*.:

```
$ rails console

$ rails c
```



Si queremos trabajar con *Ruby* pero no queremos que se modifique nada después de usarlo, existe el modo *sandbox,* que al cerrar el *IRB* hace un *rollback* de todo lo que hayamos modificado. Lo usaremos más adelante, cuando trabajemos con las bases de datos.

```
$ rails c --sandbox
```

La consola por omisión, inicia en el modo de desarrollo (*development environment*). Ruby cuenta con otros dos modos para la consola: *test* y *production*. Para los fines de este capítulo usaremos el ambiente de desarrollo.

Si por el contrario, queremos hacer uso de archivos (no usar el *intérprete*) para trabajar con *Ruby* lo podemos hacer guardando el contenido de nuestro programa en archivo con extensión *.rb*, para cargarlo debemos de ejecutar el comando:

```
$ ruby archivo.rb
```

Antes de empezar, si surge alguna duda sobre el uso de *Ruby*, se recomienda leer el *API*[20] para una explicación más detallada.

[20] *API* de *Ruby*: http://ruby-doc.org/core-2.0/

3.1. Conociendo el lenguaje de programación Ruby

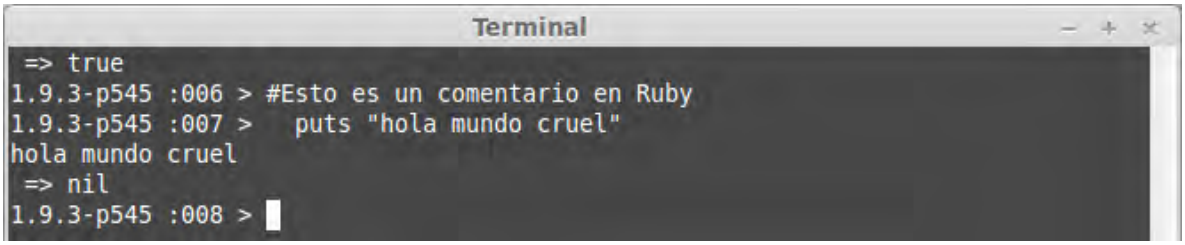
En *Ruby*, todo es un objeto, los números, los caracteres, etcétera. Esto es así porque *Ruby* tiene sus bases en clases, dichas clases se encargan de proporcionar a nuestros datos distintos métodos (clases y métodos en *Ruby* los veremos más adelante) para interactuar con ellos y utilizarlos para distintos fines. Veremos en este apartado parte de la sintaxis y de la semántica que *Ruby* maneja, así como pequeños ejemplos que ilustrarán las ideas.

Cabe aclarar que no se entrará en detalles con la explicación de los conceptos del paradigma orientado a objetos (condicionales, definición de conceptos como clase, método, literales, constantes, entre otros), ya que tales conceptos se estudian con más detalle en semestres anteriores al de Ingeniería de Software . Algo que debemos tener presente desde este momento es que no debemos de usar punto y coma (;) para especificar el fin de una instrucción en nuestro programa.

3.1.2. Comentarios

Para añadir un comentario a nuestro programa, basta con agregar el símbolo "*#*" (justo como los comentarios en *Git).* Por ejemplo:

```
>> #Esto es un comentario en Ruby

>> puts "hola mundo cruel"

"hola mundo cruel"
```



El método `puts` imprime en pantalla la cadena entre comillas que le sigue.

3.1.3. Números

Como cualquier lenguaje de programación, contamos con números y sus respectivas operaciones (y condicionales) en *Ruby*. Dichas operaciones podemos listarlas como sigue[21]:

[21] Se muestran las operaciones más comunes, para saber qué otras operaciones podemos usar, consultar: http://rubylearning.com/satishtalim/numbers_in_ruby.html

Operaciones:

- + (suma).
- - (resta).
- / (división)
- * (multiplicación)
- % (módulo).

Comparadores:

- < (menor que).
- > (mayor que).
- <= menor o igual que).
- >= (mayor o igual que).

Ejecutemos el siguiente programa (Código 3.1) con la consola de *Rails* [22]:

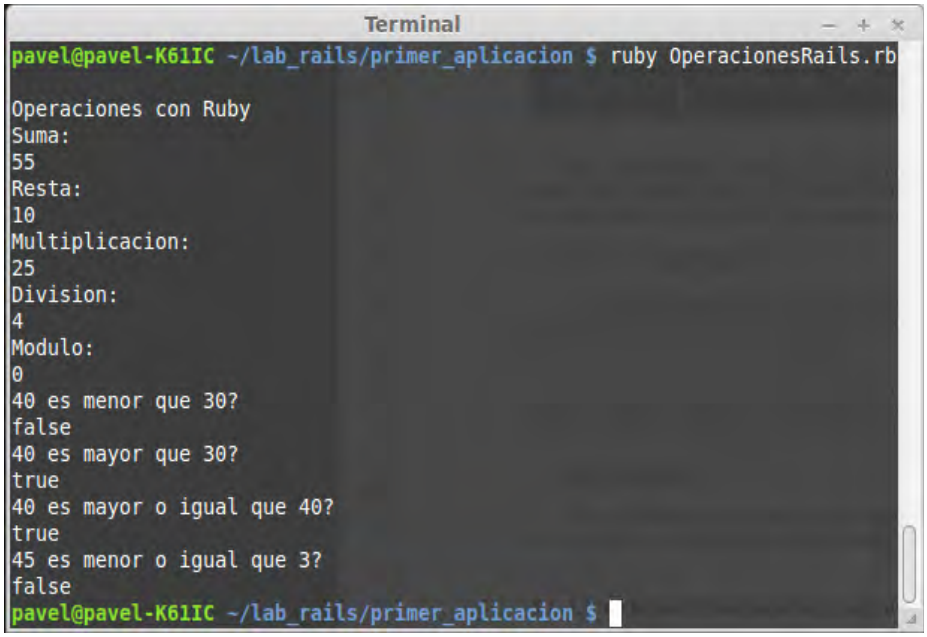
OperacionesRails.rb

OperacionesRails.rb

```
1  # encoding: utf-8
2
3  puts "Operaciones con Ruby"
4
5  puts "Suma: ", 25 + 30
6  puts "Resta: ", 40 - 30
7  puts "Multiplicación: ", 5 * 5
8  puts "División: ", 40 / 10
9  puts "Módulo: ", 100 % 20
10 puts "¿40 es menor que 30?", 40 < 30
11 puts "¿40 es mayor que 30?", 40 > 30
12 puts "¿45 es mayor o igual que 40?", 40 >= 40
13 puts "¿45 es mayor o igual que 3", 40 <= 3
```

Código 3.1. **OperacionesRails.rb**

[22] De ahora en adelante, algunos de los programas mostrados llevaran como primer línea el comentario: `#encoding: utf-8`, para evitar problemas de codificación.



Los números en *Ruby* no pueden contener comillas o apostrofes, en su lugar podemos usar el carácter, guión bajo ("_"). Nos puede ser útil por ejemplo para visualizar de mejor manera, números muy grandes:

```
total = 188_000_000
```

Los números enteros se catalogan como *Integers*, si queremos usar números decimales, podemos usar los números *flotantes* (*float*) para realizar las mismas operaciones descritas arriba.

3.1.4. Variables

Las variables como sabemos nos sirven para "almacenar" información (regularmente cadenas y números) en la memoria de nuestra computadora. Nos permite reutilizar estos valores en varias partes de nuestro código. En *Ruby* las reglas que deben seguir las variables son:

- Las variables locales se conforman de una letra minúscula o de un guión bajo seguida de una letra ya sea minúscula o mayúscula, estas variables sólo existen dentro de un bloque y su uso se restringe al mismo.

- Una variable de instancia u objeto (clases y objetos los veremos más adelante) comienza con el símbolo arroba (@), seguido de cualquier letra minúscula o mayúscula, puede ser usado por cualquier método de la clase a la que pertenece.

Este tipo de variables cuenta con tres métodos de acceso (encapsulamiento) que son:

- `attr_reader`: permite el acceso de sólo lectura de la variable.
- `attr_writer`: permite la escritura de la variable.
- `attr_accesor`: permite la lectura y escritura de la variable.

- Una constante denota un valor que no cambia su valor y se escribe siempre con todas sus letras en mayúscula.
- Las variables globales se denota con un signo de dólar antes de su nombre ($) y es accesible por cualquier clase. Su uso debe ser precavido.

Es bien sabido que no podemos usar palabras clave (*keywords*[23]) como nombres de variable, es importante tenerlo en cuenta (justo como en *Java*). Por ejemplo:

```
aves = 10
nombre_paterno = “Cedillo”
star_wars = “Chewbaka”
suma_dos_enteros = 2+2
mil_aves = aves * 100
```

son nombres válidos. Variables o constantes que incluyan las siguientes palabras reservadas no podrán ser válidas:

alias	and	break	case
class	def	defined?	do
else	elsif	end	ensure
false	for	if	in
module	next	nil	not
or	redo	rescue	retry
return	self	super	then
true	undef	unless	until
when	while	yield	_FILE_
LINE			

[23] Se pueden consultar las palabras clave en la documentación oficial de *Ruby:* http://ruby-doc.org/docs/keywords/1.9/

3.1.5. Cadenas

El uso de cadenas (*strings* en inglés) es imprescindible para cualquier aplicación web que hagamos sin importar el lenguaje que usemos. Probemos las siguientes instrucciones en la consola de *Rails*:

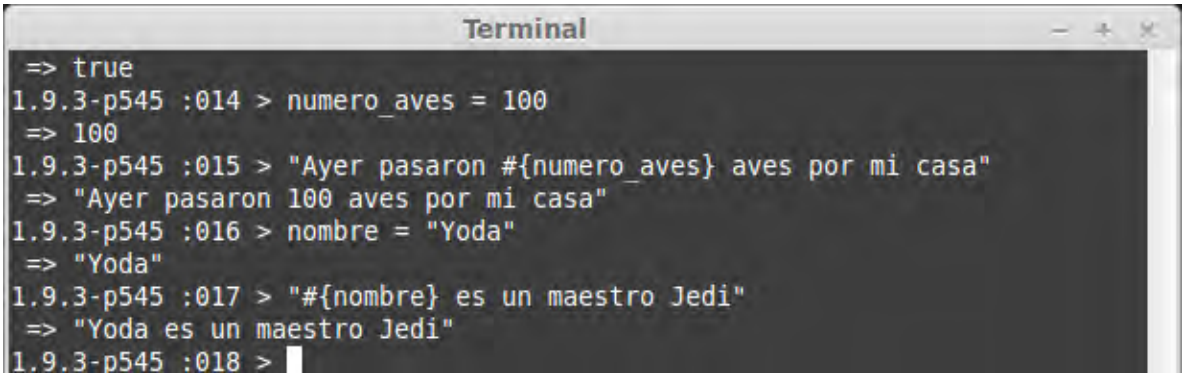
```
$ “”

$ “Hola”

$ “Hola” + “ Mundo Cruel”
```

Como vemos, la consola imprime en pantalla el resultado de haber evaluado el contenido que se encuentra entre comillas (*“ ”*). En el último ejemplo se concatenan dos cadenas, observamos que no difiere para nada de la forma en la que concatenamos dos cadenas en *Java* en el que hacemos uso del operador + (plus).

Otra manera del uso de cadenas en oraciones, es construirlas a través de la *interpolación*[24] de cadenas, haciendo uso de la sintaxis `#{}`:

```
>> numero_aves = 100
>> Ayer pasaron #{numero_aves} aves por mi casa.

>> nombre = “Yoda”
>> “#{nombre}” es un maestro Jedi
```

[24] La *interpolación* se refiere al proceso de insertar la salida o resultado de una expresión dentro de una cadena. Como vemos en el ejemplo, también se pueden evaluar operaciones numéricas.



Símbolos

Un símbolo resulta ser el objeto más básico en *Ruby*: tiene un nombre y una identidad o *ID*[25]. Dado un símbolo, nos referimos a un sólo objeto en todo nuestro programa. Esto hace que sean más eficientes que las cadenas ya que dos cadenas llamadas igual se refieren a dos objetos distintos y eso en memoria y tiempo, resulta ser costoso.

Podemos hacer notar dos situaciones de lo anterior:

- Si el contenido del objeto que usemos es importantes deberemos usar un *string*.

- Si por el contrario, lo importante es la *ID* del objeto, usaremos un símbolo. Por eso como veremos adelante, se usan como índices o claves para los hashes.

Un símbolo tiene como sintaxis: dos puntos, seguidos de un identificado. Por ejemplo:

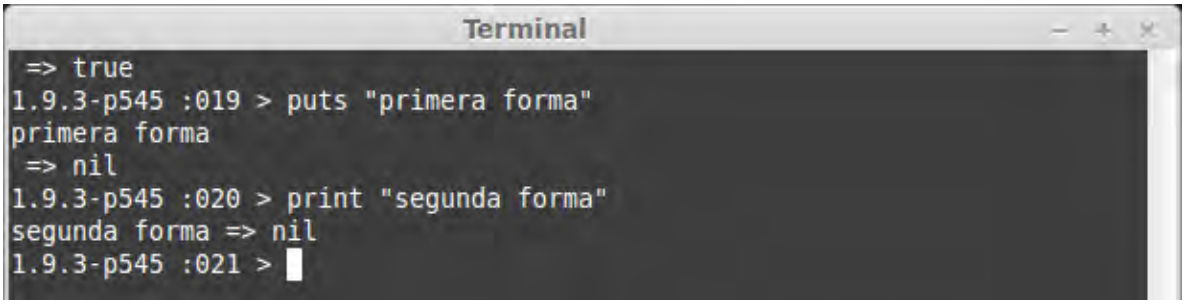
- `:identificador`
- `:nombre`
- `:clave`

Imprimir cadenas

Podemos usar dos métodos para imprimir, el primero lo hemos estado utilizando anteriormente, el método ha sido `puts`, éste va a imprimir el texto que le pasamos pero después regresa *nada* es decir regresa `nil` que vendría a representar un valor `null` en Java. También añade un salto de línea a la salida.

Existe otro método para imprimir cadenas llamado `print`. Éste método no añade un salto de línea a la salida. Estás dos formas de imprimir en pantalla se usan según las necesidades del programador.

[25] Esta *ID* se puede consultar con el método *object_id*.



Comillas simples y comillas dobles

Podemos usar los dos tipos de comillas, la diferencia se encuentra en el tiempo que toma *Ruby* en evaluar el contenido de cada una, siendo las comillas dobles las que más tiempo tardan por las siguientes dos razones:

1. Cuando usamos comillas dobles se buscan *substituciones*: las secuencias de escape dentro del contenido se sustituyen por su valor binario (como en *Java*).

2. Se buscan también *interpolaciones*, esto es, se buscan secuencias de tipo #{ex
3. expresión}, calculando la expresión dentro y sustituyendo el bloque por el resultado esperado.

Se pueden hacer casi las mismas operaciones de las comillas dobles con las simples, una diferencia notable en las últimas, es que no podemos realizar una interpolación, en lugar de imprimir el contenido de una expresión lo que se imprime es toda la expresión tal cual, veamos con un ejemplo:

```
>> ‘Hola’
“Hola”

>> ‘Hola’ + ‘Pavel’
“HolaPavel”

>> ‘#{hola} Pavel’
“#{hola} Pavel’
```

Lo mismo ocurre con las secuencias de escape, al usar comillas simples, imprime la expresión tal cual:

```
>> ‘\n’
“\\n”

>> ‘El veloz murciélago hindú \n comía feliz \t...’
“El veloz murciélago hindú \n comía feliz \t...”
```

3.1.6. Métodos

Para poder manipular objetos, necesitamos métodos, como sabemos cada método contiene una serie de instrucciones en un bloque. Para definir nuevos métodos se deben iniciar con la palabra `def` y terminar con la palabra `end`. Los parámetros van dentro de paréntesis (en *Ruby* los paréntesis son opcionales, en *Rails* se les llama a los métodos sin paréntesis). Por cierto, los métodos se pueden definir a través de la consola o en un archivo con terminación *.rb* (Código 3.2).

Los métodos devuelven el valor de su última línea.

Metodos.rb

Metodos.rb

```ruby
#encoding: utf-8

def holaMundo
    puts 'Hola Mundo'
end

#El método recibe un parámetro con paréntesis
def holaMundo2(nombre)
    puts 'Hola: ' + nombre
end

#El método recibe un parámetro sin paréntesis
def holaMundo3 nombre
    puts 'Hola: ' + nombre
end

#Ejecución del programa
holaMundo

holaMundo2('Pavel')

holaMundo3 'Elizabeth'
```

Código 3.2. **Metodos.rb**.

```
Terminal
pavel@pavel-K61IC ~/lab_rails/primer_aplicacion $ ruby Metodos.rb
Hola
Hola: Pavel
Hola: Elizabeth
pavel@pavel-K61IC ~/lab_rails/primer_aplicacion $
```

Métodos *Bang*

Estos métodos se caracterizan por terminar con un "!" (signo de exclamación), estos métodos lo que hacen es modificar al objeto que los llama. Cuando usamos un método normal, lo que hacemos es crear una copia del objeto que los llama, por el contrario, si usamos un método *bang* con !, el objeto original se modifica. Por eso, estos métodos son considerados peligrosos. Para un mejor entendimiento ejecutemos las líneas en Código 3.3:

MetodosBang.rb

MetodoBang.rb

```ruby
# encoding: utf-8

a = 'Lorem Ipsum'
puts 'Valor inicial de a: ' + a

puts 'Sin uso de método bang'
#Pasamos a mayúsculas el contenido de la variable a sin un método bang
b = a.upcase

puts 'Valor de b: ' + b
puts 'Valor de a: ' + a

puts 'Usando el método bang'

#Ahora hacemos uso de método bang
c = a.upcase!

puts 'Valor de c: ' + c
puts 'Valor de a: ' + a
```

Código 3.3. **MetodoBang.rb**

```
Terminal
pavel@pavel-K61IC ~/lab_rails/primer_aplicacion $ ruby MetodoBang.rb
Valor inicial de a: Lorem Ipsum
Sin uso de método bang
Valor de b: LOREM IPSUM
Valor de a: Lorem Ipsum
usando el método bang
Valor de c: LOREM IPSUM
Valor de a: LOREM IPSUM
```

Métodos: argumentos

Ruby nos permite especificar los valores por defecto de los argumentos que recibe un método por si no se especifica algún valor. Para hacerlo basta con usar el operador de asignación ”=” (igual):

```
def mtd(nombre = “Goyo”, lugar = “UNAM”)
  “#{nombre}, #{lugar}”
end

>> puts mtd
“Goyo, UNAM”

>> puts mtd(“Mixtli”)
“Mixtli, UNAM”
```

Otra ventaja sobre los argumentos en *Ruby* es que podemos especificar que el número de argumentos sea variable, sólo basta con añadir un “*” (asterisco) que preceda al argumento. En el ejemplo guardamos los argumentos en un arreglo llamado “mis_nombres” (más adelante tocaremos el tema de arreglos y otras estructuras de datos). Aunque los argumentos lleven asterisco, el método puede recibir ningún argumento.

```
def argumentoVar(*mis_nombres)
  mis_nombres.each do |palabras|
    puts palabras
  end
end

>> argumentoVar(‘Mixtli’, ‘Goyo’, ‘UNAM’)
“Mixtli”
“Goyo”
“UNAM”
```

Tenemos también argumentos opcionales, estos se especifican después de los los argumentos que sí deben de incluirse. Esto es porque los argumentos se interpretan de izquierda a derecha, así los opcionales deben estar al final por prioridad, más no al revés. Si agregamos un * a lado de nuestro argumento, le indicamos al método que puede recibir más de un argumento (por ejemplo, un método recibe como argumento un nombre, pero si no sabemos cuántos nombres vayan a ser pasados como argumentos, utilizamos un *)

```
>> def opcionales(a=1, b=2, *opcional=a+b)
>>   [a, b, c]
>> end
```

Método *each*

Hemos usado anteriormente el método `each` pero ¿qué es lo que hace? El método `each`, extrae los elementos que existen dentro de un arreglo en una variable que nosotros especifiquemos, está variable estará dentro de dos barras `||`. Todo dentro de un bloque con los respectivos `do` y `end` (Código 3.4):

EjemploEach.rb

EjemploEach.rb

```ruby
# encoding: utf-8

nombres = %w[Pavel Elizabeth Pablo]

nombres.each do |amigos|
    puts 'Mis amigos son: ' + amigos
end

nombres.delete('Pablo')

nombres.each do |amigos|
    puts 'Ahora mis amigos son:' + amigos
end
```

Código. 3.4. **EjemploEach.rb**

```
Terminal
pavel@pavel-K61IC ~/lab_rails/primer_aplicacion $ ruby EjemploEach.rb
Mis amigos son: Pavel
Mis amigos son: Elizabeth
Mis amigos son: Pablo
Ahora, mis amigos son:Pavel
Ahora, mis amigos son:Elizabeth
```

El método `each` es un ciclo que nos permite interactúar con todos lo elementos dentro de un arreglo, se puede usar cualquier nombre de variable pero conviene usar un nombre que tenga cierto significado.

Bloques

Un bloque no es más que una forma de agrupar las instrucciones de una porción de código y sólo puede aparecer después del haber usado un método, el bloque empieza en la misma línea donde se declara el método. Los bloques pueden agrupar las instrucciones en un par de llaves {} o entre do..end[26]. *Ruby* no ejecuta el bloque al instante en que lo lee, "recordará" el bloque y después entrará en el método, ejecutando así el bloque cuando es preciso.

[26] Se recomienda usar las llaves para bloques de una sola línea.

```
>> metodoUno {puts ‘Hola’}

>> metodoDos {“argumento_metodo”} {puts “bloque de metodoDos}

>> (1..10).each do |i|
>>   puts i * 2
>> end
```

3.1.7. Estructuras de Datos

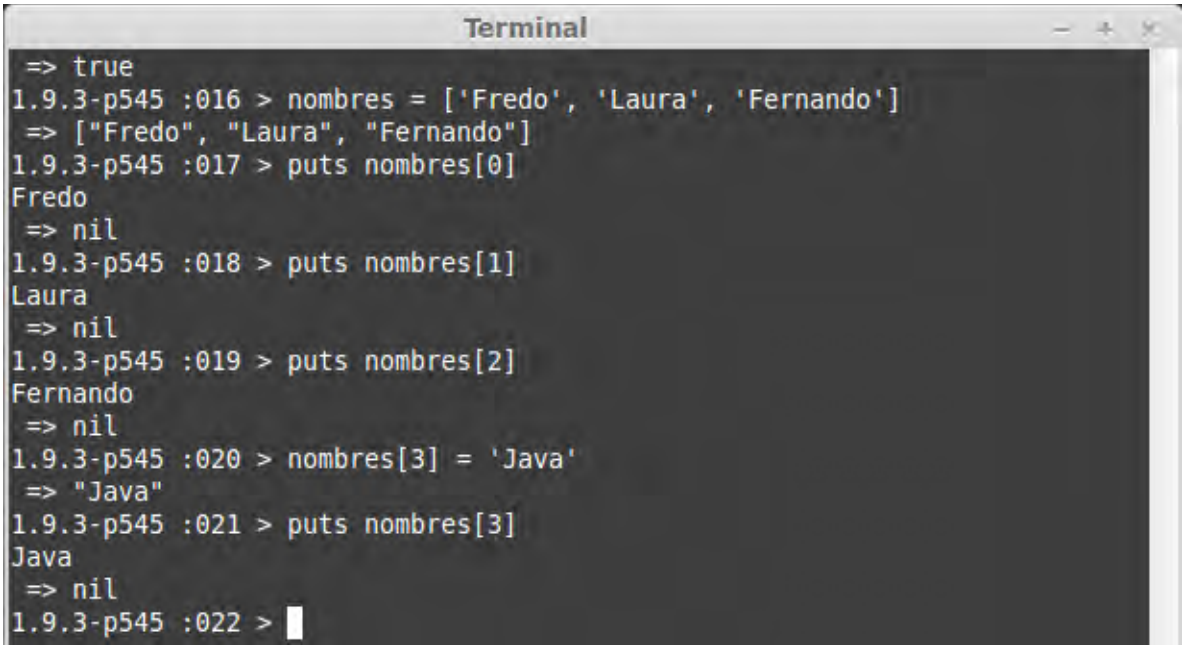
Arreglos

Como debemos saber, un arreglo no es más que una colección de ciertos elementos, que tienen un cierto orden. Cada posición en el arreglo es una variable que se puede tanto leer como escribir.

El arreglo vacío se declara con dos corchetes nada más:

```
>> arreglo_vacío = []
```

Iniciando un arreglo con tres cadenas (nombres):



Podemos añadir elementos de manera fácil a un arreglo previamente definido, como podemos notar en la imagen anterior se añade el nombre *Java* con la instrucción:

```
>> nombres[3] = ‘Java’
```

Un arreglo puede tener un conjunto de distintos elementos, podemos combinar números, cadenas, objetos, etcétera.

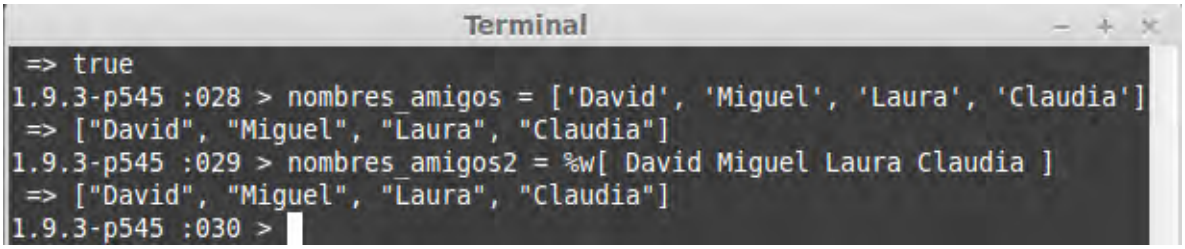
Contamos con una herramienta en *Ruby* que nos facilita la asignación de cadenas a un arreglo. Si tenemos varias cadenas que asignar, la tarea se vuelve tediosa, para evitarlo usamos `%w`.

La forma tediosa:

```
>> nombres_amigos = [‘David’, ‘Miguel’, ‘Laura’, ‘Claudia’]
```

Usando `%w`:

```
nombres_amigos2 = %w[ David Miguel Laura Claudia ]
```



Los arreglos cuentan con distintos métodos, algunos de estos métodos son[27]:

- `sort`
- `length`
- `first`
- `last`
- `split`
- `push`
- `join`

Rangos

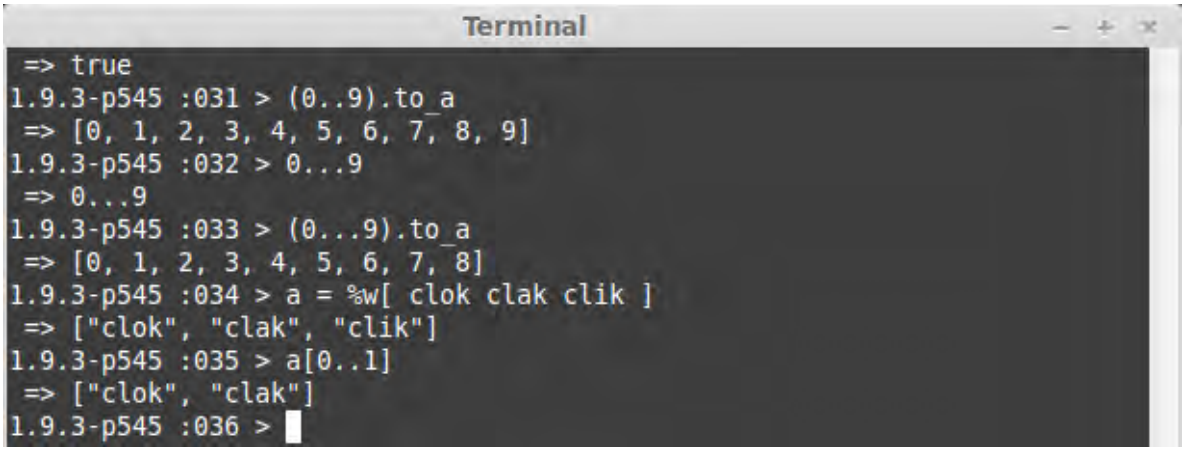
Un rango no es más que una secuencia en *Ruby*. Una secuencia comienza con un punto inicial y punto final, los operandos `..` y `...` producen los valores sucesivos entre ambos. Podemos crear un rango a través de un arreglo:

[27] Para más información sobre arreglos, consultar la documentación: http://www.ruby-doc.org/core-2.1.1/Array.html

```
>> 0..9
“0..9”

>> (0..9).to_a
[0, 1, 2 ,3, 4, 5, 6, 7, 8, 9]

>> 0...9
“0...9”

>> (0...9).to_a
[0, 1, 2 ,3, 4, 5, 6, 7, 8]
```

Los rangos nos ayudan también a restringir arreglos:

```
>> a = %w[clok clak clik]
[“clok”, “clak”, “clik”]

>> a[0..1]
[“clok”, “clak”]
```



Los arreglos también funcionan con caracteres:

```
>> (‘a’..’e’).to_a
[“a”, “b”, “c”, “d”, “e”]
```

Algunos de los métodos con los que cuentan los rangos son[28]:

- *include?*[29]
- *min*
- *max*
- *reject*

Hashes

Este tipo de estructura se maneja de igual manera que en *Java*. Los *hashes* se parecen a los arreglos en que ambos son una colección de referencias a objetos (todo debidamente indexado). La diferencia está en los índices, mientras que en un arreglo el índice es un número, en los *hashes* se puede indexar con objetos (cadenas por ejemplo). Sin embargo, los elementos de un índice en un *hash* no están ordenados (si queremos orden, debemos usar un arreglo).

Los *hashes* cuentan con un valor por defecto si se quieren usar índices que no existen. El valor devuelto es `nil`.

```
>> usuario = {}
“{}”

>> usuario[“nombre”] = 'Tom'
“Tom”

>> usuario[“apellido”] = 'Hanks'
“Hanks”

>> usuario[“apellido”]
“Hanks”

>> usuario[“mascota”]
“nil”

>> usuario
{“nombre” =>”Tom”, “apellido”=>”Hanks”}
```

Un *hash* se indican con una llave que abre y una llave que cierra ({}). Vale la pena mencionar que estás llaves tienen nada que ver con las llaves para bloques. Es conveniente que usemos símbolos como claves o índices. Podemos ahora, definir de la siguiente manera un *hash usuario* como:

[28] Más información sobre rangos: http://www.ruby-doc.org/core-2.1.1/Range.html

[29] Los métodos que tienen un símbolo *?* retornan un valor booleano.

```
>> usuario = { :nombre => 'Pavel', :apellido => 'Cedillo' }[30]

>> usuario[:nombre]
“Pavel”
```

Para denotar el par clave-valor se usa como separador un *hashrocket* representado por `=>`. Existe otra sintaxis equivalente a la anterior para definir *hashes* (soportada en la versión 1.9 o mayores de *Ruby*):

```
>> usuario = { nombre: 'Pavel', apellido: 'Cedillo' }

>> usuarios = {}

>> usuarios[:tom] = { email: 'tom@tvp.com', mascota: 'Fluffy' }
{:email=>"tom@tvp.com", :mascota=>"Fluffy"}”


>> usuarios
{:tom=>{:email=>"tom@tvp.com", :mascota=>"Fluffy"}}

>> usuarios[:tom][:mascota]
“Fluffy”
```

Lo anterior es un *hash* anidado, una característica que se usa bastante con *Rails*, así como con los arreglos, podemos usar el método[31] `each`. Si usamos dicho método debemos de tener en cuenta que no tomará sólo un valor, si no dos: la clave y el valor de la clave. Por ejemplo:

```
>>     mensaje   =   {   :exito   ‘¡Funcionó!’,   fallo:   ‘No
funcionó...’ }

>> mensaje.each do |clave, valor|
>>   puts “Clave #{clave.inspect} tiene asignado el contenido
#{valor.inspect}”
>> end

“Clave :exito tiene asignado el contenido “¡Funcionó!”
“Clave :fallo tiene asignado el contenido “No funcionó...”
```

[30] Se usa por convención un espacio después de un corchete abierto y otro espacio antes de un corchete cerrado. Para más información sobre las convenciones usadas en *Ruby*, consultar: https://github.com/bbatsov/ruby-style-guide

[31] Para una lista de método para la clase *Hash* consultar: http://www.ruby-doc.org/core-2.1.1/Hash.html

3.1.8. Condicionales

Las condicionales son casi iguales a las condicionales usadas en *Java*, la lógica del `if else` se conserva, lo que difiere es que debemos terminar las sentencias con la palabra `end`. Para *Ruby* `nil` y `false` significan falso (0 significa `true`), ver Código 3.5. Vale recordar que las condicionales pueden ir anidadas.

UsoCondicional.rb

UsoCondicional.rb

```
1  # encoding: utf-8
2
3  x = 20;
4
5  if x > 10
6      puts 'x es mayor que 10'
7  else
8      puts 'x es menor a 10'
9  end
```

Código. 3.5. **UsoCondicional.rb**

```
Terminal
pavel@pavel-K61IC ~/lab_rails/primer_aplicacion $ ruby UsoCondicional.rb
x es mayor que 10
pavel@pavel-K61IC ~/lab_rails/primer_aplicacion $
```

Si queremos hacer uso de más decisiones en nuestras condicionales, debemos usar `elsif`[32] (Código 3.6):

UsoCondicional2.rb

UsoCondicional2.rb

```
1  # encoding: utf-8
2
3  nombre = 'Pavel'
4
5  if nombre == 'Lucas'
6      puts 'Extravagante nombre'
7  elsif nombre == 'Pavel'
8      puts '... ¡excelente nombre!'
9  end
```

Código. 3.6. **UsoCondicional2.rb**

[32] `elsif` resulta ser implemente una abreviación a escribir `else if`.

```
Terminal
pavel@pavel-K61IC ~/lab_rails/primer_aplicacion $ ruby UsoCondicional2.rb
... excelente nombre!
pavel@pavel-K61IC ~/lab_rails/primer_aplicacion $
```

Por último, contamos con otra instrucción que es lo contrario a un `if`, la instrucción `unless`. En un `if` si se cumple la condición se ejecuta el bloque con `unless` si no se cumple la condición se ejecuta el bloque (Código 3.7).

UsoCondicional3.rb

UsoCondicional3.rb

```ruby
# encoding: utf-8

nombre = 'Pepe'

unless nombre == 'Pavel'
    puts 'mucho gusto ' + nombre
end
```

Código. 3.7. **UsoCondicional3.rb**

```
Terminal
pavel@pavel-K61IC ~/lab_rails/primer_aplicacion $ ruby UsoCondicional3.rb
mucho gusto Pepe
pavel@pavel-K61IC ~/lab_rails/primer_aplicacion $
```

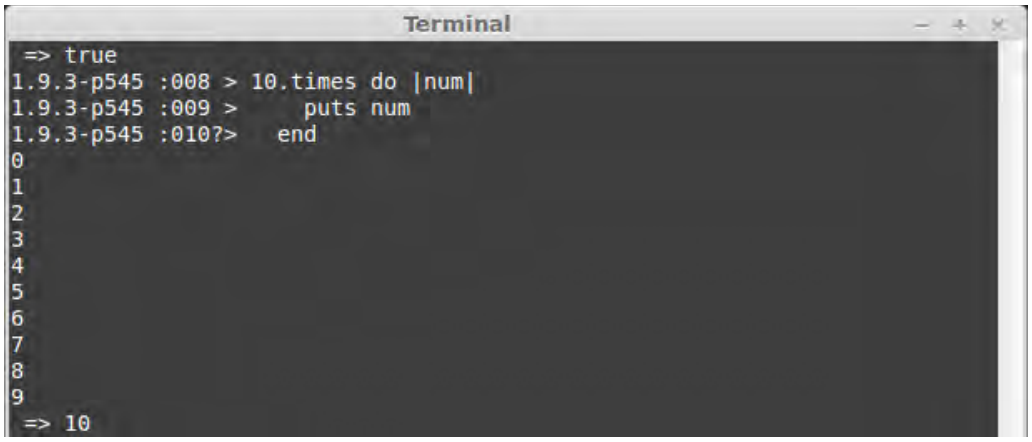
3.1.9. Ciclos

Como vimos anteriormente, el método `each` junto con el bloque que lo acompaña es prácticamente un ciclo. Tenemos otros tres ciclos: `while, times` y `for`.

Haciendo ciclos con `while`:

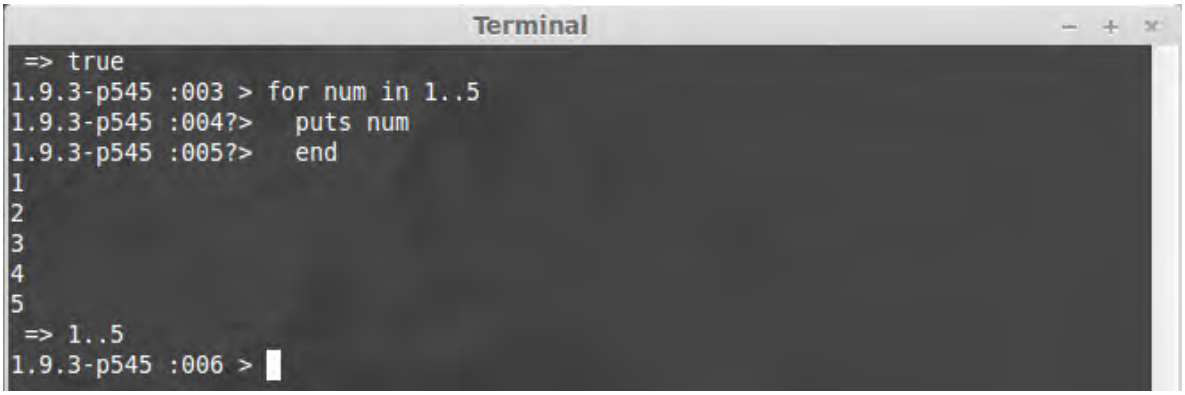
```
Terminal
 => true
1.9.3-p545 :002 > variable = 0
 => 0
1.9.3-p545 :003 > while variable < 5
1.9.3-p545 :004?>   puts variable
1.9.3-p545 :005?>   variable += 1
1.9.3-p545 :006?>   end
0
1
2
3
4
```

Para ciclos usando `times`:



Este ejemplo nos muestra cómo es que todo en *Ruby* es un objeto. El número 10, o mejor dicho, el objeto 10 contiene el método `times` seguido de un bloque que itera la variable `num` 10 veces.

Sabiendo el número de iteraciones que tendrá nuestro ciclo, podemos usar también el ciclo `for`, como sigue:



3.1.10. Clases y objetos en Ruby

Una clase es usada para construir objetos. A su vez, los objetos son una instancia de la clase. Los objetos son contenedores de datos que a su vez controlan el acceso a dichos datos. Los objetos se asocian con una serie variables llamadas atributos y un conjunto de funciones interaccionan con el objeto, los métodos.

Las clases y la creación de objetos siguen casi la misma sintaxis de *Java* en la que se tiene el nombre de nuestra clase, constructores y métodos. Podemos apreciarlo en el Código 3.8:

EjemploPerro.rb

EjemploPerro.rb

```ruby
# encoding: utf-8

#Nombre de la clase
class Perro

#Constructor de la clase
    def initialize(raza, nombre)
#atributos
      @raza = raza
      @nombre = nombre
    end

#Definiendo métodos
    def ladrido
        puts 'iWoof! iWoof!'
    end

    def saludo
        puts "Hola, soy un perro de raza #{@raza}"
    end
end

#Ejecución
perroA = Perro.new('Chow Chow', 'Daria')

perroA.ladrido
perroA.saludo

```

Código. 3.8. **EjemploPerro.rb**

```
Terminal
pavel@pavel-K61IC ~/lab_rails/primer_aplicacion $ ruby EjemploPerro.rb
iwoof! iwoof!
Hola soy un perro de raza Chow Chow
pavel@pavel-K61IC ~/lab_rails/primer_aplicacion $
```

Cada objeto en *Ruby* cuenta con propia *ID* que podemos verificar con el método `object_id`. Para saber si un objeto nos va a responder con algún mensaje o para saber si contiene algún método se usa `respond_to?`

Para saber de qué clase proviene un objeto podemos usar el método `.class`. Por defecto los atributos de un objeto en *Ruby* son privados, Para darles visibilidad desde otras partes de la aplicación se usa `attr_accessor` (lectura y escritura), `attr_reader`(lectura), y `attr_writer` (escritura) como se menciono en el apartado 3.1.4. Los métodos pueden ser públicos, privados, o protegidos, se definen como en el Código 3.9:

EncapsulamientoMetodosRuby.rb

EncapsulamientoMetodosRuby.rb

```
1  class ClaseTest
2    #métodos públicos
3
4    protected
5      #métodos protegidos
6
7    private
8      #metodos privados
9  end
```

Código 3.9. **EncapsulamientoMetodosRuby.rb**

Constructores

Un constructor se inicializa definiendo el método `initialize`, *Ruby* identificará el nombre y da por hecho que este método se restringe a la definición de constructores (ver el programa **EjemploPerro.rb**).

```
>> class Perro
>>   def initialize raza = 'Labrador'
>>     @raza = raza
>>   end
>> end
```

Herencia de clases

La herencia como en *Java*, se maneja de manera similar. Tenemos una clase *padre* o *superclase* que contiene ciertos métodos. Si necesitamos clases que requieran de algunos o todos los métodos de la *superclase*, no valdría hacer una clase repitiendo código contenido en la primera, en cambio, usando la herencia sólo heredamos de la clase padre, una clase *hija o subclase* que hereda los métodos de la *superclase*.

Se usa la sintaxis *nombre_clase* < *superclase.* Un ejemplo de herencia se muestra a continuación (Código 3.10):

Herencia.rb

Herencia.rb

```ruby
# encoding: utf-8

class Perro

    def darPata
        puts 'toma mi pata, humano'
    end
end

#Definiendo la subclase de Perro
class Labrador < Perro

    def ladrar
        puts 'woof woof'
    end
end

#Ejecución del programa
perro = Labrador.new
perro.ladrar
perro.darPata
```

Código 3.10. **Herencia.rb**

Ejecución del programa:

```
Terminal
pavel@pavel-K61IC ~/lab_rails/primer_app $ ruby Herencia.rb
woof woof
toma mi pata, humano
```

Como vemos, la *superclase* `Perro` hereda el método `darPata` a sus *subclases*, aunque no todos los perros puedan dar la pata. Añadimos un método propio de la clase `Labrador` para mostrar que podemos trabajar tranquilamente en otros métodos que no se hayan heredados en las *subclases*.

Si deseamos sobrescribir métodos heredados en cualquier *subclase*. Para hacerlo, basta definir con el mismo nombre el método heredado de la clase padre. Veamos un ejemplo (Código 3.11):

ModificandoMetodos.rb

ModificandoMetodos.rb

```ruby
# encoding: utf-8

class Avion

    def vuela
        puts 'Despegando...'
    end

    def aterrizar
        puts 'Aterrizando...'
    end
end

#Subclase Auto
class Auto < Avion
#Método que sobreescribe el contenido de 'vuela'
    def vuela
        puts 'No puedo volar... por ahora...'
    end
end

#Ejecución del programa
auto = Auto.new
auto.aterrizar
auto.vuela
```

Código 3.11. **ModificandoMetodos.rb**

Ejecución del programa:

```
Terminal
pavel@pavel-K61IC ~/lab_rails/primer_app $ ruby ModificandoMetodos.rb
Aterrizando...
No puedo volar... por ahora...
```

Capítulo 4: Aplicación Ejemplo

4. Desarrollo

Empecemos el desarrollo, generando el esqueleto de nuestra aplicación. Usamos el comando `rails new` para hacerlo.

```
$ rails new primer_app
```

Usaremos exactamente el mismo archivo *Gemfile* del capítulo 2,, Figura 5 e instalaremos las *gemas* de dicho archivo usando *Bundler*:

```
$ bundle install --without production

$ bundle update

$ bundle install
```

Luego, a nuestro proyecto debemos aplicarle el control de versiones de *Git* y lo subimos aun repositorio remoto en *Github*, como ya sabemos esto lo hacemos con:

```
$ git init

$ git add .

$ git commit -m “Primer commit”

$ git push -u origin master[33]
```

4.1. Modelando usuarios y mensajes

El modelo de usuarios será sencillo, tendremos una tabla llamada *usuarios* (Tabla 4.1) que tendrá los siguientes atributos:

- *id*: como su nombre lo indica, será el identificador de los usuarios. El tipo será un *integer*.
- *nombre*: este parámetro es el nombre del usuario y será una cadena (*string*).
- *email*: al igual que el nombre será un *string* y será el correo electrónico del usuario, siendo además el parámetro *usuario* en la autenticación de la aplicación.

[33] Para esta parte ya contamos con un repositorio creado en *Github*, en caso contrario, debemos crear el repositorio remoto y añadirlo a *Git*, luego hacer push.

Tabla 4.1. Modelo de Usuarios.

Modelo Usuario	
id	Integer
nombre	String
email	String

Por otra parte tendremos una tabla llamada *mensajes* (Tabla 4.2) que tendrá los siguientes atributos:

- *id*: sera el identificador de los mensajes que haga el usuario, de igual forma será un i*nteger*.
- *contenido*: como es de suponerse, este atributo contiene el texto de cada mensaje. El tipo es *string*.
- *usuario_id*: para poder asociar un mensaje a un usuario debemos guardar el identificador del usuario en la tabla, por lo tanto tendremos otro *integer*.

Tabla 4.2. Modelo de Mensajes.

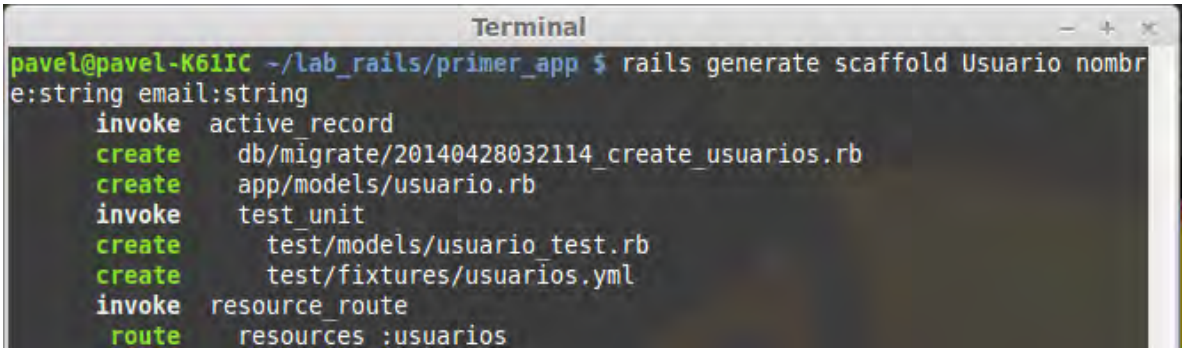
Modelo Mensaje	
id	Integer
contenido	String
usuario_id	String

4.1.1. Modelo de Usuarios

Como se menciona en los objetivos, la interfaz que obtendremos es bastante simple pero funcional, esto es tanto para facilitar la construcción rápida del usuario como para sus mensajes. Para este punto debemos tener un *CRUD* de usuario, es decir, un usuario que pueda ser creado, que se pueda consultar, actualizar y eliminar (*CRUD* viene de *Create*, *Read*, *Updated*, *Deleted*) a través del protocolo *HTTP*.

Haremos uso del comando `scaffold` (que ya viene como estándar en proyecto de *Rails*) y el *script* `rails generate`, sólo debemos pasar el nombre del recurso como argumento, especificando opcionalmente los parámetros para agregar los atributos del modelo:

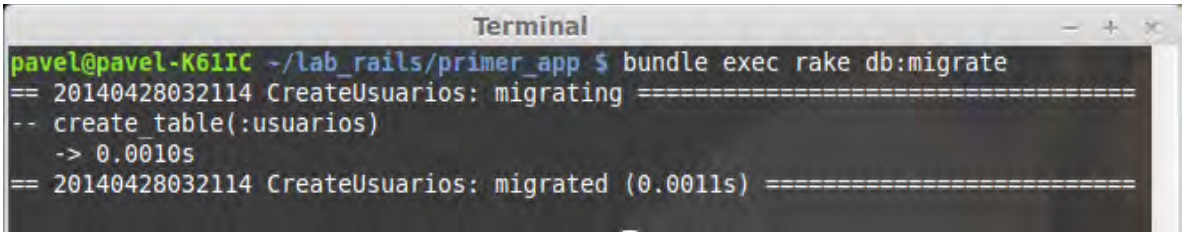
```
$ rails generate scaffold Usuario nombre:string
email:string[34]
```



Resulta claro que tanto `nombre:string` y `email:string` serán incluidos en el modelo como atributos de la base de datos de los usuarios. No se añadió algo como `id:integer` porque *Rails* genera una llave primaria en la base de datos que sirve de identificador de cada usuario.

Debemos actualizar la base de datos con el nuevo modelo *Usuario*, para hacerlo realizamos una *migración* con el siguiente comando:

```
$ bundle exec rake db:migrate[35]
```



Si ejecutamos nuestro programa podemos ver que accedemos a la página principal o *home*, bajo la ruta: http://localhost:3000:

```
$ rails s
```

[34] Le nombramos *Usuario* y no *Usuarios* porque es una convención poner en singular el nombre del modelo, aún haciéndolo en plural, *Rails* lo añade en singular.

[35] Se usa `bundle exec rake` para ejecutar la versión de la gema *rake* que está en el archivo *Gemfile*.

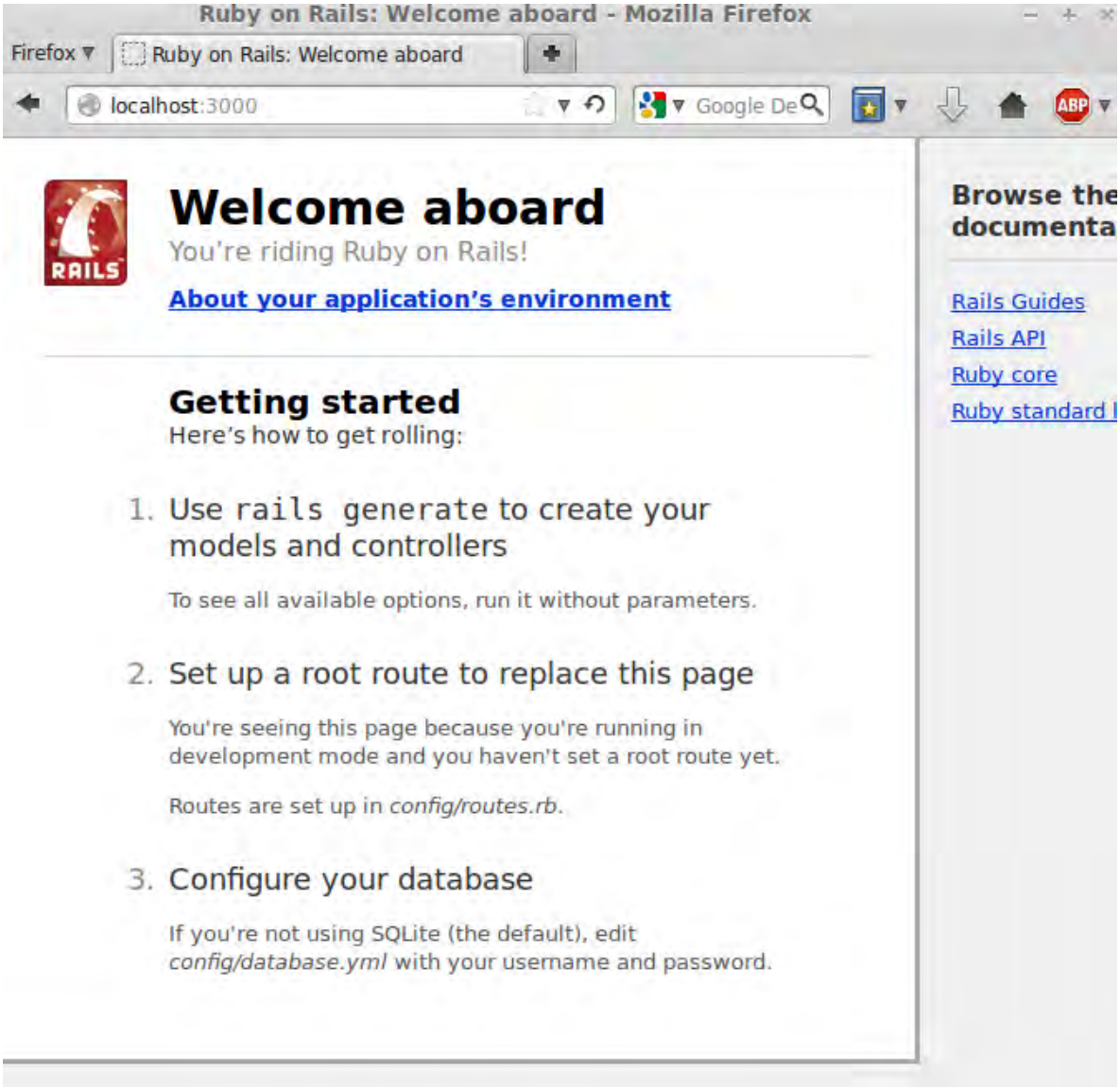


Figura 4.1. Página principal.

Pero al haber generado el *scaffold* de nuestro modelo *Usuario*, también hemos añadido otras rutas (páginas) que pueden modificar a los usuarios. Podemos observar las páginas creadas en la Tabla 4.3.

Tabla 4.3. Rutas para el recurso *usuario*.

Ruta/URL	Acción	Función
/usuarios	`index`	Lista todos los usuarios
/usuarios/x	`show`	Muestra al usuario que tenga el id x
/usuarios/new	`new`	Hace un nuevo usuario
/usuarios/x/edit	`edit`	Edita al usuario con id x

Ejecutamos `rails s` y abrimos nuestro navegador, dirigiéndonos hacia la página del *index* mostrado arriba, que vendría a ser la dirección http://localhost:3000/usuarios. Vemos que la página no tiene usuarios, para agregar algunos nos dirigimos a la ruta http://localhost:3000/new con la que se desplegará un formulario de registro (Figura 4.2).

Es muy fácil crear un usuario, basta con llenar los dos campos que son requeridos y confirmar dando *clic* en el botón *create usuario*.

Figura 4.2. Registro.

Si registramos a un nuevo usuario, éste se mostrará en la página *show* correspondiente a la dirección: http://localhost:3000/usuarios/1. El mensaje de confirmación con letra color verde, utiliza una utilidad llamada *flash* (nada que ver con

flash de Adobe) que se usará más adelante. Destacamos también el id del usuario, que en este caso es 1 y si creamos otro usuario se incrementa en uno (Figura 4.3).

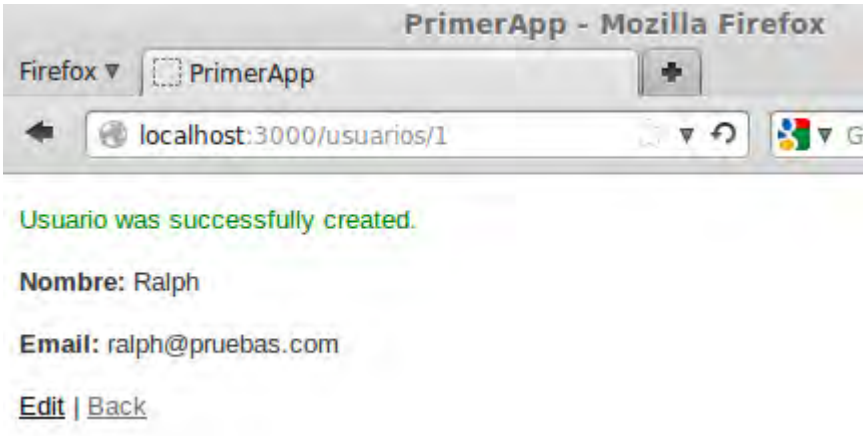


Figura 4.3. Confirmación de registro.

Para modificar la información de un usuario, debemos acceder a la página correspondiente, dadas las rutas de la Tabla 4.3 sería la dirección http://localhost:3000/usuarios/x/edit. Modificamos y damos *clic* al botón *update usuario* (Figura 4.4).

Figura 4.4. Editar información.

Si añadimos más usuarios, la página *show (*http://localhost:3000/usuarios*)* mostrará una lista de usuarios registrados como se ve en la figura 4.5.

Figura 4.5. Lista de Usuarios.

Por último sólo basta probar la eliminación de usuarios de la base de datos. Para esto, damos clic sobre la opción *destroy* y confirmar en la ventana emergente (Figura 4.6).

Figura 4.6. Eliminar usuarios.

.

Cuando nosotros hacemos una petición al navegador a través de una página (hacer un nuevo usuario con /usuario/new por ejemplo), esta petición la toma el *router* de *Rails* que se encargar de enviar las peticiones al controlador de acción (c*ontroller action*) apropiado basado en la *URL* y al tipo de *request* .

El mapeo de las *URLs* del usuario a acciones del controlador para nuestro recurso *usuarios,* lo encontrarnos en el archivo **routes.rb** (Código 4.1) de nuestro proyecto. El código del archivo mencionado establece el par *URL*/acción que vemos en la Tabla 4.4.

config/routes.rb

routes.rb

```
1  PrimerApp::Application.routes.draw do
2    resources :usuarios
3
4    # The priority is based upon order of
5    # See how all your routes lay out with
```

Código 4.1. **routes.rb**

Como se mencionó anteriormente tenemos acciones que pertenecen a un controlador. Este controlador (Código 4.2) es generado al usar el comando `scaffold`, en nuestro caso se generó el controlador `Usuarios`, que es un conjunto de acciones vistas anteriormente.

app/controllers/usuarios_controller.rb

usuarios_controller.rb

```
1  class UsuariosController < ApplicationController
2    before_action :set_usuario, only: [:show, :edit, :update, :destroy]
3
4    # GET /usuarios
5    # GET /usuarios.json
6    def index
7      @usuarios = Usuario.all
8    end
9
10   # GET /usuarios/1
11   # GET /usuarios/1.json
12   def show
13   end
14
15   # GET /usuarios/new
16   def new
17     @usuario = Usuario.new
18   end
19
20   # GET /usuarios/1/edit
21   def edit
22   end
23
24   # POST /usuarios
25   # POST /usuarios.json
26   def create
27     @usuario = Usuario.new(usuario_params)
28
29     respond_to do |format|
30       if @usuario.save
31         format.html { redirect_to @usuario, notice: 'Usuario was successfully created.' }
32         format.json { render action: 'show', status: :created, location: @usuario }
33       else
34         format.html { render action: 'new' }
35         format.json { render json: @usuario.errors, status: :unprocessable_entity }
36       end
37     end
38   end
```

```ruby
39
40    # PATCH/PUT /usuarios/1
41    # PATCH/PUT /usuarios/1.json
42    def update
43      respond_to do |format|
44        if @usuario.update(usuario_params)
45          format.html { redirect_to @usuario, notice: 'Usuario was successfully updated.' }
46          format.json { head :no_content }
47        else
48          format.html { render action: 'edit' }
49          format.json { render json: @usuario.errors, status: :unprocessable_entity }
50        end
51      end
52    end
53
54    # DELETE /usuarios/1
55    # DELETE /usuarios/1.json
56    def destroy
57      @usuario.destroy
58      respond_to do |format|
59        format.html { redirect_to usuarios_url }
60        format.json { head :no_content }
61      end
62    end
63
64    private
65      # Use callbacks to share common setup or constraints between actions.
66      def set_usuario
67        @usuario = Usuario.find(params[:id])
68      end
69
70      # Never trust parameters from the scary internet, only allow the white list through.
71      def usuario_params
72        params.require(:usuario).permit(:nombre, :email)
73      end
74  end
```

Código 4.2. **usuarios_controller.rb**

Existen otras acciones que no mencionamos antes y que corresponden a acciones que se encargan de modificar la información de usuarios dentro de la base de datos. Estás acciones siguen la representación de la arquitectura llamada *REST*[36] (que no abarcaremos en detalle). *REST* bajo el contexto de *Rails*, hace que los componentes de la aplicación sean tratados como recursos que son parte de la aplicación *CRUD* y más aún cuatro métodos del protocolo *HTTP (request): POST*, *GET*, *PATCH* y *DELETE*.

[36] *REST* o *Transferencia de Estado Representacional* es una técnica en arquitecura de software, basado en la tesis doctoral de *Roy Fielding*. *REST* es un estilo de arquitectura en el diseño de las aplicaciones web, tiene como base la idea de sólo utilizar como conexión entre equipos el protocolo *HTTP*.

Tabla 4.4. Rutas *RESTful* para el recurso *Usuario*.

HTTP Request	Ruta/URL	Acción	Función
GET	/usuarios	`index`	Lista todos los usuarios
GET	/usuarios/x	`show`	Muestra al usuario que tenga el id x
GET	/usuarios/new	`new`	Página para hacer un nuevo usuario
POST	/usuarios	`create`	Crea un usuario nuevo
GET	/usuarios/x/edit	`edit`	Edita al usuario con id x
PATCH	/usuarios/x	`update`	Actualiza al usuario con id x
POST	/usuarios/x/edit	`edit`	Edita al usuario con id x

De manera simple, podemos ver la relación entre el modelo *Usuario* y el controlador `Usuarios` viendo la acción `index` de éste último (ver **usuarios_controller.rb**). Con la línea `@usuarios = Usuario.all` el modelo *Usuario* retorna una lista todos los usuarios en la base de datos y la asigna a la variable de instancia `@usuarios`. Podemos observar también como se emplea la herencia de clases como se muestra en el Código 4.2.

app/models/usuario.rb

usuario.rb

```
1  class Usuario < ActiveRecord::Base
2  end
3
```

Código 4.2. **usuario.rb**

Una vez definida la variable `@usuarios`, el controlador se encarga de llamar a la vista **index.html.erb**[37] (Código 4.3), la cual interactúa con la variable de instancia

[37] Los archivos con extensión *.erb*, son archivos *embedded ruby*. Estos archivos combinan código *ruby* dentro de etiquetas *HTML*. Como analogía serían el equivalente a los *JSP* de *Java*.

`@usuarios` y despliega su contenido en pantalla (las variables de instancia están siempre disponibles para las vistas). El contenido que despliega la vista está en *HTML*.

app/views/usuarios/index.html.erb

index.html.erb

```
<h1>Listing usuarios</h1>

<table>
  <thead>
    <tr>
      <th>Nombre</th>
      <th>Email</th>
      <th></th>
      <th></th>
      <th></th>
    </tr>
  </thead>

  <tbody>
    <% @usuarios.each do |usuario| %>
      <tr>
        <td><%= usuario.nombre %></td>
        <td><%= usuario.email %></td>
        <td><%= link_to 'Show', usuario %></td>
        <td><%= link_to 'Edit', edit_usuario_path(usuario) %></td>
        <td><%= link_to 'Destroy', usuario, method: :delete, data: { confirm: 'Are you
          sure?' } %></td>
      </tr>
    <% end %>
  </tbody>
</table>

<br>

<%= link_to 'New Usuario', new_usuario_path %>

```

Código 4.3. **index.html.erb**.

El método `link_to` añade un enlace *HTML* con la *URL* correspondiente a una acción del controlador, también añadiremos más funcionalidades, restricciones y seguridad al recurso *usuarios*, por ahora es sólo un recurso muy básico que tiene como fin ilustrar el funcionamiento de *Rails*.

4.1.2. Modelo Mensaje

Generaremos el esqueleto del recurso *mensajes* de la misma forma que generamos el esqueleto del recurso *usuarios*.

```
$ rails generate scaffold Mensaje contenido:string
usuario_id:integer
```

Actualizamos la base de datos ejecutando una migración:

```
$ bundle exec rake db:migrate
```

El archivo **routes.rb** se actualiza añadiendo el recurso:

routes.rb

```
PrimerApp::Application.routes.draw do
  resources :mensajes

  resources :usuarios
```

Vemos que el archivo **mensajes_controller.rb** (Código 4.4) que se generó con el comando `scaffold` de *Rails* tiene exactamente la misma estructura que el archivo **usuarios_controller.rb**, haciéndose notar la arquitectura *REST* en ambos archivos (Tabla 4.5).

app/controllers/mensajes_controller.rb

mensajes_controller.rb

```
class MensajesController < ApplicationController
  before_action :set_mensaje, only: [:show, :edit, :update, :destroy]

  # GET /mensajes
  # GET /mensajes.json
  def index
    @mensajes = Mensaje.all
  end

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

  # GET /mensajes/new
  def new
    @mensaje = Mensaje.new
  end

  # GET /mensajes/1/edit
  def edit
  end

  # POST /mensajes
  # POST /mensajes.json
  def create
    @mensaje = Mensaje.new(mensaje_params)

    respond_to do |format|
      if @mensaje.save
        format.html { redirect_to @mensaje, notice: 'Mensaje was successfully created.' }
        format.json { render action: 'show', status: :created, location: @mensaje }
      else
        format.html { render action: 'new' }
        format.json { render json: @mensaje.errors, status: :unprocessable_entity }
      end
    end
  end
```

Código 4.4. **index.html.erb**.

```ruby
  # PATCH/PUT /mensajes/1
  # PATCH/PUT /mensajes/1.json
  def update
    respond_to do |format|
      if @mensaje.update(mensaje_params)
        format.html { redirect_to @mensaje, notice: 'Mensaje was successfully updated.' }
        format.json { head :no_content }
      else
        format.html { render action: 'edit' }
        format.json { render json: @mensaje.errors, status: :unprocessable_entity }
      end
    end
  end

  # DELETE /mensajes/1
  # DELETE /mensajes/1.json
  def destroy
    @mensaje.destroy
    respond_to do |format|
      format.html { redirect_to mensajes_url }
      format.json { head :no_content }
    end
  end

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

    # Never trust parameters from the scary internet, only allow the white list through.
    def mensaje_params
      params.require(:mensaje).permit(:contenido, :usuario_id)
    end
end

```

Tabla 4.5. Rutas *RESTful* para el recurso *Mensaje*.

HTTP Request	Ruta/URL	Acción	Función
GET	/mensajes	`index`	Lista todos los usuarios
GET	/mensajes/x	`show`	Muestra el mensaje que tenga el id x
GET	/mensajes/new	`new`	Página para hacer un nuevo mensaje
POST	/mensajes	`create`	Crea un nuevo mensaje
GET	/mensajes/x/edit	`edit`	Edita el mensaje con id x
PATCH	/mensajes/x	`update`	Actualiza el mensaje con id x
DELETE	/mensajes/x	`delete`	Edita al usuario con id x

Podemos realizar los mismos pasos que con el recurso u*suarios*, crear nuevos mensajes a partir de la *URL* asociada a dicha acción. La diferencia es que podemos asociar un mensaje a un usuario en particular a través de su id, por lo demás, es exactamente igual (Figura 4.7 y 4.8).

Figura 4.7. Registro de Mensajes.

Figura 4.8. Mostrando mensajes.

4.1.3. Restringir la longitud de los mensajes

Para restringir la longitud de caracteres en un mensaje, debemos de usar el método `length` (como en *Java*). Abrimos el archivo modelo **mensaje.rb** (Código 4.5) y hacemos la validación de la longitud a 140 caracteres, como se muestra a continuación:

app/models/mensaje.rb

```
mensaje.rb
1  class Mensaje < ActiveRecord::Base
2      validates :contenido, length: { maximum:140 }
3  end
```

Código 4.5. **mensaje.rb**.

Si queremos verificar los cambios, hacemos un mensaje con más de 140 caracteres y verificamos que no nos deje crear el mensaje como en la Figura 4.9:

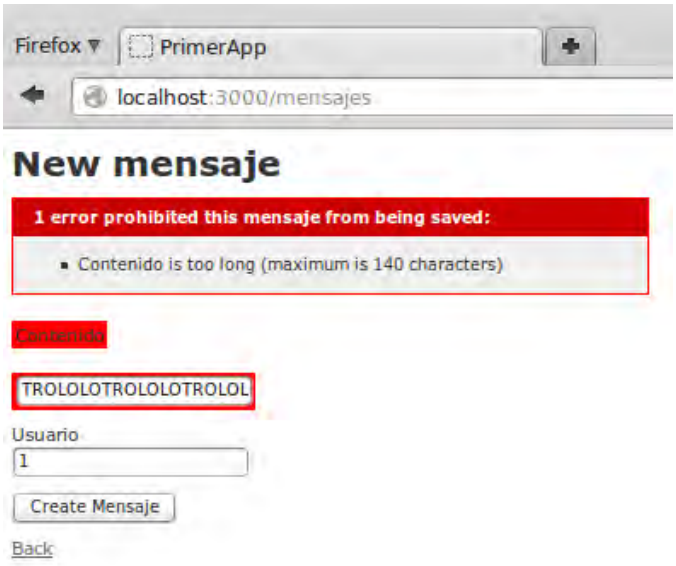


Figura 4.9. Error detectado durante la validación.

4.1.4. Asociando usuarios y mensajes

Rails cuenta con una forma de asociar distintos modelos, en nuestro caso, asociamos usuarios con sus respectivos mensajes. Podemos expresar que un usuario cuenta con tantos mensajes añadiendo a los respectivos modelos el método `has_many` seguido del recurso que queremos asociar. *Rails* infiere la relación entre el *id* del modelo *Usuario* y el *id_usuarios* del modelo *Mensaje* (Figura 4.10).

```
usuario.rb
1  class Usuario < ActiveRecord::Base
2      has_many :mensajes
3  end
```

```ruby
# mensaje.rb
class Mensaje < ActiveRecord::Base
    belongs_to :usuario
    validates :contenido, length: { maximum:140 }
end
```

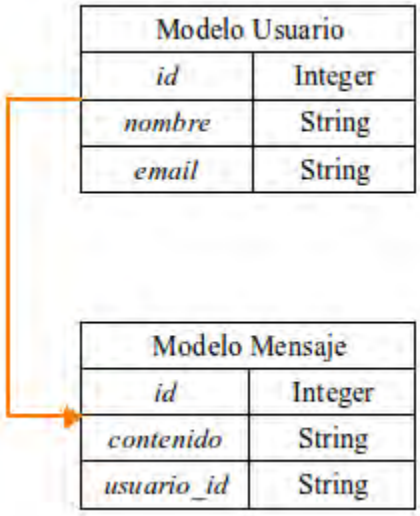


Figura 4.10. Asociación entre modelos.

Hagamos uso de la consola de *Rails* para interactuar con nuestra aplicación, abrimos una terminal y tecleando `rails c`, hacemos lo siguiente:

```
>> rails c
```

```
>> primer_usuario = Usuario.first
```

```
pavel@pavel-K61IC ~/lab_rails/primer_app $ rails c
Loading development environment (Rails 4.0.4)
1.9.3-p545 :001 > primer_usuario = Usuario.first
  Usuario Load (0.4ms)  SELECT "usuarios".* FROM "usuarios" ORDER BY "usuarios".
"id" ASC LIMIT 1
 => #<Usuario id: 2, nombre: "Lilith", email: "lilith@ideas.edu", created_at: "2
014-04-28 03:42:26", updated_at: "2014-04-28 03:43:38">
```

```
>> primer_usuario.mensajes
```

```
  Mensaje Load (4.1ms)  SELECT "mensajes".* FROM "mensajes" WHERE "mensajes"."us
uario_id" = ?  [["usuario_id", 1]]
 => #<ActiveRecord::Associations::CollectionProxy [#<Mensaje id: 1, contenido: "
Este es un buen día para salir a caminar", usuario_id: 1, created_at: "2014-05-1
2 03:46:22", updated_at: "2014-05-12 03:46:22">, #<Mensaje id: 2, contenido: "Es
te es mi segundo mensaje", usuario_id: 1, created_at: "2014-05-12 03:46:39", upd
ated_at: "2014-05-12 03:46:39">]>
```

Como podemos ver, es muy sencillo interactuar con el modelo y obtener la información que necesitamos.

4.1.5. Herencia

Tenemos dos modelos: *Usuario* y *Mensaje*, estos dos modelos heredan de la clase `ActiveRecord::Base`[38], que es la base proporcionada por `ActiveRecord` para los modelos, otorgándoles la posibilidad de comunicarse con la base de datos, tratar las columnas de una base como atributos de *Ruby*, etcétera.

Así como con los modelos, los controladores *Usuarios* y *Mensajes* heredan de `ApplicationController` y éste a su vez hereda de `ActionController::Base`[39], que los dota de funcionalidades como accede y manipular objetos de los modelos, administrar peticiones *HTTP*, mostrar vistas como *HTML*, etcétera. Al heredar de `ApplicationController`, todas las reglas que definamos en el controlador `Application`, automáticamente se aplican para todas las acciones de nuestra aplicación.

4.1.6. Guardando cambios

Por último no olvidemos guardar los cambios en git y hacer un *commit* para confirmar, a su vez, subir a nuestro repositorio el proyecto.

```
>> git commit -am ‘Finalizando mi primer aplicación’


>> git push origin master
```

[38] Para conocer más sobre ActiveRecord::Base, se puede consultar el siguiente enlace: http://api.rubyonrails.org/classes/ActiveRecord/Base.html

[39] Ídem http://api.rubyonrails.org/classes/ActionController/Base.html

Capítulo 5: Páginas Estáticas y Dinámicas

5. Desarrollo

Creamos un nuevo proyecto como ya es costumbre y modificamos nuestro archivo *Gemfile* mostrado en el capítulo 2, Figura 2.5 y hacemos una instalación de las gemas usando *Bundler*:

```
$ rails new base_app

$ bundle install --without production

$ bundle update

$ bundle install
```

No olvidemos añadir nuestro proyecto a *Git* y subirlo al repositorio online del que hagamos uso, en nuestro caso a *Github*:

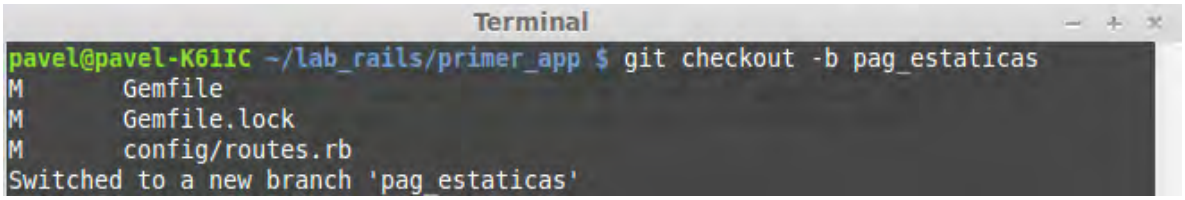
```
$ git init

$ git add .

$ git commit -m 'Primer commit'

$ git remote add origin https://github.com/laboratorio-
rubyrails/def_app.git

$ git push origin master
```

5.1. Páginas estáticas

En est sección crearemos páginas estáticas usando acciones y vistas con contenido en *HTML* estático, que en un futuro podremos añadir contenido dinámico. Por lo anterior estaremos trabajando bajo los directorios correspondientes a los controladores y a las vistas, respectivamente.

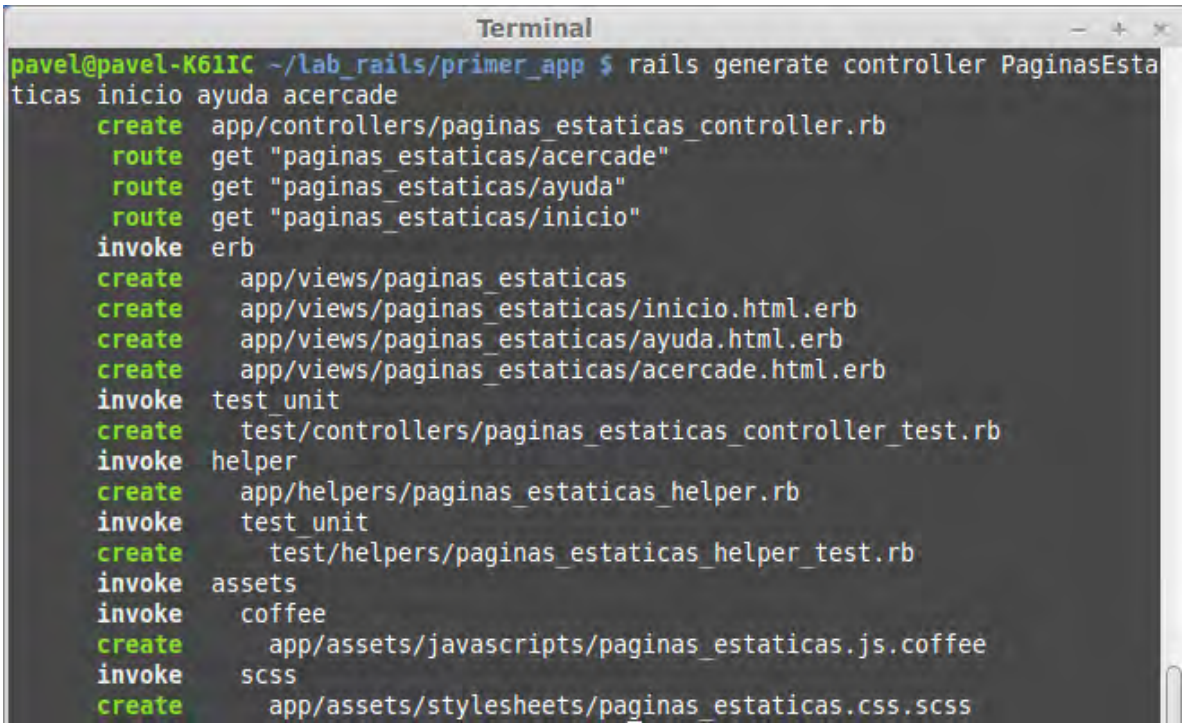
Está vez *Git* nos va a ayudar a hacer las modificaciones al proyecto usando una rama dedicada. Entonces creamos nuestra rama llamada *estaticas*:

```
$ git checkout -b pag_estaticas
```



Hemos estado usando el *script* `generate` de *Rails* para crear los controladores que necesitamos, no será la excepción está vez, así que vamos a generar el controlador llamado `PaginasEstaticas`, creamos también en el mismo comando tres acciones correspondientes a la página *inicio*, *ayuda* y la página *acercade*. Notemos que aunque usamos el estilo *CamelCase*[40] para nombrar al controlador, *Rails* genera un archivo llamado **paginas_estaticas_controller.rb**, esto lo hace por convención, así que no hay de que preocuparnos.

```
$ rails generate controller PaginasEstaticas inicio ayuda
acercade
```



Al generar el controlador, se actualiza el archivo **routes.rb**, que ya sabemos, se usa para encontrar la correspondencia entre páginas web y *URLs*. Vemos los tres controladores ya dentro de las rutas del mencionado archivo.

[40] Para más información sobre este estilo de escritura: http://es.wikipedia.org/wiki/CamelCase

```
routes.rb
1  BaseApp::Application.routes.draw do
2    get "paginas_estaticas/inicio"
3    get "paginas_estaticas/ayuda"
4    get "paginas_estaticas/acercade"
```

La regla: `get "paginas_estaticas/inicio"` mapea todas las peticiones que se hagan desde la *URL /paginas_estaticas/inicio* a su acción correspondiente, siendo en este caso la acción `inicio` perteneciente al controlador `PaginasEstaticas` dentro del archivo **paginas_estaticas_controller.rb** (Código 5.1).

app/controllers/paginas_estaticas_controller.rb

```
paginas_estaticas_controller.rb
1  class PaginasEstaticasController < ApplicationController
2    def inicio
3    end
4
5    def ayuda
6    end
7
8    def acercade
9    end
10 end
```

Código 5.1. **paginas_estaticas_controller.rb**

El `get` que vemos en el archivo de rutas, indica que la ruta va a responder a una petición *GET*, es decir, que cada vez que creamos una acción dentro del controlador, automáticamente obtenemos una página con ruta /nombre_controlador/nombreAccion (Figura 5.1).

Figura 5.1. Página de *inicio*.

Si observamos el archivo **paginas_estaticas_controller.rb** y lo comparamos con el archivo **users_controller.rb**, podemos notar que el primero no hace uso de las acciones de *REST*, esto es normal si se trata de un conjunto de páginas estáticas. Vemos que en el

controlador de páginas estáticas las acciones `inicio`, `ayuda` y `acercade` se encuentran vacías, esto no quiere decir que no hagan nada, notamos que `PaginasEstaticasController` hereda de `ApplicationController` lo cual brinda a estos métodos funciones especificas para *Rails* (si fuera sólo *Ruby*, en efecto, no harían nada).

Cuando hacemos una visita a una página, *Rails* busca en el controlador de páginas estáticas y ejecuta el código contenido de la acción correspondiente y entonces genera la vista de dicha acción. En nuestro caso, si visitamos la página http://localhost:3000/paginas_estaticas/inicio y verificamos que la acción `inicio` en el controlador está vacía, podemos preguntarnos cómo es que *Rails* genera dicha página sin que la acción tenga código dentro. *Rails* genera la vista porque cuando generamos nuestro controlador anteriormente, se generó también la correspondiente vista que en el ejemplo vendría a ser **inicio.html.erb** (Código 5.2).

app/views/paginas_estaticas/inicio.html.erb

inicio.html.erb

```
1  <h1>PaginasEstaticas#inicio</h1>
2  <p>Find me in app/views/paginas_estaticas/inicio.html.erb</p>
```

Código 5.2. **inicio.html.erb**

Vemos que el archivo contiene código en *HTML* con un encabezado y un párrafo solamente, lo cual, hace ver que las vistas de *Rails* pueden contener sólo HTML estático. Análogamente **ayuda.html.erb** (Código 5.3), y **acercade.html.erb** (Código 5.4), así que podemos cambiar su contenido por algo como:

inicio.html.erb

```
1  <h1>Mi aplicación</h1>
2  <p>Esta es la página de prueba para
3     las vistas de mi aplicación
4  </p>
```

app/views/paginas_estaticas/ayuda.html.erb

ayuda.html.erb

```
1  <h1>Ayuda</h1>
2  <p>
3     Esta es la página de prueba para las vistas de mi aplicación.
4     Para obtener ayuda sobre Ruby on Rails, puedes visitar:
5     <a href="http://rubyonrails.org"></a>
6  </p>
7
```

Código 5.3. **ayuda.html.erb**

app/views/paginas_estaticas/acercade.html.erb

```
acercade.html.erb
1  <h1>Acerca de</h1>
2  <p>
3      Esta página tendrá en el futuro, información sobre la aplicación
4  </p>
5
```

Código 5.4. **acercade.html.erb**

5.1.1. Añadiendo una página manualmente

Para añadir una página a nuestra aplicación ya sea porque se nos pasó a la hora de generar los controladores o simplemente porque surgió la necesidad, podemos añadir la página manualmente a nuestra aplicación haciendo lo siguiente:

1. Debemos añadir al archivo **routes.rb** la ruta correspondiente y el tipo de petición. Para el ejemplo usaremos una petición de tipo *GET* y una página llamada *prueba*:

```
routes.rb
1  BaseApp::Application.routes.draw do
2    get "paginas_estaticas/inicio"
3    get "paginas_estaticas/ayuda"
4    get "paginas_estaticas/acercade"
```

2. Añadimos la acción correspondiente al controlador (en nuestro caso añadimos la acción `prueba` al controlador `PaginasEstaticas`).

```
paginas_estaticas_controller.rb
1  class PaginasEstaticasController < ApplicationController
2    def inicio
3    end
4
5    def ayuda
6    end
7
8    def acercade
9    end
10
11   def prueba
12   end
13 end
```

3. Luego, añadimos la vista (página) de la acción que añadimos al controlador:

prueba.html.erb

```
<h1>Prueba</h1>
<p>
  Esta página sirve de ejemplo para
  verificar que se añadió una
  página manualmente a nuestra
  aplicación. :)
</p>
```

4. Por último, basta verificar a través del navegador la *URL* de la página que creamos (previamente haber arrancado el servidor de *Rails*) y que especificamos en el archivo **routes.rb** (http://localhost:3000/paginas_estaticas/prueba, Figura 5.2):

Figura 5.2. Página de *prueba*.

5.2. Páginas dinámicas ligeras

Vamos añadir un poco de dinamismo en las páginas *inicio*, *ayuda* y *acercade*. Añadiremos un título que cambia según el contenido de la página, para esto debemos de editar las páginas antes mencionadas para que cambien los títulos de cada una, usaremos la etiqueta `<title>`.

Añadimos la siguiente estructura *HTML* a nuestras páginas:

inicio.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title>Mi aplicación | Inicio</title>
  </head>
  <body>
    <h1>Mi Aplicación</h1>
    <p>
      Esta es la página de prueba para
      las vistas de mi aplicación
    </p>
  </body>
</html>
```

ayuda.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title>Mi aplicación | Ayuda</title>
  </head>
  <body>
    <h1>Ayuda</h1>
    <p>
      Esta es la página de prueba para las vistas de mi aplicación.
      Para obtener ayuda sobre Ruby on Rails, puedes visitar:
      <a href="http://rubyonrails.org"></a>
    </p>
  </body>
</html>
```

acercade.html.erb

```

<!DOCTYPE html>
<html>
  <head>
    <title>Mi aplicación | Acerca De</title>
  </head>
  <body>
    <h1>Acerca De</h1>
    <p>
      Esta página tendrá en el futuro, información sobre la aplicación
    </p>
  </body>
</html>
```

Con los cambios que hemos efectuado, estamos duplicando mucha información y eso es una violación grave al principio de *No te repitas* (*Don't repeat yourself*, *DRY* por sus siglas en inglés), Los títulos de las páginas son casi idénticos a excepción de una palabra que especifica en qué página estamos, por lo demás, cuenta con "*Mi aplicación"* como título. También la estructura *HTML* se repite en cada página. Debemos de eliminar esa redundancia de información usando código de *Ruby* en las vistas (*Embedded Ruby*).

Vamos a usar una función especial de *Rails* llamada `provide` para establecer los títulos de cada página. Veamos cómo funciona cambiando la palabra *Inicio* en el archivo **inicio.html.erb**:

inicio.html.erb

```
<% provide(:titulo, 'Inicio')%>
<!DOCTYPE html>
<html>
  <head>
    <title>Mi aplicación | <%= yield(:titulo) %></title>
  </head>
  <body>
    <h1>Mi Aplicación</h1>
    <p>
      Esta es la página de prueba para
      las vistas de mi aplicación
    </p>
  </body>
</html>
```

El código `<% provide(:titulo, 'Inicio') %>` indica que *Rails* debe asociar la cadena `'Inicio'` con el símbolo `:titulo`, luego para insertar el título añadimos la función `yield` al cuerpo de la etiqueta `<title>` con `<title>Mi aplicación | <%= yield(:titulo) %></title>`

Antes de verificar los cambios realizados, cambiemos el nombre de un *layout* creado por *Rails* al inicio, el archivo es **application.html.erb**:

```
$ mv app/views/layout/application.html.erb pendiente
```

Lo anterior es porque ese archivo es la base para generar cualquier vista, se detalla más adelante. Por el momento ejecutemos el servidor de *rails* y nos dirigimos a la página de inicio, la página resultante es igual que la anterior con la excepción de que la parte variable del título se genera de manera dinámica.

Hacemos lo correspondiente a los archivo **ayuda.html.erb** y **acercade.html.erb**:

ayuda.html.erb

```
<% provide(:titulo, 'Ayuda')%>
<!DOCTYPE html>
<html>
  <head>
    <title>Mi aplicación | <%= yield(:titulo) %></title>
  </head>
  <body>
    <h1>Ayuda.</h1>
    <p>
      Esta es la página de prueba para las vistas de mi aplicación.
      Para obtener ayuda sobre Ruby on Rails, puedes visitar:
      <a href="http://rubyonrails.org"></a>
    </p>
  </body>
</html>
```

acercade.html.erb

```
1  <% provide(:titulo, 'Acerca De')%>
2
3  <!DOCTYPE html>
4  <html>
5    <head>
6      <title>Mi aplicación | <%= yield(:tituld) %></title>
7    </head>
8    <body>
9      <h1>Acerca De</h1>
10     <p>
11       Esta página tendrá en el futuro, información sobre la aplicación
12     </p>
13   </body>
14 </html>
```

5.2.1. Evitando duplicar información: uso de layouts

Hasta ahora hemos modificado las vistas de nuestra aplicación, los títulos de cada página se generan ligeramente de manera dinámica y además cada página cuenta con la misma estructura *HTML* siendo el contenido dentro de la etiqueta `<body>` lo que cambia. Podemos ahora, simplificar las vistas (dado que son idénticas en estructura) haciendo uso de un *layout*[41] especial llamado **application.html.erb** que previamente habíamos cambiado su nombre, ahora restauremos el archivo:

```
$ mv pendiente app/views/layouts/application.html.erb
```

Ahora, para que este *layout* funcione en nuestra aplicación, reemplazaremos al igual que las vistas anteriores el contenido de la etiqueta `<title>`:

application.html.erb

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title>Mi aplicación | <%= yield(:titulo) %></title>
5    <%= stylesheet_link_tag    "application", media: "all", "data-turbolinks-track" => true
       %>
6    <%= javascript_include_tag "application", "data-turbolinks-track" => true %>
7    <%= csrf_meta_tags %>
8  </head>
9  <body>
10
11 <%= yield %>
12
13 </body>
14 </html>
```

Observamos que el contenido de la etiqueta `<body>` contiene la línea `<%= yield %>`, lo que hace esta línea es insertar el contenido de cada página en el layout, cuando

[41] Podemos tomarlo como un arreglo o diseño base para una página web en la que se especifica la distribución de los elementos de un diseño.

visitamos una página convierte el contenido a *HTML* y lo inserta en lugar de `<%= yield %>`.

De lo anterior podemos intuir que en el caso del título, el *layout* busca el título del método `provide` y lo inserta a través de `yield` y el argumento `:titulo`.

Sobre el código restante que contiene el *layout*, corresponde a la inclusión de hoja de estilo de la aplicación y de *JavaScript*, las cuales son parte del llamado *asset pipeline* (se desarrolla más adelante). Todo esto junto con el método csrf_meta_tags, previenen un ataque *cross-site request forgery*[42] (*CSRF*).

Hablaremos más sobre los estilos en el capítulo 6, por ahora sólo requerimos saber saber que la función `stylesheet_link_tag`, recibe una cadena y un *hash* con dos elementos, indicando el tipo de medios (*media types*[43] en inglés) a los que se aplicará el estilo y el uso de *Turbolinks*[44].

Una vez realizados los cambios en el *layout*, debemos de eliminar el código *HTML* sobrante de las páginas *inicio*, *ayuda* y *acercade*; dejando sólo el contenido de la etiqueta `<body>`.

inicio.html.erb

```
<% provide(:titulo, 'Inicio')%>
<h1>Mi Aplicación</h1>
<p>
  Esta es la página de prueba para
  las vistas de mi aplicación
</p>
```

ayuda.html.erb

```
<% provide(:titulo, 'Ayuda')%>
<h1>Ayuda</h1>
<p>
  Esta es la página de prueba para las vistas de mi aplicación.
  Para obtener ayuda sobre Ruby on Rails, puedes visitar:
  <a href="http://rubyonrails.org"></a>
</p>
```

acercade.html.erb

```
<% provide(:titulo, 'Acerca De')%>
<h1>Acerca De</h1>
<p>
  Esta página tendrá en el futuro, información sobre la aplicación
</p>
```

[42] Para más información sobre este tipo de ataque, consultar: http://*en.wikipedia.org*/wiki/Cross-site_request_forgery

[43] *Media types:* http://www.w3.org/TR/CSS2/media.html

[44] Prácticamente lo que hace es mejorar el rendimiento y velocidad de carga de páginas, sin dejar que el navegador recompile *JavaScript* y *CSS* entre páginas. Más información: *https://github.com/rails/turbolinks*

En este punto hemos eliminado la redundancia de código en las vistas *inicio*, *ayuda* y *acercade*, las páginas muestran el mismo contenido en el navegador pero ahora con menos duplicidad.

5.3. Guardando cambios

No olvidemos añadir los archivos a *Git* y confirmarlos.

```
$ git commit -am 'Finalizando Capítulo 5 – Páginas Estáticas'
```

Luego, añadir los cambios de la rama *pag_estaticas* a la rama *master* y por último subir a nuestro repositorio remoto el proyecto:

```
$ git checkout master

$ git merge pag_estaticas

$git push origin master
```

Capítulo 6: Añadiendo Estilo

6. Desarrollo

En el capítulo anterior, generamos el controlador `PaginasEstaticas` junto con tres vistas nuevas para nuestra aplicación: *inicio*, *ayuda* y a*cercade*. Una vez generadas, añadimos un poco de dinamismo al hacer que los títulos de cada página, cambien de acuerdo a su nombre con ayuda del *layout* de *Rails* **application.html.erb**.

Para simplificar la estructura de las vistas y dejarles sólo el contenido, modificamos el *layout* para que, como se dijo en el párrafo anterior, modificase el título en base al nombre de la página y además que insertará el contenido de dichas páginas a través de la función `yield`.

Si recordamos, existía una línea de código (entre otras) que hacía uso de una función propia de *Rails* llamada `stylesheet_link_tag`[45] que añade el estilo **application.css** a todos los tipos de multimedia.

Ahora toca implementar las nuevas funciones y empezamos creando una rama para trabajar sobre el estilo:

```
$ git checkout -b usando_estilos
```

6.1. Helpers

Además de las funciones ya predefinidas en *Rails* para su uso en las vistas, podemos crear nuestras propias funciones, llamadas *helpers*, el objetivo de estas funciones es mantener el código lo más simple posible y evitando el *DRY*. Un ejemplo del uso de *helpers* es si nos planteamos la siguiente pregunta: ¿qué pasa si nuestra página no tiene un título? Sería bueno contar un título por default que se use cuando no exista un título definido. Si quitamos la llamada a la función `provide` obtenemos lo que muestra la Figura 6.1:

Figura 6.1. Página sin título especifico.

[45] Para una descripción detallada de este método, consultar el *API*de *Rails*: http://api.rubyonrails.org/classes/ActionView/Helpers/AssetTagHelper.html#method-i-stylesheet_link_tag

El título "*Mi aplicación* |" parece bueno pero la barra vertical no ayuda mucho. Para solucionar este problema haremos uso del *helper* llamado `titulo_base`. Como el nombre indica, `titulo_base` retorna el título "*Mi aplicación*" si no se definió uno y añade la barra vertical si uno es definido. Modificamos el archivo **application_helper.rb**[46]:

app/helpers/application_helper.rb

application_helper.rb

```ruby
# encoding: utf-8
module ApplicationHelper

    def titulo_base(titulo_pagina)
        titulo = "Mi aplicación"
        if titulo_pagina.empty?
            titulo
        else
            "#{titulo} | #{titulo_pagina}"
        end
    end
end

```

Código 6.1. **application_helper.rb**.

Sustituimos el contenido de la etiqueta `<title>` de **application.html.erb** por nuestro *helper* como se muestra:

application.html.erb

```erb
<!DOCTYPE html>
<html>
<head>
  <title><%= titulo_base(yield(:titulo)) %></title>
  <%= stylesheet_link_tag    "application", media: "all", "data-turbolinks-track" => true
      %>
  <%= javascript_include_tag "application", "data-turbolinks-track" => true %>
  <%= csrf_meta_tags %>
</head>
<body>

<%= yield %>

</body>
</html>

```

Para verificar que todo lo anterior funcione, debemos quitar la línea donde está la función `provide` de la vista *Inicio* (o de alguna otra) y abrir la página (como vemos en la Figura 6.2):

[46] Si usamos acentos no olvidemos agregar # encoding: utf-8

Figura 6.2. Título generado por un *helper*.

6.2. Usando estilos CSS

Ahora vamos a actualizar nuestro *layout* **application.html.erb** con elementos *HTML* adicionales. Incluiremos algunas divisiones de la página, clases *CSS* y como elemento principal (por el momento) la barra de navegación. Primero explicaremos las partes que vamos a añadir y al final mostramos todo el código del nuevo *layout*.

Incluimos un encabezado a través de la etiqueta `<header>`, haremos divisiones a nuestra página con ayuda de la etiqueta `<div>` y también añadimos una lista de elementos con sus respectivos enlaces de navegación.

application.html.erb

```
<!DOCTYPE html>
<html>
<head>
  <title><%= titulo_base(yield(:titulo)) %></title>
  <%= stylesheet_link_tag    "application", media: "all", "data-turbolinks-track" => true %>
  <%= javascript_include_tag "application", "data-turbolinks-track" => true %>
  <%= csrf_meta_tags %>
</head>
<body>
    <header class="navbar navbar-fixed-top navbar-inverse">
      <div class="navbar-inner">
        <div class="container">
          <%= link_to "Mi Aplicación", '#', id: "encabezado" %>
          <nav>
            <ul class="nav pull-right">
              <li><%= link_to "Inicio",    '#' %></li>
              <li><%= link_to "Ayuda",     '#' %></li>
              <li><%= link_to  "Entrar", '#' %></li>
            </ul>
          </nav>
        </div>
      </div>
    </header>
    <div class="container">
      <%= yield %>
    </div>
  </body>
```

La etiqueta `<header>` indica los elementos en la parte superior de la página. En el encabezado agregamos tres clases *CSS* (que no tienen nada que ver con las clases de *Ruby*). Las clases que agregamos son `navbar`, `navbar-fixed-top` y `navbar-inverse`.

Podemos usar tanto clases como *ids* para asignar todos los elementos *HTML*; la diferencia está en que las clases pueden ser utilizadas varias veces sobre una página y los *ids* sólo una. Para nuestro caso, todas las clases *navbar* tienen un significado para el marco *Bootstrap*, más adelante veremos cómo instalarlo y cómo es que interacciona con dichas clases.

Como habíamos dicho antes, usaremos divisiones para las páginas web, en este caso están en el encabezado y en el cuerpo de éste. La etiqueta `<div>` sólo hace una división genérica en la página. Con la llegada de *HTML5*, éste se encarga de añadir `header`, `nav` y `section` a las divisiones más comunes entre aplicaciones. Las divisiones de nuestra página contienen una clase CSS, que como el encabezado, tienen un significado para el marco *Bootstrap*.

```
<header class="navbar navbar-fixed-top navbar-inverse">
  <div class="navbar-inner">
    <div class="container">
      <%= link_to "Mi Aplicación", '#', id: "encabezado" %>
      <nav>
        <ul class="nav pull-right">
          <li><%= link_to "Inicio",    '#' %></li>
          <li><%= link_to "Ayuda",     '#' %></li>
          <li><%= link_to  "Entrar", '#' %></li>
        </ul>
      </nav>
    </div>
  </div>
</header>
```

Pasamos a la parte que contiene código de *Ruby:*

```
<ul class="nav pull-right">
  <li><%= link_to "Inicio",    '#' %></li>
  <li><%= link_to "Ayuda",     '#' %></li>
  <li><%= link_to  "Entrar", '#' %></li>
</ul>
```

En el código anterior se hace uso del *helper* `link_to` que nos crea los enlaces de manera automática (si usamos sólo *HTML*, usaríamos la etiqueta `<a>`); el primer argumento es el texto que tendrá el enlace dentro de la página, el segundo elemento es la ruta del enlace,que en este caso al no tener uno, asignamos el símbolo "#"[47], como tercer argumento tenemos un *hash* que añade el *id encabezado* (*CSS*) al enlace.

Luego, tenemos una lista desordenada de enlaces (etiqueta `<ul>`) junto con las clases `nav` y `pull-right` que todo junto, tienen un significado especial para *Bootstrap*. Cuando *Rails* procese la página con el código de *Ruby*, nos va a generar un código como sigue:

```
<ul class="nav pull-right">
  <li><a href="#">Inicio</a></li>
  <li><a href="#">Ayuda</a></li>
  <li><a href="#">Entrar</a></li>
</ul>
```

Por último tenemos la última división con una clase `container` creada para *Bootstrap* y la función `yield` para insertar el contenido de cada página en el *layout*. Modifiquemos un poco la página *Inicio* y verifiquemos que dichos cambios se apliquen en la página (Figura 6.3).

[47] Lo que hace este símbolo en un enlace, es que al darle clic, simplemente se recarga la página sin cambio alguno.

inicio.html.erb

```
1  <div class="center hero-unit">
2
3  <h1>Mi Aplicación</h1>
4  <h2>Esta página es el comienzo de una aplicación sencilla pero que en un futuro crecerá</h2>
5  <p>
6    Esta es la página de prueba para
7    las vistas de mi aplicación
8  </p>
9  <%= link_to "¡Registrate, es rápido y fácil!", '#', class: "btn btn-large btn-primary" %>
10 </div>
11
12 <%= link_to image_tag("rails_icono.png", alt: "Rails Icono"), 'http://rubyonrails.org/' %>
```

Figura 6.3. Prueba de *Inicio*.

Las clases *CSS* como `hero-unit`, `btn`, `btn-large`, `btm-primary` serán usadas por *Bootstrap* y el enlace que se genera después de hacer uso del *helper* `link_to` es:

```
<a class="btn btn-large btn-primary" href="#">¡Registrate, es rápido y fácil!</a>
</div>

<a href="http://rubyonrails.org/"><img alt="Rails Icono" src="/assets/rails_icono.png" /></a>
    </div>
  </body>
```

Similar a `link_to` está `image_tag` que recibe como argumentos la ruta a una imagen y un *hash* opcional, para este último se hace uso del atributo `alt`[48] que se asigna a un símbolo. *Rails* automáticamente asocia el archivo de imagen con los archivos en el directorio `images` del servidor. Colocando los activos o *assets* dentro del directorio `assets` permite ser procesados más rápido.

El icono de *Rails* se encuentra en http://rubyonrails.org/images/rails.png, debemos colocarlo dentro del directorio **app/assets/images/**.

6.3. Bootstrap

En las secciones pasadas hicimos uso de varias clases *CSS* para dar flexibilidad a nuestro *layout*, muchas de las clases que usamos son especificas de *Bootstrap*[49], un marco de trabajo de *Twitter* que se encarga de añadir un diseño web vistoso y también de añadir elementos de la interfaz de usuario a una aplicación *HTML5*. Es ahora cuando toca saber cómo se complementan las clases *CSS* usadas con el marco *Bootstrap*.

Bootstrap usa el lenguaje *LESS CSS* para crear hojas de estilo dinámicas pero el *asset* pipeline de *Rails* soporta el lenguaje *SASS* por defecto, entonces para convertir de *LESS* a *SASS* debemos instalar la *gema bootstrap-sass* que se encarga de esa tarea y que hace disponible todos los archivos de *Bootstrap* a nuestra aplicación.

Gemfile

```
1  source 'https://rubygems.org'
2
3  # Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
4  gem 'rails', '4.0.4'
5
6  # Instalando Bootstrap
7  gem 'bootstrap-sass', '2.3.2.0'
8
9  .
10 .
11 .
```

Como sabemos, instalamos la gema a través del comando `bundle install`. Reiniciamos el servidor web presionando `ctrl-c` en la terminal con el servidor corriendo y luego `rails server` para iniciarlo de nuevo. Como paso final debemos añadir una línea al archivo **application.rb** para que el *asset pipeline* de *Rails* sea compatible con *bootstrap-sass*:

```
22      .
23      .
24      .
25      config.assets.precompile += %w(*.png *.jpg *.jpeg *.gif)
26    end
27  end
```

[48] El atributo `alt` se mostrará si no existe una imagen o ésta se encuentra dañada. Si no se especifica alguna, *Rails* añade un atributo `alt` tomando el nombre del archivo sin incluir la extensión.

[49] Página principal de *Bootstrap:* http://getbootstrap.com/

Para añadir un archivo CSS debemos crearlo primero:

```
$ subl app/assets/stylesheets/estilo.css.scss
```

Es importante destacar que el nombre del directorio y del archivo son muy importantes el directorio app/assets/stylesheets es parte del *asset pipeline* y cualquier hoja de estilo se añade automáticamente como parte del archivo **application.css** incluido en el *layout* del sitio. El archivo que creamos cuenta con la extensión *.css* que indica el uso de un archivo *CSS* y la extensión *.scss* que prepara el archivo para el *asset pipeline* usando *Sass*.

Para incluir *Bootstrap* en el archivo **estilo.css.scss** debemos de importarlo añadiendo la línea `@import "bootstrap"` (ver Código 6.2).

app/assets/stylesheets/estilo.css.scss

```
estilo.css.scss  x
1  @import "bootstrap";
```

Código 6.2. **estilo.css.scss**.

Con la línea anterior añadimos todo el marco de trabajo *CSS* de *Bootstrap*, los resultados podemos notarlos al abrir la página Inicio que se muestra más atractiva visualmente aunque a un nivel muy sencillo (Figura 6.4).

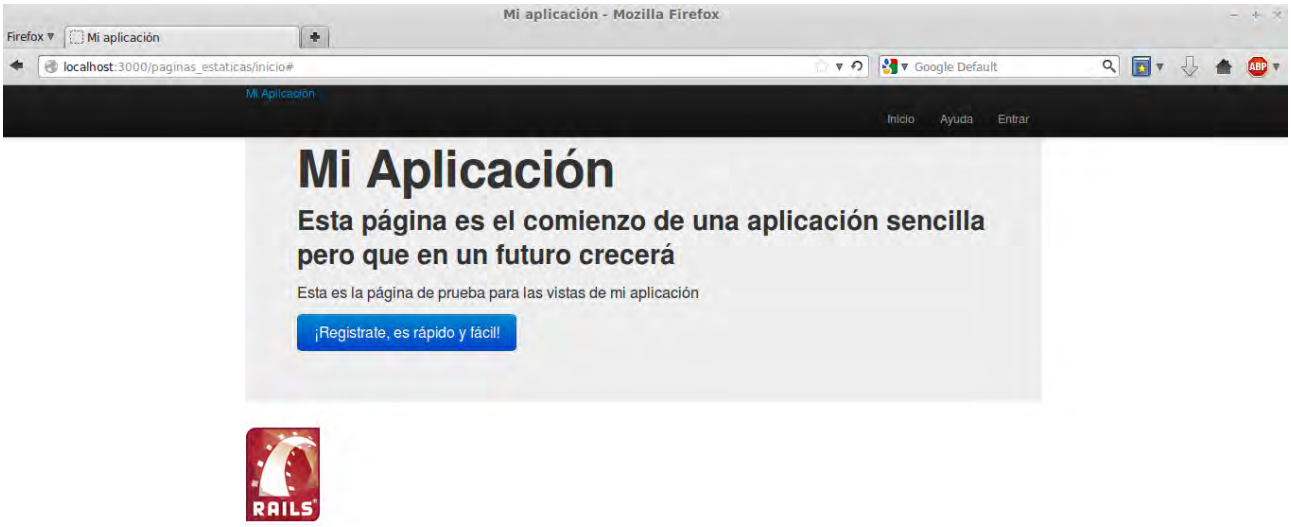


Figura 6.4. Página de inicio.

Ahora vamos a añadir algunas reglas a nuestro archivo **estilo.css.scss** las cuales darán estilo a cada página de nuestro sitio. Definimos las reglas que refieren a clases, *ids*, etiquetas *HTML* entre otras, seguidos de una lista de comando de estilo:

estilo.css.scss

```
1  @import "bootstrap";
2
3  /* Estilo general */
4
5  html {
6      overflow-y: scroll;
7  }
8
9  body {
10   padding-top: 60px;
11 }
12
13 section {
14   overflow: auto;
15 }
16
17 textarea {
18   resize: vertical;
19 }
20
21 .center {
22   text-align: center;
23 }
24
25 .center h1 {
26   margin-bottom: 10px;
27 }
```

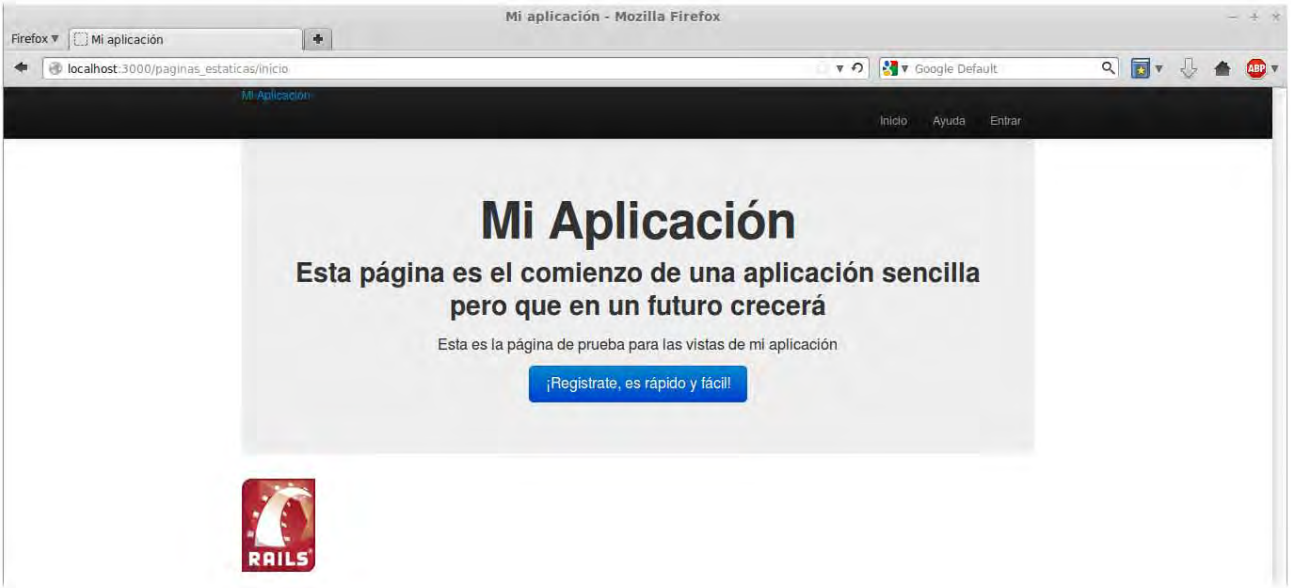


Figura 6.5. Página de inicio con estilo.

En el código anterior colocamos una `padding`[50] de 70 píxeles al inicio de la página. Cuando añadimos la clase `navbar-fixed-top` en la etiqueta `<header>`, *Bootstrap* se encarga de colocar barra de navegación al inicio de la página, es cuando el `padding` nos sirve para separar el texto principal de la navegación (el uso de `navbar-inverse` hace que el color de navbar sea más oscuro).

La regla:

```
21  .center {
22    text-align: center;
23  }
```

Se encarga de asociar la clase `center` con la propiedad `text-align: center`. El punto que se antepone en `.center`, nos indica que se va a dar estilo a una clase (el símbolo `#` indica que se dará estilo a un *id CSS*, como veremos más adelante). Esto significa que los elementos que se encuentren dentro de una etiqueta con la clase `center` van a ser centrados en la página.

Bootstrap nos permite añadir reglas *CSS* para la tipografía de nuestras páginas, vamos a añadir las siguientes líneas a nuestra hoja de estilo:

```
29  /* Estilo para texto */
30
31  h1, h2, h3, h4, h5, h6 {
32    line-height: 1;
33  }
34
35  h1 {
36    font-size: 5em;
37    letter-spacing: -2px;
38    margin-bottom: 30px;
39    text-align: center;
40  }
41
42  h2 {
43    font-size: 1.3em;
44    letter-spacing: -1px;
45    margin-bottom: 30px;
46    text-align: center;
47    font-weight: normal;
48    color: #959;
49  }
50
51  p {
52    font-size: 1.1em;
53    line-height: 1.7em;
54  }
```

[50] Para evitar confusiones al alumno, omitimos la traducción al español de los comando de estilo *CSS*.

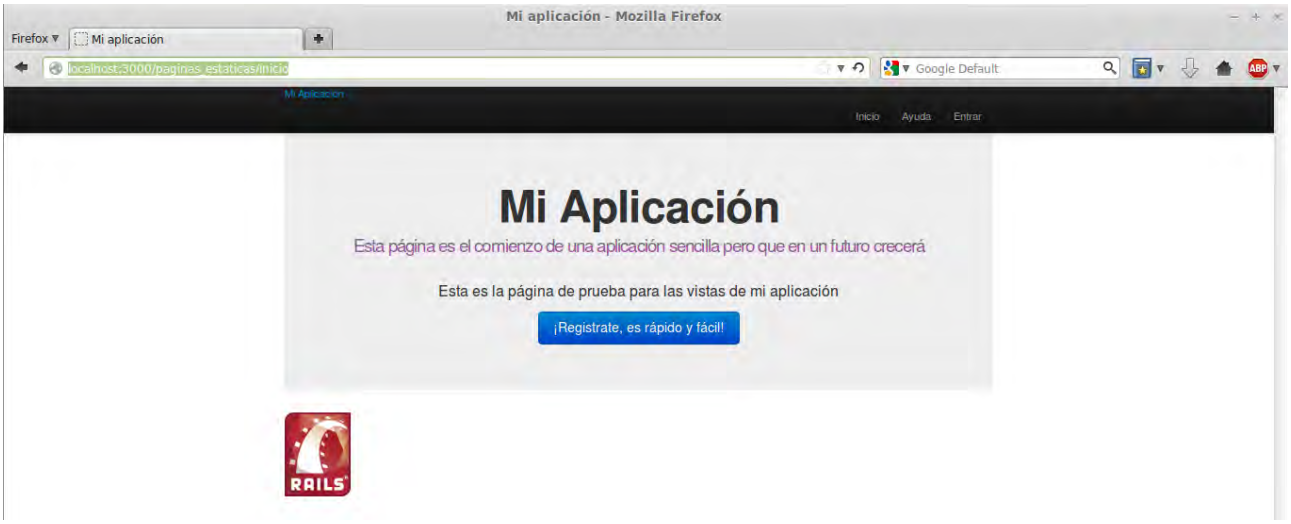


Figura 6.6. Inicio con estilo en tipografía.

Ahora vamos a darle estilo a nuestro encabezado, primeramente pasamos el texto a mayúsculas (`text-transform: uppercase`), posteriormente modificamos el tamaño (`font-size: 1,7em`), el lugar que ocupa (que vendrían a ser las demás propiedades en `#logo`) y por último el color a través de `#logo:hover` y la propiedad `color: #F2F2F2`.

```
55
56  /* Estilo para encabezado */
57
58  #encabezado {
59    float: left;
60    margin-right: 10px;
61    font-size: 1.8em;
62    color: #F2F2F2;
63    text-transform: uppercase;
64    letter-spacing: -1px;
65    padding-top: 9px;
66    font-weight: bold;
67    line-height: 1;
68  }
69
70  #encabezado:hover {
71    color: #FFF;
72    text-decoration: none;
73  }
```

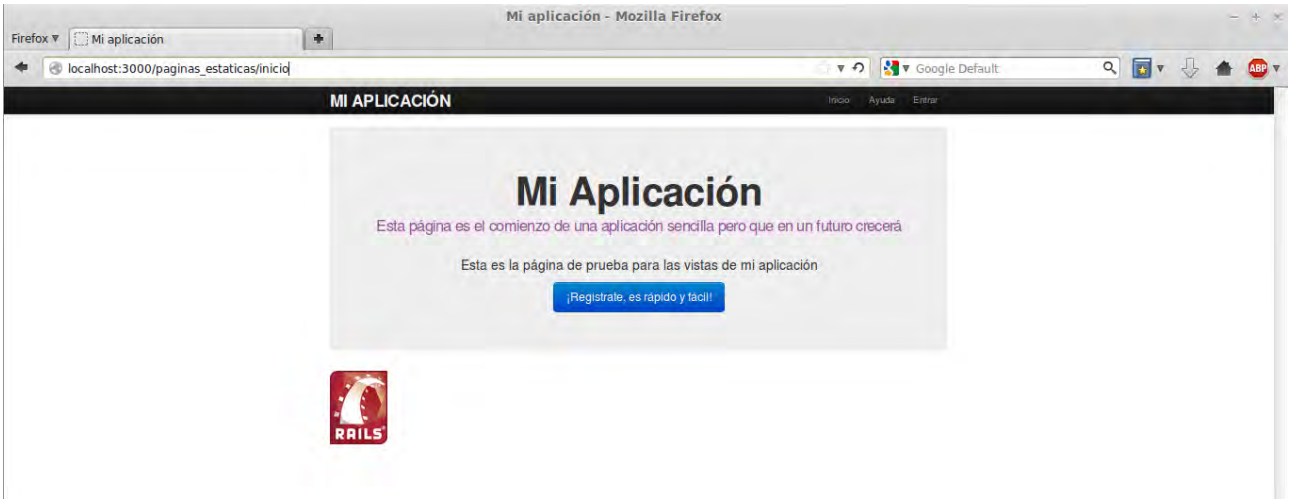


Figura 6.7. Estilo en encabezado.

6.4. Parciales (Partials)

Para entender el funcionamiento de un parcial (*partial*), veamos que nuestro *layout* **application.html.erb** se está empezando a hacer muy desordenado. Si queremos hacer que el *layout* sea más organizado teniendo menos líneas, usamos parciales. Veamos el siguiente código:

application.html.erb

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title><%= titulo_base(yield(:titulo)) %></title>
5    <%= stylesheet_link_tag    "application", media: "all", "data-turbolinks-track" => true %>
6    <%= javascript_include_tag "application", "data-turbolinks-track" => true %>
7    <%= csrf_meta_tags %>
8  </head>
9  <body>
10     <%= render 'layouts/etiquetaHeader'%>
11     <div class="container">
12       <%= yield %>
13     </div>
14   </body>
15 </html>
```

```
<%= render 'layouts/etiquetaHeader'%>
```

La línea anterior se encarga de buscar un archivo llamado **eiquetaHeader.html.erb**, evalúa el contenido e inserta el resultado en la vista. El guión bajo es la convención para nombrar parciales, además de poder identificarlos más rápido. Para que funcione el *helper* `render`, creamos el parcial como en Código 6.3:

app/views/layouts/_etiquetaHeader.html.erb

_etiquetaHeader.html.erb

```
<header class="navbar navbar-fixed-top navbar-inverse">
  <div class="navbar-inner">
    <div class="container">
      <%= link_to "Mi Aplicación", '#', id: "encabezado" %>
      <nav>
        <ul class="nav pull-right">
          <li><%= link_to "Inicio",    '#' %></li>
          <li><%= link_to "Ayuda",     '#' %></li>
          <li><%= link_to "Entrar", '#' %></li>
        </ul>
      </nav>
    </div>
  </div>
</header>
```

Código 6.3. **_etiquetaHeader.html.erb**

Ya que agregamos el contenido de la etiqueta `<header>`, hagamos un pie de página con la etiqueta `<footer>` y realizamos el correspondiente parcial. El pie de página contará con los enlaces para las páginas de c*ontacto* y a*cercade* y que al igual que los enlaces de la etiqueta `<header>` llevan el símbolo "`#`" como *URL* (Código 6.4).

app/views/layouts/_etiquetaFooter.html.erb

_etiquetaFooter.html.erb

```
<footer class="footer">
  Aplicación creada en <a href="http://rubyonrails.org/">Ruby on Rails.</a>
  <nav>
    <ul>
      <li><%= link_to "Acerca De",   '#' %></li>
      <li><%= link_to "Contacto", '#' %></li>
      <li><a href="http://rubyonrails.org/">¿Qué hay de nuevo en Rails?</a></li>
    </ul>
  </nav>
</footer>
```

Código 6.4. **_etiquetaFooter.html.erb**

Al final nuestro archivo **application.html.erb** queda de la forma:

application.html.erb

```
<!DOCTYPE html>
<html>
<head>
  <title><%= titulo_base(yield(:titulo)) %></title>
  <%= stylesheet_link_tag    "application", media: "all", "data-turbolinks-track" => true %>
  <%= javascript_include_tag "application", "data-turbolinks-track" => true %>
  <%= csrf_meta_tags %>
</head>
<body>
    <header class="navbar navbar-fixed-top navbar-inverse">
      <div class="navbar-inner">
        <div class="container">
          <%= link_to "Mi Aplicación", '#', id: "encabezado" %>
          <nav>
            <ul class="nav pull-right">
              <li><%= link_to "Inicio",    '#' %></li>
              <li><%= link_to "Ayuda",     '#' %></li>
              <li><%= link_to  "Entrar", '#' %></li>
            </ul>
          </nav>
        </div>
      </div>
    </header>
    <div class="container">
      <%= yield %>
    </div>
  </body>
```

Verificamos que todo haya salido bien visitando la página de inicio y verificando que sí aparezca nuestro pie de página con los enlaces hacia c*ontacto* y a*cercade*:

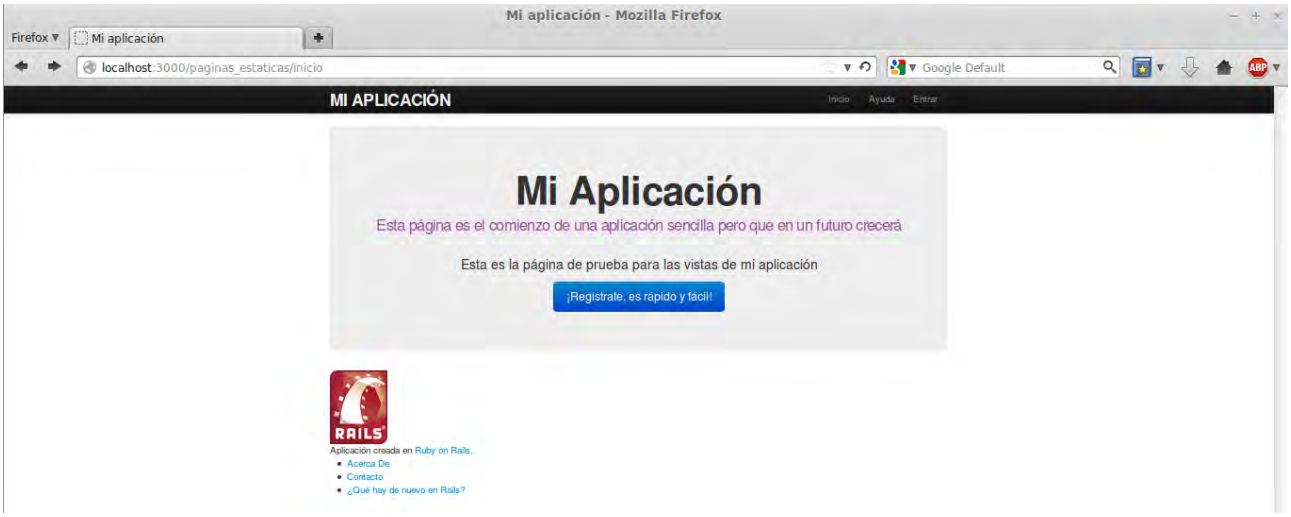


Figura 6.8. Pie de página en *Inicio*.

El pie de página como vemos, no tiene estilo, toca turno a modificar nuestro archivo **estilo.css.scss**:

```
74
75  /* Estilo para pie de página */
76
77  footer {
78    margin-top: 50px;
79    padding-top: 5px;
80    border-top: 1px solid #000001;
81    color: #999;
82  }
83
84  footer a {
85    color: #999;
86  }
87
88  footer a:hover {
89    color: #000001;
90  }
91
92  footer small {
93    float: left;
94  }
95
96  footer ul {
97    float: right;
98    list-style: none;
99  }
100
101 footer ul li {
102   float: left;
103   margin-left: 15px;
104 }
105
```

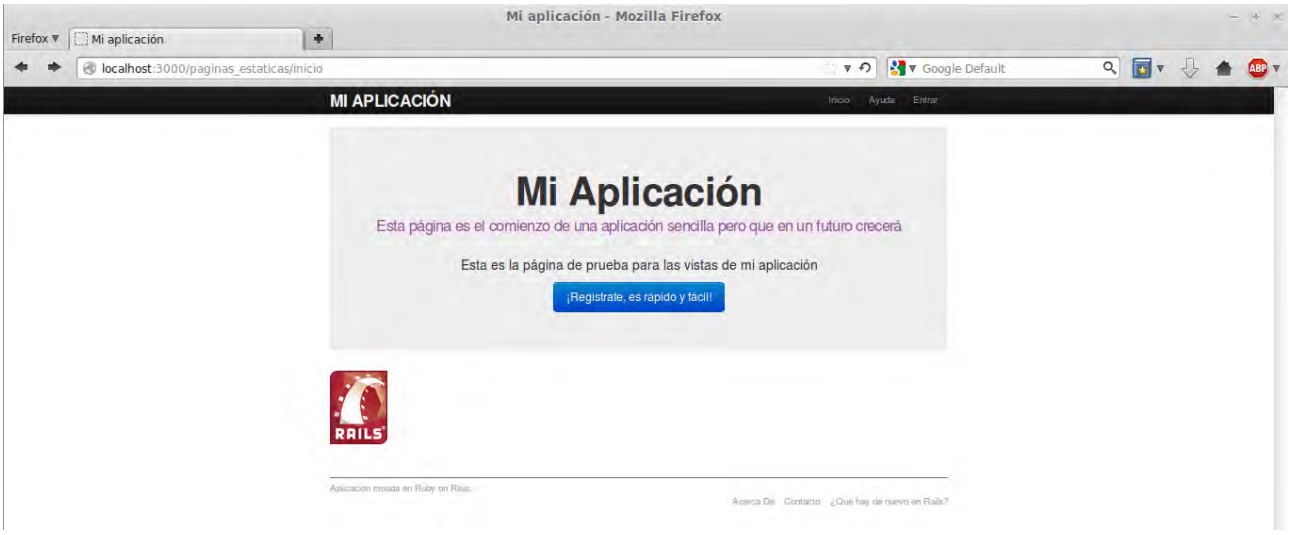


Figura 6.9. Pie de página con estilo.

6.5. Rails: Asset pipeline y Sass

Hemos estado mencionando previamente el uso del *asset pipeline* de *Rails*, la ventaja de éste es optimar la producción y administración de recursos estáticos, que como hemos manejado, pueden ser hojas de estilo *CSS, JavaScript,* imágenes, entre otros. Revisaremos en los apartados siguientes, más detallado el *asset pipeline* y luego cómo usar *Sass*.

6.5.1. Conociendo el asset pipeline

Para entender mejor lo que hace el *asset pipeline*, necesitamos entender los siguientes conceptos:

Directorios de activos (*Asset Directories*)

Para la versión que manejamos de *Rails* (4,0) tenemos tres directorios para activos estáticos[51]:

- lib/assets: aquí se encuentras las bibliotecas desarrolladas por el equipo de trabajo.
- vendor/assets: activos creados por terceros
- app/assets: activos específicos de la aplicación que estemos desarrollando actualmente.

Es por lo anterior el archivo **estilo.css.scss** se colocó en la ruta app/assets/stylesheets al ser parte especifica de nuestro proyecto.

Pre procesamiento

Una vez que los *assets* están en sus respectivos lugares, éstos se preparan para la plantilla de nuestro sitio a través de distintos motores de procesamiento y usando archivos *manifest* que se combinan para su envío al navegador. Para especificar que procesador vamos a usar, utilizamos las extensiones de archivo, principalmente son tres: **.coffe** para el procesador *CoffeeScript*[52], **.scss** para *Sass* y **.erb** para *Ruby* incrustado (*Ruby embedded*).

Podemos encadenar distintos procesadores a través de la extensión:

```
miarchivo.js.erb
```

```
miarchivo.js.erb.coffee
```

[51] En versiones previas de *Rails* se utilizaba el directorio /public

[52] *CoffeeScript* es un lenguaje de programación que termina compilándose en código JavaScript. Para más información, visitar: http://coffeescript.org/

Eficiencia

Un problema que surge al programador en la etapa de producción es cuando el método tradicional para organizar *CSS* y *JavaScript* implica dividir la funcionalidad en archivos distintos usando un formato correcto y confuso. Con el *asset pipeline* todos los estilos de nuestra aplicación (producción) quedan envueltos en un solo archivo *CSS* (**application.css**) y todos los archivos *JavaScript* de igual forma, se guardan en otro archivo único (**javascripts.js**). Esto nos ayuda a tener archivos con un formato correcto para el programador y archivos únicos para producción.

6.5.2. Sass

Syntactically awesomes stylesheets (*Sass*, por sus siglas en inglés), en un lenguaje diseñado para escribir y mejorar hojas de estilo *CSS*. Sólo se harán ver tres de las mejoras más importantes, de las cuales la tercera se verá más adelante. *Sasss* soporta el formato *SCSS* el cual sólo se encarga de añadir nuevas características a *CSS*, por lo que cualquier *CSS* es también un archivo *SCSS*, esto nos ayuda mucho por si tenemos hojas de estilo existentes. Anteriormente usamos *SCSS* desde el principio para tener las ventajas de *Bootstrap*.

Anidar

Algo que podemos notar cuando usamos muchas clases con estilos es que a la larga se vuelven confusos. Para remediar eso, se anidan las clases y se aplican las reglas de estilo a los elementos anidados. Por ejemplo:

```
1  .left {
2    text-align: left;
3  }
4
5  .left h1 {
6    margin-bottom: 10px;
7  }
```

Con *Sass* quedaría:

```
1  .left {
2    text-align: left;
3    h1 {
4      margin-bottom: 10px;
5    }
6  }
```

La regla anidada `h1` se hereda de `.left` de manera automática.

Notamos que anidar es más conveniente ya que hace más sencilla la sintaxis. Ahora, existe otra forma de anidar cuando tenemos por ejemplo un id dos veces, la primera por si misma y la segunda con un atributo. Por ejemplo:

```
56  /* Estilo para encabezado */
57
58  #encabezado {
59    float: left;
60    margin-right: 10px;
61    font-size: 1.8em;
62    color: #F2F2F2;
63    text-transform: uppercase;
64    letter-spacing: -1px;
65    padding-top: 9px;
66    font-weight: bold;
67    line-height: 1;
68  }
69
70  #encabezado:hover {
71    color: #FFF;
72    text-decoration: none;
73  }
74
```

En este caso para anidar la segunda regla (`:hover`) debemos de hacer referencia al elemento padre (`#encabezado` en el ejemplo) a través del símbolo "&":

```
56  /* Estilo para encabezado */
57
58  #encabezado {
59    float: left;
60    margin-right: 10px;
61    font-size: 1.8em;
62    color: #F2F2F2;
63    text-transform: uppercase;
64    letter-spacing: -1px;
65    padding-top: 9px;
66    font-weight: bold;
67    line-height: 1;
68    &:hover {
69      color: #FFF;
70      text-decoration: none;
71    }
72  }
73
```

Podemos realizar lo mismo pero con el pie de página que habíamos creado anteriormente:

```
/* Estilo para pie de página */

footer {
  margin-top: 50px;
  padding-top: 5px;
  border-top: 1px solid #6E6E6E;
  color: #999;
  a {
    color: #999;
    &:hover {
      color: #000001;
    }
  }
  small {
    float: left;
  }
  ul {
    float: right;
    list-style: none;
    li {
      float: left;
      margin-left: 15px;
    }
  }
}
```

6.5.2. Variables

Como cualquier lenguaje, *Sass* nos permite definir y hacer uso de variables para no complicarnos la vida con duplicaciones y escribir código que se entienda mejor. Para definir una variable, usamos la siguiente sintaxis:

```
$nombreVariable: #atributo;
```

El uso de variables nos facilita por ejemplo, no escribir más de dos veces el mismo valor en una hoja de estilo. Si tuviéramos por ejemplo:

```
h1{
  color: #8D00FF;
}

h3 {
  color: #8D00FF;
}
```

Podríamos construir la siguiente variable:

```
$color_purpura: #8D00FF;
```

Nos quedaría:

```
1   $color_purpura: #8D00FF;
2
3   h1{
4     color: $color_purpura;
5   }
6
7   h3 {
8     color: $color_purpura;
9   }
10
11
```

No sólo es necesario usar variables cuando se repiten valores, podemos usarlos incluso cuando no tenemos esa situación, esto es porque nos ayudan a entender mejor el código usando nombres comunes para definir cualquier atributo.

El marco de trabajo *Bootstrap* cuenta con una gran cantidad de variables predefinidas para su uso directo, podemos revisarlas a través de http://bootstrapdocs.com/v2.0.4/docs/less.html. Aunque la página anterior tiene las variables para su uso con *LESS* y no para *Sass*, no es problema ya que gracias a la *gema bootstrap-sass* podemos obtener los equivalentes con sólo sustituir el símbolo "@" de *LESS* por el símbolo "$" para *Sass*.

Haciendo uso de la técnica de anidar y del uso de variables, podemos hacer que nuestra hoja de estilo **custom.css.scss** quede de la siguiente forma:

estilo.css.scss

```scss
@import "bootstrap";

/* Estilo general */

$color_gris: #F2F2F2;

$color_gris_oscuro: #6E6E6E;

$color_blanco: #999;

$color_negro: #000001;

html {
  overflow-y: scroll;
}

body {
  padding-top: 60px;
}

section {
  overflow: auto;
}

textarea {
  resize: vertical;
}

.center {
  text-align: center;
  h1 {
    margin-bottom: 10px;
  }
}

```

```
35
36  /* Estilo para texto */
37
38  h1, h2, h3, h4, h5, h6 {
39    line-height: 1;
40  }
41
42  h1 {
43    font-size: 5em;
44    letter-spacing: -2px;
45    margin-bottom: 30px;
46    text-align: center;
47  }
48
49  h2 {
50    font-size: 1.3em;
51    letter-spacing: -1px;
52    margin-bottom: 30px;
53    text-align: center;
54    font-weight: normal;
55    color: #959;
56  }
57
58  p {
59    font-size: 1.1em;
60    line-height: 1.7em;
61  }
62
63  /* Estilo para encabezado */
64
65  #encabezado {
66    float: left;
67    margin-right: 10px;
68    font-size: 1.8em;
69    color: $color_gris;
70    text-transform: uppercase;
71    letter-spacing: -1px;
72    padding-top: 9px;
73    font-weight: bold;
74    line-height: 1;
75    &:hover {
76      color: $color_gris;
77      text-decoration: none;
78    }
79  }
```

```
80
81  /* Estilo para pie de página */
82
83  footer {
84    margin-top: 50px;
85    padding-top: 5px;
86    border-top: 1px solid $color_gris_oscuro;
87    color: $color_blanco;
88    a {
89      color: $color_blanco;
90      &:hover {
91        color: $color_negro;
92      }
93    }
94    small {.
95      float: left;
96    }
97    ul {
98      float: right;
99      list-style: none;
100     li {
101       float: left;
102       margin-left: 15px;
103     }
104   }
105 }
```

6.6. Enlaces de estilo

Recordemos que a los enlaces que añadimos a nuestras páginas no tenían ninguna *URL* y en su lugar colocamos el símbolo "**#**" , podemos sustituirlo por:

```
<a href =”/paginas_estaticas/acercade”>Acerca De</a>
```

Ahora, nos sería más fácil tener *URLs* de la forma /acercade en lugar de /paginas_estaticas/acercade. *Rails* usa *rutas personalizadas* que implican código como este:

```
<%= link_to “Acerca De”, acercade_path%>
```

Las rutas que usaremos son:

Tabla 6.1. Rutas personalizadas.

Página	URL	Ruta personalizada
Inicio	/inicio	raiz_path
Acerca De	/acercade	acercade_path
Ayuda	/ayuda	ayuda_path
Contacto	/contacto	contacto_path
Registro	/registro	registro_path
Entrar	/entrar	entrar_path

El nombre *path* se asigna automáticamente, lo veremos en el siguiente apartado. Ahora para continuar, crearemos la página *Contacto* que la creamos de manera similar a la página *Acerca De* del capítulo 5. Primeramente añadimos la ruta al archivo **routes.rb**, luego creamos la acción correspondiente en el controlador de páginas estáticas y por último creamos la vista.

routes.rb

```
1  BaseApp::Application.routes.draw do
2    get "paginas_estaticas/inicio"
3    get "paginas_estaticas/ayuda"
4    get "paginas_estaticas/acercade"
5    get "paginas_estaticas/contacto"
6
```

paginas_estaticas_controller.rb

```
1  class PaginasEstaticasController < ApplicationController
2    def inicio
3    end
4
5    def ayuda
6    end
7
8    def acercade
9    end
10
11   def prueba
12   end
13
14   def contacto
15   end
16 end
```

contacto.html.erb

```
1  <% provide(:title, 'Contacto') %>
2  <h1>Contacto</h1>
3  <p>
4    Página de contacto para el desarrollador del proyecto.
5  </p>
```

6.6.1. Personalizando rutas en Rails

Ya que sabemos qué rutas queremos personalizar, debemos realizar los cambios a través del archivo **routes.rb**. Para definir las rutas personalizadas hacemos lo siguiente:

La línea:

```
get 'paginas_estaticas/contacto'
```

Se reemplaza por:

```
match '/contacto', to: 'paginas_estaticas#contacto', via:
'get'
```

Lo anterior hace los arreglos para una página valida en /ayuda que responde a peticiones de tipo *get*, y genera también un ruta llamada *contacto_path* con la línea `match '/contacto/'` (de aquí el porque de las rutas de la Tabla 6.1) que nos retorna la ruta a la página. Apliquemos lo anterior a todas nuestras rutas a excepción de la página de Inicio que más adelante veremos porque:

routes.rb

```
1  BaseApp::Application.routes.draw do
2    get "paginas_estaticas/inicio"
3    match '/ayuda', to: 'paginas_estaticas#ayuda', via: 'get'
4    match '/acercade', to: 'paginas_estaticas#acercade', via: 'get'
5    match '/contacto', to: 'paginas_estaticas#contacto', via: 'get'
```

Además de generarse las rutas personalizadas del tipo *pagina_path* se genera otra del tipo *pagina_url* que contiene la ruta completa de la página, por ejemplo:

```
contacto_path = '/contacto'
contacto_url = 'http://localhost:3000/contacto'
```

La *URL* completa, se utiliza cuando se hacen redireccionamientos, esto porque después de redireccionar el estándar *HTML* requiere de una *URL* completa.

Ahora, falta agregar una ruta personalizada para nuestra página de *inicio*, podríamos hacerlo como en las demás páginas pero cuando se trata de la página raíz, *Rails* tiene una forma especificada en los comentarios del archivo:

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

# root 'welcome#index'
```

Con lo anterior nos quedaría el archivo **routes.rb** como se muestra:

routes.rb

```
1  BaseApp::Application.routes.draw do
2    root 'paginas_estaticas#inicio'
3    match '/ayuda', to: 'paginas_estaticas#ayuda', via: 'get'
4    match '/acercade', to: 'paginas_estaticas#acercade', via: 'get'
5    match '/contacto', to: 'paginas_estaticas#contacto', via: 'get'
6
```

Ahora ya no se abrirá la página que *Rails* nos genera por default cuando creamos un nuevo proyecto. Con esto hemos llenado los enlaces del *layout.*

6.6.2. Usando rutas personalizadas

Ya teniendo las rutas personalizadas, vamos a colocarlas en nuestro *layout*, para hacerlo sólo debemos de completar el segundo argumento de la función `link_to` con las rutas apropiadas, es decir, sustituir esto:

```
<%= link_to “Contacto”, ‘#’%>
```

Con esto (de la misma manera con los demás enlaces(:

```
<%= link_to “Contacto”, contacto_path%>
```

En nuestro parcial con el encabezado, podemos añadir los enlaces hacia las páginas *Ayuda* e *Inicio*, éste último se enlaza al id “encabezado” por convención en páginas web. Por el momento, el enlace que apunta hacia la página de inicio de sesión seguirá teniendo el símbolo “**#**” como enlace.

_etiquetaHeader.html.erb

```
<header class="navbar navbar-fixed-top navbar-inverse">
  <div class="navbar-inner">
    <div class="container">
      <%= link_to "Mi Aplicación", '#', id: "encabezado" %>
      <nav>
        <ul class="nav pull-right">
          <li><%= link_to "Inicio",    root_path %></li>
          <li><%= link_to "Ayuda",    ayuda_path %></li>
          <li><%= link_to "Entrar", '#' %></li>
        </ul>
      </nav>
    </div>
  </div>
</header>
```

Luego, continuamos con el parcial del pie de página, añadiendo los enlaces correspondientes a las páginas a*cercade* y c*ontacto.*

Para verificar que las rutas personalizadas tengan efecto, abrimos el navegador y probamos los enlaces uno a uno.

_etiquetaFooter.html.erb

```
<footer class="footer">
  Aplicación creada en <a href="http://rubyonrails.org/">Ruby on Rails.</a>
  <nav>
    <ul>
      <li><%= link_to "Acerca De",   acercade_path %></li>
      <li><%= link_to "Contacto", contacto_path %></li>
      <li><a href="http://rubyonrails.org/">¿Qué hay de nuevo en Rails?</a></li>
    </ul>
  </nav>
</footer>
```

Figura 6.10. Página *contacto*.

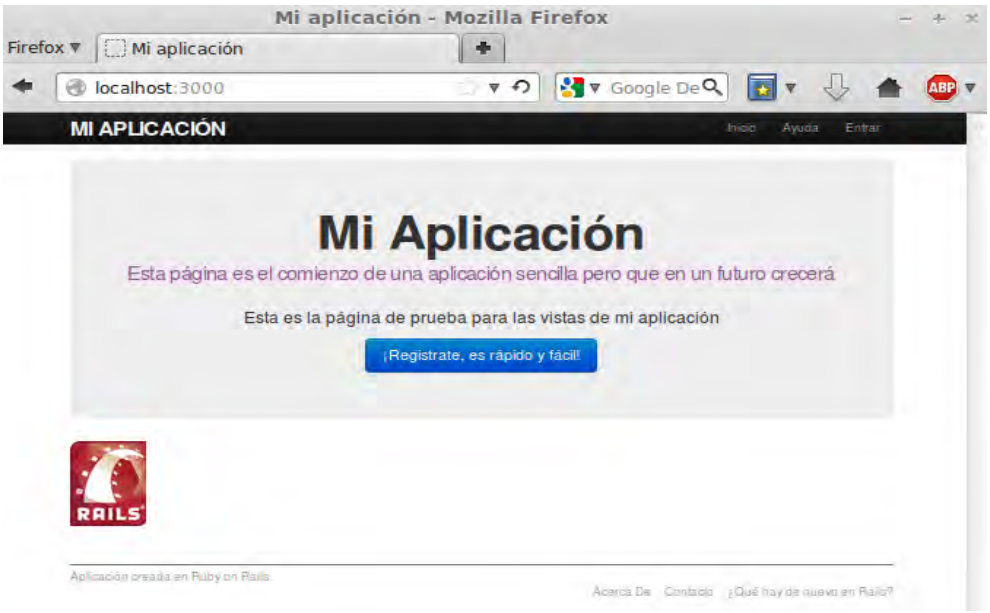


Figura 6.11. Página de inicio modificada.

6.7. Registro de usuarios

Hemos llegado a la parte en dónde empezaramos a darle forma a nuestra aplicación web y el primer paso será la adición de usuarios. Debemos de generar el controlador (que es el segundo después del controlador `PaginasEstáticas`. Usaremos de nueva cuenta el *script* `generate`, también usando la arquitectura *REST* tendremos una acción `new` para los nuevos usuarios:

```
$ rails generate controller Usuarios new
```

```
pavel@pavel-K61IC ~/lab_rails/base_app $ rails generate controller Usuarios new
      create  app/controllers/usuarios_controller.rb
       route  get "usuarios/new"
      invoke  erb
      create    app/views/usuarios
      create    app/views/usuarios/new.html.erb
      invoke  test_unit
      create    test/controllers/usuarios_controller_test.rb
      invoke  helper
      create    app/helpers/usuarios_helper.rb
      invoke    test_unit
      create      test/helpers/usuarios_helper_test.rb
      invoke  assets
      invoke    coffee
      create      app/assets/javascripts/usuarios.js.coffee
      invoke    scss
      create      app/assets/stylesheets/usuarios.css.scss
```

Teniendo la siguiente estructura nuestro nuevo controlador:

usuarios_controller.rb

```
class UsuariosController < ApplicationController
  def new
  end
end
```

Y una vista como se muestra:

new.html.erb

```
<h1>Usuarios#new</h1>
<p>Find me in app/views/usuarios/new.html.erb</p>
```

6.7.1. Ruta personalizada de registro

Con lo anterior tenemos una ruta de la forma /usuarios/new que nos va a ayudar en la creación de nuevos usuarios. Ahora para tener la ruta para el registro de usuarios con un `match '/registro'`, abrimos el archivo **routes.rb** y añadimos:

routes.rb

```
BaseApp::Application.routes.draw do
  get "usuarios/new"
  root 'paginas_estaticas#inicio'
  match '/registro', to: 'usuarios#new', via: 'get'
  match '/ayuda', to: 'paginas_estaticas#ayuda', via: 'get'
  match '/acercade', to: 'paginas_estaticas#acercade', via: 'get'
  match '/contacto', to: 'paginas_estaticas#contacto', via: 'get'
```

Cambiemos la vista:

new.html.erb

```
<% provide (:title, 'Registro') %>
<h1>Usuarios#new</h1>
<p>Crear nuevos usuarios...</p>
```

Ahora, en la página *inicio* vamos a añadir el enlace que apunta a la página de registro a través de la ruta personalizada *registro_path*:

inicio.html.erb

```
1  <div class="center hero-unit">
2
3    <h1>Mi Aplicación</h1>
4    <h2>Esta página es el comienzo de una aplicación sencilla pero que en un futuro crecerá</h2>
5    <p>
6      Esta es la página de prueba para
7      las vistas de mi aplicación
8    </p>
9    <%= link_to "¡Registrate, es rápido y fácil!", registro_path, class: "btn btn-large btn-primary" %>
10   </div>
11
12   <%= link_to image_tag("rails_icono.png", alt: "Rails Icono"), 'http://rubyonrails.org/' %>
13
```

Verificamos que el enlace funciona y habremos terminado.

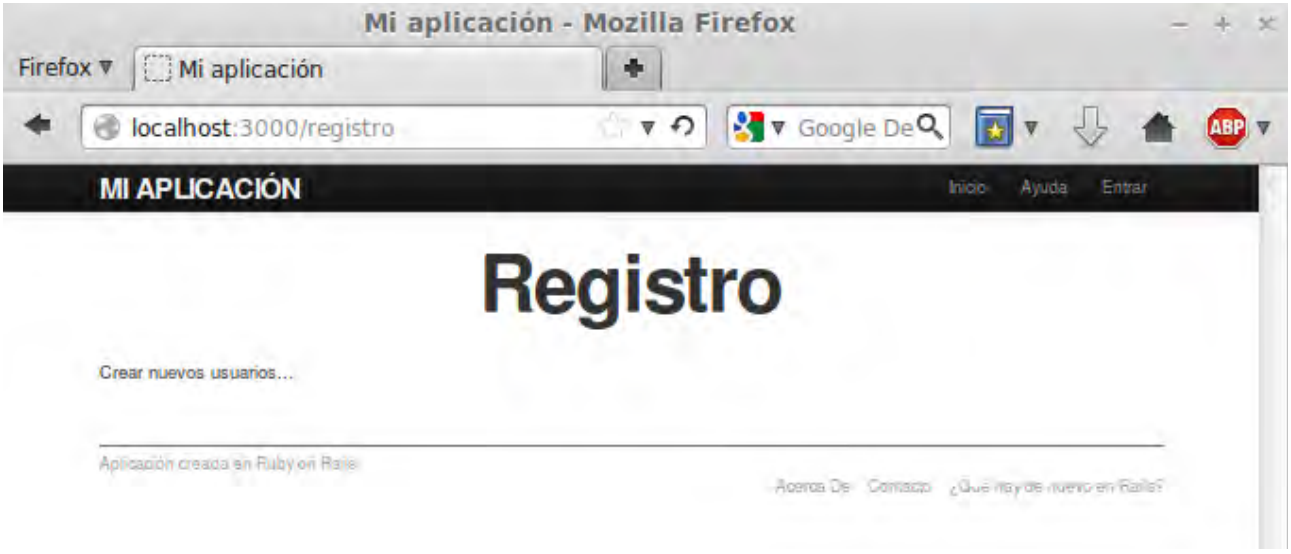


Figura 6.12. Página de registro de usuarios.

6.8. Guardando cambios

Añadimos los archivos a *Git* y confirmamos.

```
$ git commit -am 'Estilos y rutas completado'
```

Luego, añadir los cambios de la rama *usando_estilos* a la rama *master* y por último subir a nuestro repositorio remoto el proyecto:

```
$ git checkout master
$ git merge usando_estilos
$ git push origin master
```

Capítulo 7: Usuarios

7. Desarrollo

Anteriormente dejamos en inicios lo que vendría a ser el registro de usuarios. Vamos a crear el modelo para los usuarios de nuestra aplicación y las acciones necesarias para guardar la información.

Hacemos la nueva rama como de costumbre.

```
$ git checkout master

$ git checkout -b modelo_usuarios
```

7.1 Modelo para usuarios

La solución que nos propone *Rails* para el problema de la persistencia en nuestra información es el uso de una base de datos para el almacenamiento de la misma a largo plazo, y para realizar está interacción con la base de datos, se usa una biblioteca llamada *Active Record* que en capítulos previos hemos estado utilizando. *Active Record* viene con varias funciones para crear, almacenar y encontrar objetos de datos, sin tener que usar el lenguaje *SQL*. *Rails* por su parte tiene algo llamado, *migraciones*que permiten alojar definiciones de datos que serán escritas en *Ruby*.

7.1.1. Migraciones

El objetivo de esta sección es crear un modelo para usuarios que cumpla con la persistencia de nuestros datos. Empezamos el modelo de un usuario con dos atributos, un *nombre* y un *email* que servirá como nombre de usuario único (el atributo de contraseña se añadirá más adelante). Cuando usamos *Rails* para modelar usuarios, no se necesita identificar a los atributos de manera explícita, es decir, podemos omitir el uso del método `attr_accesor`.

Para almacenar nuestros datos, *Rails* usa bases de datos relacionales por default, para almacenar a los usuarios con sus nombres y correos, añadimos las columnas *nombre* e *email*, cada fila de la tabla corresponde a un usuario. Con lo anterior *ActiveRecord* automáticamente asignará los atributos del objeto `Usuario` para nosotros. El bosquejo de los datos se muestra en la Figura 4.10 del capítulo 4

Vamos a generar el modelo *Usuario* con el siguiente comando:

```
$ rails generate model Usuario nombre:string email:string
```

Al especificar en el comando los atributos (opcionales) `nombre:string` y `email:string`, estamos especificando a *Rails* los atributos y el tipo que va a tener nuestro modelo (cadenas en nuestro caso).

Con el comando anterior, se crea un nuevo archivo en la ruta db/migrate/ al que se le llama *migración*. Las llamadas migraciones nos dan una alternativa para alterar la estructura de la base de datos, esto sobretodo para adaptar la base a los requerimientos necesarios.

20140728154614_create_usuarios.rb

```
1  class CreateUsuarios < ActiveRecord::Migration
2    def change
3      create_table :usuarios do |t|
4        t.string :nombre
5        t.string :email
6
7        t.timestamps
8      end
9    end
10 end
```

Observamos que el nombre del archivo lleva como prefijo un *timestamp*[53] que nos proporciona la fecha de cuándo se generó la migración. Podemos ver que la migración consiste en un método `change` que determina los cambios que se harán a nuestra base de datos. Podemos ver del código anterior, que el método `change` utiliza también un método propio de *Rails* llamado `create_table` que creará una tabla para almacenar a los usuarios. El método `create_table` acepta un bloque con una variable de bloque `t` que dentro del bloque esta variable es usada por método `create_table` para crear las dos columnas (*nombre*, *email*) de tipo `string`. Por último, la línea `t.timestamps` es un comando que crea dos columnas automáticamente que son *created_at* y *updated_at*, que son muy útiles ya que contienen el *timestamp* de cuando un usuario se crea o se modifican sus datos.

Para ejecutar nuestra migración, usamos el comando `rake`:

```
$ bundle exec rake db:migrate
```

```
pavel@pavel-K61IC ~/lab_rails/base_app $ bundle exec rake db:migrate
== 20140728154614 CreateUsuarios: migrating ===================================
-- create_table(:usuarios)
   -> 0.0009s
== 20140728154614 CreateUsuarios: migrated (0.0010s) ==========================
```

Cuando ejecutamos este comando por primera vez, nos crea el archivo db/development.sqlite3. En este archivo se encuentra la estructura de la base de datos. La

[53] Secuencia de caracteres, que denotan la hora y fecha en la cual ocurrió determinado evento. .

mayoría de las migraciones pueden revertirse y deshacer los cambios con una tarea de tipo *Rake* que se llama `db:rollback`:

```
$ bundle exec rake db:rollback
```

Teniendo en cuenta que si deshacemos la migración, debemos de migrar de nuevo la base de datos. Para este momento nuestra base cuenta con lo siguiente:

Tabla 7.1. Modelo Usuarios.

Usuarios	
id	integer
nombre	string
email	string
created_at	datetime
updated_at	datetime

El código del modelo Usuario se puede ver en el archivo **usuario.rb**, siendo éste muy compacto y de alguna manera, sencillo. La herencia que se tiene de `ActiveRecord:Base`, añade toda la funcionalidad de esa clase.

app/models/usuario.rb

usuario.rb

```
1  class Usuario < ActiveRecord::Base
2  end
```

Código 7.1. **usuario.rb**

7.1.2. Creación de usuarios

Para esta sección usaremos la consola de *Rails* para explorar los datos en nuestros modelos, para evitar realizar cualquier cambio en la base, usaremos la consola en modo *sandbox*:

```
$ rails console --sandbox
```

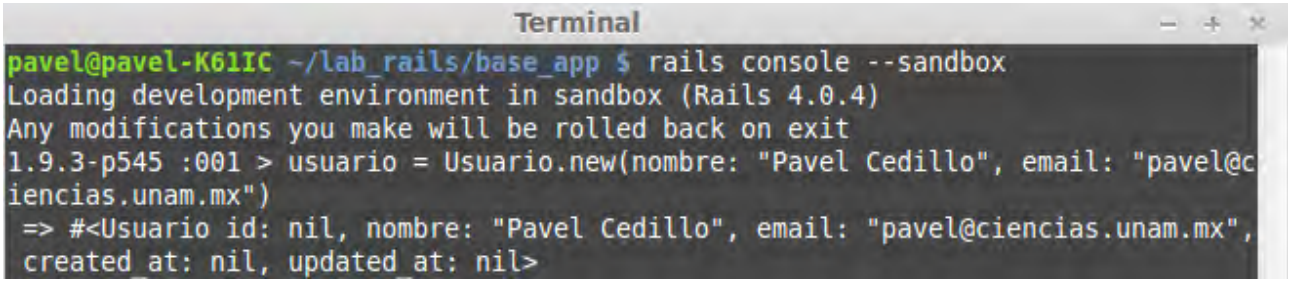
La consola de *Rails* carga todo el ambiente de trabajo cuando la iniciamos, incluyendo los modelos. Esto nos ayuda para crear cualquier objeto usuario con sólo escribir:

```
>> Usuario.new
```

El valor que tendrán los parámetros del usuario anterior serían todos `nil` (incluyendo las columnas creadas automáticamente: *id, created_at, updated_at*), si en cambio, nosotros asignamos los valores, éstos no se guardan en la base de datos porque `Usuario.new` no toca la base de datos, sólo crea un objeto de *Ruby* en memoria. Para salvar los cambios hechos en la base de datos, debemos usar el método `save` sobre la variable `usuario`:

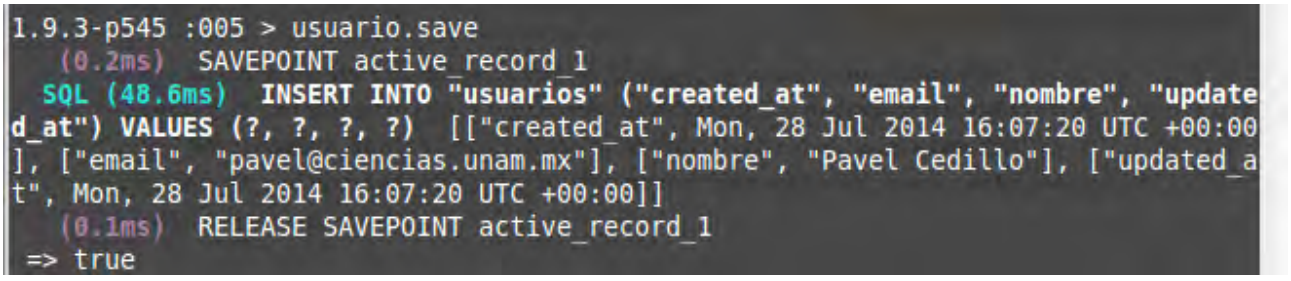
Creamos un usuario nuevo y lo asignamos a la variable `usuario`:

```
>> usuario = Usuario.new(nombre: “Pavel Cedillo”, email:
“pavel@ciencias.unam.mx”)
```



Ahora, guardamos al usuario anterior en nuestra base de datos:

```
>> usuario.save
```



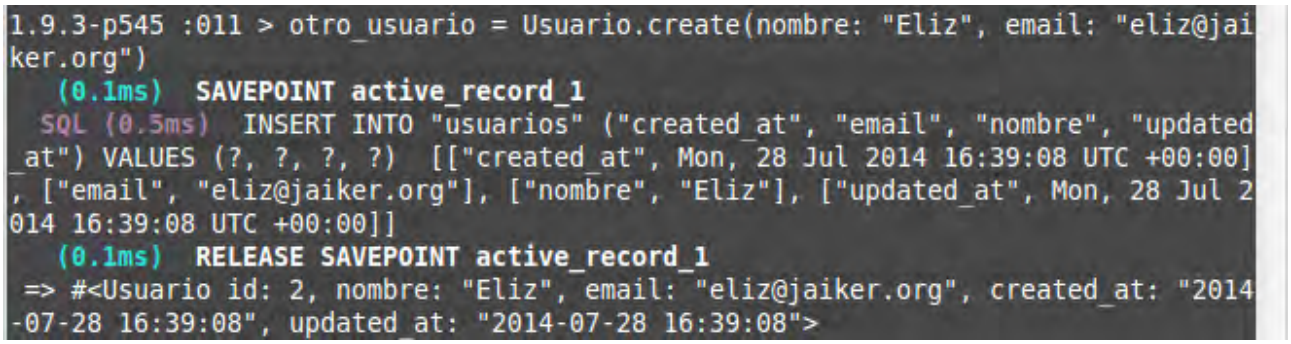
El comando anterior regresa `true` si se salva la información y `false` si no. Ahora que los cambios se guardaron en nuestra base de datos, las columnas: *id, created_at y updated_at*; han sido actualizadas y guardan la información del nuevo registro.

Ahora podemos acceder a los atributos del modelo *Usuario* a través de la consola de *Rails* de la siguiente forma:

```
>> usuario.nombre
=> “Pavel Cedillo”

>> usuario.email
=> “pavel@ciencias.unam.mx”
```

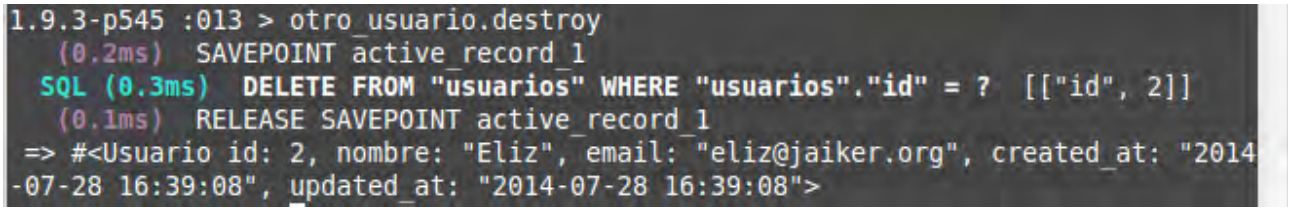
Hemos visto que debemos crear a nuestro usuario primero y después, salvar a éste en la base, podemos realizar ambas acciones a través de la función `create` como se muestra a continuación:

```
>> otro_usuario = Usuario.create(nombre: “Eliz”, email:
“eliz@jaiker.org”)
```



Como podríamos suponer, la acción contraria a crear es eliminar, y para eliminar el objeto usuario de nuestra base de datos usamos la función `destroy`:

```
>> otro_usuario.destroy
```



Incluso al haber destruido nuestro objeto, éste existe en la memoria:

```
>> otro_usuario
```

```
1.9.3-p545 :016 > otro_usuario
 => #<Usuario id: 2, nombre: "Eliz", email: "eliz@jaiker.org", created_at: "2014
-07-28 16:39:08", updated_at: "2014-07-28 16:39:08">
```

7.1.3. Encontrando Usuarios

Para encontrar objetos, *Active Record* nos provee de distintas opciones para completar tal tarea. Busquemos al primer usuario que creamos y luego al usuario que eliminamos anteriormente:

```
>> Usuario.find(1)
```

```
1.9.3-p545 :019 > Usuario.find(1)
  Usuario Load (0.2ms)  SELECT "usuarios".* FROM "usuarios" WHERE "usuarios"."id
" = ? LIMIT 1  [["id", 1]]
 => #<Usuario id: 1, nombre: "Pavel Cedillo", email: "pavel@ciencias.unam.mx", c
reated_at: "2014-07-28 16:07:20", updated_at: "2014-07-28 16:07:20">
```

Ahora busquemos al usuario con id 2 (el que fue destruido):

```
>> Usuario.find(2)
```

```
1.9.3-p545 :021 > Usuario.find(2)
  Usuario Load (0.1ms)  SELECT "usuarios".* FROM "usuarios" WHERE "usuarios"."id
" = ? LIMIT 1  [["id", 2]]
ActiveRecord::RecordNotFound: Couldn't find Usuario with id=2
        from /home/pavel/.rvm/gems/ruby-1.9.3-p545@rubyrails_lab/gems/activereco
rd-4.0.4/lib/active_record/relation/finder_methods.rb:199:in `raise_record_not_f
ound exception!'
```

Como se muestra arriba, *Active Record* nos lanza una excepción en la ejecución, en este caso, la falta del registro con el *id* especificado genera la excepción *ActiveRecord::RecordNotFound*.

Podemos incluso buscar usuarios a través de atributos específicos (como por ejemplo, el *email*):

```
>> Usuario.find_by_email(“pavel@ciencias.unam.mx”)
```

```
1.9.3-p545 :023 > Usuario.find_by_email("pavel@ciencias.unam.mx")
  Usuario Load (0.3ms)  SELECT "usuarios".* FROM "usuarios" WHERE "usuarios"."email" =
'pavel@ciencias.unam.mx' LIMIT 1
 => #<Usuario id: 1, nombre: "Pavel Cedillo", email: "pavel@ciencias.unam.mx", created_
at: "2014-07-28 16:07:20", updated_at: "2014-07-28 16:07:20">
```

El método anterior (`find_by_email`) es creado por *Active Record* basándose en el atributo *email* en la tabla de *usuarios*. De manera análoga, *Active Record* crea el método `find_by_nombre`. En *Rails* se usa preferentemente el método `find_by` con un *hash* como atributo:

```
>> Usuario.find_by(nombre: “Pavel Cedillo”)
```

```
1.9.3-p545 :025 > Usuario.find_by(nombre: "Pavel Cedillo")
  Usuario Load (0.2ms)  SELECT "usuarios".* FROM "usuarios" WHERE "usuarios"."nombre" =
 'Pavel Cedillo' LIMIT 1
 => #<Usuario id: 1, nombre: "Pavel Cedillo", email: "pavel@ciencias.unam.mx", created_
at: "2014-07-28 16:07:20", updated_at: "2014-07-28 16:07:20">
```

Otra forma de encontrar usuarios es con el método `first` que retorna el primer usuario en la base de datos:

```
>> Usuario.first
```

```
1.9.3-p545 :028 > Usuario.first
  Usuario Load (0.3ms)  SELECT "usuarios".* FROM "usuarios" ORDER BY "usuarios"."id" AS
C LIMIT 1
 => #<Usuario id: 1, nombre: "Pavel Cedillo", email: "pavel@ciencias.unam.mx", created_
at: "2014-07-28 16:07:20", updated_at: "2014-07-28 16:07:20">
```

La función `all` retorna un arreglo con todos los usuarios en la base de datos:

```
>> Usuario.all
```

```
1.9.3-p545 :030 > Usuario.all
  Usuario Load (0.2ms)  SELECT "usuarios".* FROM "usuarios"
 => #<ActiveRecord::Relation [#<Usuario id: 1, nombre: "Pavel Cedillo", email: "pavel@c
iencias.unam.mx", created_at: "2014-07-28 16:07:20", updated_at: "2014-07-28 16:07:20">
```

7.1.4. Actualizar usuarios

Ya que tenemos los objetos creados, podemos modificar sus parámetros. Una de las formas para hacerlo, es asignando atributos de forma individual:

```
>> usuario

>> usuario.email = “pavel.cedillo@newmail.com”
```

```
>> usuario.save

>> usuario
```

```
1.9.3-p545 :032 > usuario
 => #<Usuario id: 1, nombre: "Pavel Cedillo", email: "pavel@ciencias.unam.mx", created_
at: "2014-07-28 16:07:20", updated_at: "2014-07-28 16:07:20">
1.9.3-p545 :033 > usuario.email = "pavel.cedillo@nuevoemail.com"
 => "pavel.cedillo@nuevoemail.com"
1.9.3-p545 :034 > usuario.save
   (0.2ms)  SAVEPOINT active_record_1
  SQL (0.5ms)  UPDATE "usuarios" SET "email" = ?, "updated_at" = ? WHERE "usuarios"."id
" = 1  [["email", "pavel.cedillo@nuevoemail.com"], ["updated_at", Mon, 28 Jul 2014 18:1
1:04 UTC +00:00]]
   (0.1ms)  RELEASE SAVEPOINT active_record_1
 => true
1.9.3-p545 :035 > usuario
 => #<Usuario id: 1, nombre: "Pavel Cedillo", email: "pavel.cedillo@nuevoemail.com", cr
eated_at: "2014-07-28 16:07:20", updated_at: "2014-07-28 18:11:04">
```

Una manera de comprobar que hemos hechos los cambios (por si no nos acordamos si salvamos los nuevos datos en la base), usamos la función `reload` que vuelve a cargar al objeto basado en la información contenida en la base de datos.

```
>> usario.email

>> usuario.email = “nuevocorreo@nuevoserv.com”

>> usuario.reload.email
```

```
1.9.3-p545 :037 > usuario.email
 => "pavel.cedillo@nuevoemail.com"
1.9.3-p545 :038 > usuario.email = "nuevocorreo@nuevoserv.com"
 => "nuevocorreo@nuevoserv.com"
1.9.3-p545 :039 > usuario.reload.email
  Usuario Load (0.2ms)  SELECT "usuarios".* FROM "usuarios" WHERE "usuarios"."id" = ? L
IMIT 1  [["id", 1]]
 => "pavel.cedillo@nuevoemail.com"
```

Cuando actualizamos y salvamos nueva información de un usuario, las columnas *created_at* y *updated_at* difieren, como deber ser.

La segunda forma de actualizar a un usuario es usar `update_attributes` como se muestra:

```
>> usuario.update_attributes(nombre: “Menelao”, email:
“menelao@ciencias.unam.mx”)

>> usuario.nombre
```

```
>> usuario.email
```

```
1.9.3-p545 :049 > usuario.update_attributes(nombre: "Menelao", email: "menelao@ciencias
.unam.mx")
   (0.2ms)  SAVEPOINT active_record_1
  SQL (0.4ms)  UPDATE "usuarios" SET "nombre" = ?, "email" = ?, "updated_at" = ? WHERE
"usuarios"."id" = 1  [["nombre", "Menelao"], ["email", "menelao@ciencias.unam.mx"], ["u
pdated_at", Mon, 28 Jul 2014 18:44:16 UTC +00:00]]
   (0.1ms)  RELEASE SAVEPOINT active_record_1
 => true
1.9.3-p545 :050 > usuario.nombre
 => "Menelao"
1.9.3-p545 :051 > usuario.email
 => "menelao@ciencias.unam.mx"
```

Esta forma de actualizar usuarios con `update_attribute` a través de un *hash* de atributos, realiza las dos acciones actualizar y salvar en un sólo paso. Si queremos actualizar sólo un atributo a través de `update_attribute`, lo hacemos de la siguiente manera:

```
>> usuario.update_attribute(:nombre, “Alfredo”)
```

7.2. Validación de usuarios

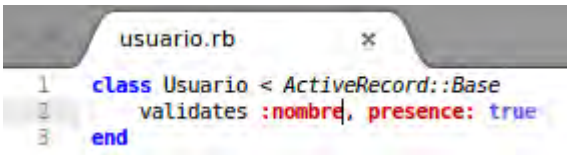
Hasta ahora podemos guardar la información de los usuarios en la base de datos pero está información resulta genérica, cualquier cadena que yo agregue a los atributos *nombre* y *email* será correcta. Más aún, debemos de restringir que los atributos no sean vacíos y que el email cumpla con un estándar de formato apropiado y sobretodo, que no exista duplicidad de emails en la base, si éstos serán los nombres de usuarios para el inicio de sesión.

Active Record nos permite hacer validaciones para enfrentar los “problemas” citados anteriormente. En los apartados siguientes validaremos, *presencia, formato, longitud, unicidad y confirmación* en nuestros atributos.

7.2.1. Validación de presencia

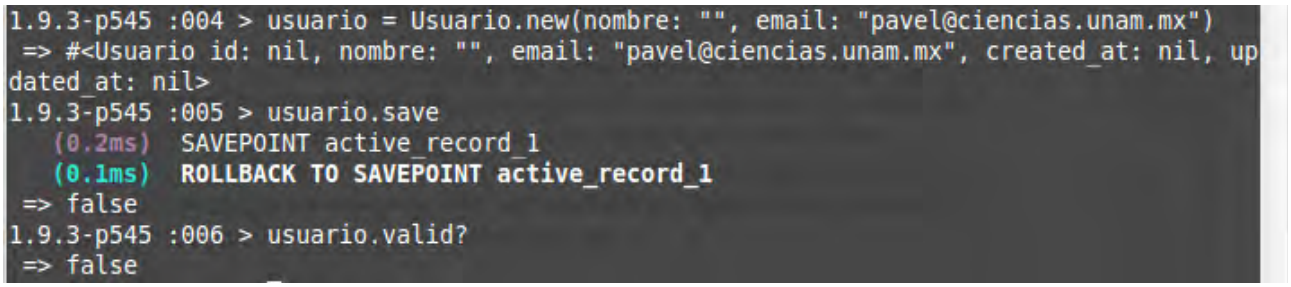
Una de las validaciones más elementales es la presencia de los atributos. Para nuestra aplicación validaremos el nombre y correo antes de que un usuario sea dado de alta en la base de datos.

Para validar la presencia de los atributos, debemos usar el método `validates` con el argumento `presence:true`. El argumento anterior no es más que un *hash* de opciones de un elemento. Modificamos el modelo **usuario.rb** para validar[54]:



Comprobemos que las validaciones tengan efecto, abrimos la consola de *Rails* y tecleamos:

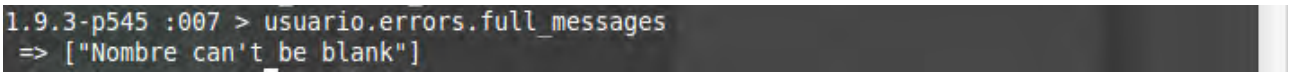
```
$ rails console --sandbox

>> usuario = Usuario.new(nombre: “”, email:
“pavel@ciencias.unam.mx”)

>> usuario.save

>> usuario.valid?
```



Observamos en la imagen que al querer salvar al usuario en la base de datos, nos retorna un mensaje `false`, indicando un fallo al querer salvar. Al final usamos el método `valid?` que retorna `false` cuando un objeto no pasa una o más validaciones y `true` si todas las validaciones pasan.

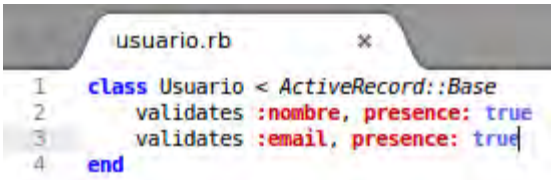
Para ver con detalle el problema, veamos que nos dice el objeto de tipo error que se genera al fallar la verificación de presencia:

```
>> usuario.errors.full_messages
```



[54] Cómo vimos en el Capítulo 3, podemos hacer uso o no de paréntesis. Un equivalente sería: `validates (:name, presence: true)`

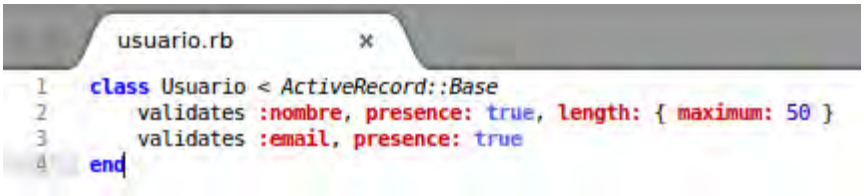
Para la validación del email del usuario es lo mismo que antes:



7.2.2. Validación de longitud

Como nuestro modelo *Usuario* requiere de un nombre para cada usuario, debemos añadir una de las validaciones básicas más usadas que es la validación de la longitud de nombre. Para este ejercicio, tendremos como límite 50 caracteres.

Para hacerlo, modificamos nuestro modelo con la validación correspondiente:



Comprobamos que lo anterior funcione:

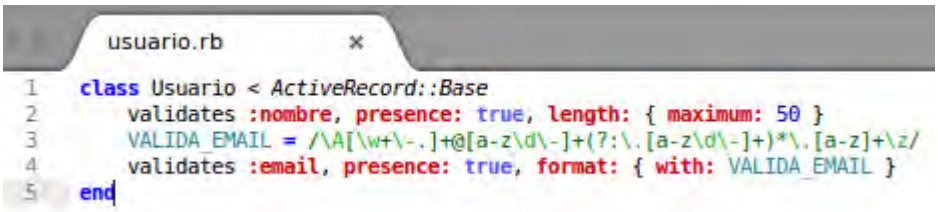
```
$ rails console --sandbox

>> usuario = Usuario.new(nombre: “A” * 50, email:
“alf@server.com”)

>> usuario.save

>> usuario.valid?

>> usuario.error.full_messages
```

```
1.9.3-p545 :006 > usuario = Usuario.new(nombre: "A" * 60, email: "alf@server.com")
 => #<Usuario id: nil, nombre: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA..."
, email: "alf@server.com", created_at: nil, updated_at: nil>
1.9.3-p545 :007 > usuario.save
   (0.2ms)  SAVEPOINT active_record_1
   (0.1ms)  ROLLBACK TO SAVEPOINT active_record_1
 => false
1.9.3-p545 :008 > usuario.valid?
 => false
1.9.3-p545 :009 > usuario.errors.full_messages
 => ["Nombre is too long (maximum is 50 characters)"]
```

7.2.3. Validación de formato para correo electrónico

Las validaciones del atributo `nombre` fueron mininas, sólo validamos que el nombre no fuera vacío y que no fuera mayor a 60 caracteres. Para validar el correo electrónico, debemos considerar otro tipo de requisitos. Además de no estar vacío este atributo (*email*), debe cumplir con un formato correcto, es decir, de la forma: `miusuario@servidor.com`.

Para cumplir nuestro objetivo, haremos uso de *expresiones regulares* (también conocidas como *regex*) para definir el formato, junto con el método `validates` y el argumento `:format`.



Reconocemos las constante VALIDA_EMAIL por las mayúsculas, indicando las partes que debe de tener el correo en sintaxis. Para verificar que la expresión regular funcione, podemos usar el editor online de http://www.regexplanet.com/advanced/ruby/index.html o *Rubular* en http://rubular.com/.

De igual forma que las validaciones anteriores, comprobamos que funcione la validación de correo:

```
>> usuario = Usuario.new(nombre: “Alfredo” , email:
“alf@@server.com”)

>> usuario.save

>> usuario.valid?

>> usuario.error.full_messages
```

```
1.9.3-p545 :007 > usuario = Usuario.new(nombre: "Alfredo", email: "alf@@server.com")
 => #<Usuario id: nil, nombre: "Alfredo", email: "alf@@server.com", created_at: nil, up
dated_at: nil>
1.9.3-p545 :008 > usuario.save
   (0.2ms)  SAVEPOINT active_record_1
   (0.1ms)  ROLLBACK TO SAVEPOINT active_record_1
 => false
1.9.3-p545 :009 > usuario.valid?
 => false
1.9.3-p545 :010 > usuario.errors.full_messages
 => ["Email is invalid"]
```

Con la validación de formato hecha, sólo falta validar la unicidad del correo.

7.2.4. Validación de unicidad

La unicidad en los correos electrónicos nos ayuda para poder utilizar dichos correos como nombres de usuario (*usernames*). Para lograr tal cometido, usamos la opción `:unique` del método `validates`.

Modificamos nuevamente nuestro archivo, añadiendo la validación de unicidad al correo:

usuario.rb

```
1  class Usuario < ActiveRecord::Base
2      validates :nombre, presence: true, length: { maximum: 50 }
3      VALIDA_EMAIL = /\A[\w+\-.]+@[a-z\d\-]+(?:\.[a-z\d\-]+)*\.[a-z]+\z/
4      validates :email, presence: true, format: { with: VALIDA_EMAIL },
5                        uniqueness: true
6  end
```

Hemos validado la unicidad pero no hemos acabado aún, debemos de restringir el atributo *email* que sea sensible a mayúsculas y minúsculas. Resulta muy sencillo con la opción `:case_sensitive` dentro de la validación de unicidad como se muestra:

usuario.rb

```
1  class Usuario < ActiveRecord::Base
2      validates :nombre, presence: true, length: { maximum: 50 }
3      VALIDA_EMAIL = /\A[\w+\-.]+@[a-z\d\-]+(?:\.[a-z\d\-]+)*\.[a-z]+\z/
4      validates :email, presence: true, format: { with: VALIDA_EMAIL },
5                        uniqueness: { case_sensitive: false }
6  end
```

Comprobamos que las validaciones hagan su trabajo:

```
$ rails console --sandbox
```

```
>> usuario1 = Usuario.new(nombre: “Juan”, email:
“juan@miempresa.com”)

>> usuario2 = Usuario.create(nombre: “Pepe”, email:
“juan@miempresa.com”)
```

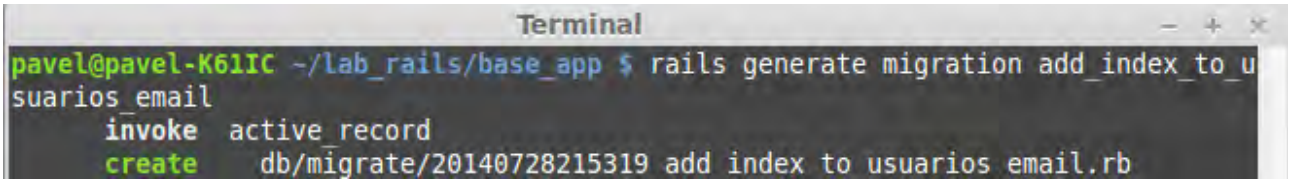
```
1.9.3-p545 :001 > usuario1 = Usuario.create(nombre: "Juan", email: "juan@miempre
sa.com")
   (0.1ms)  SAVEPOINT active_record_1
  Usuario Exists (0.2ms)  SELECT 1 AS one FROM "usuarios" WHERE LOWER("usuarios"
."email") = LOWER('juan@miempresa.com') LIMIT 1
  SQL (34.2ms)  INSERT INTO "usuarios" ("created_at", "email", "nombre", "update
d_at") VALUES (?, ?, ?, ?)  [["created_at", Mon, 28 Jul 2014 21:48:49 UTC +00:00
], ["email", "juan@miempresa.com"], ["nombre", "Juan"], ["updated_at", Mon, 28 J
ul 2014 21:48:49 UTC +00:00]]
   (0.1ms)  RELEASE SAVEPOINT active_record_1
 => #<Usuario id: 1, nombre: "Juan", email: "juan@miempresa.com", created_at: "2
014-07-28 21:48:49", updated_at: "2014-07-28 21:48:49">
1.9.3-p545 :002 > usuario2 = Usuario.create(nombre: "Pepe", email: "juan@miempre
sa.com")
   (0.1ms)  SAVEPOINT active_record_1
  Usuario Exists (0.2ms)  SELECT 1 AS one FROM "usuarios" WHERE LOWER("usuarios"
."email") = LOWER('juan@miempresa.com') LIMIT 1
   (0.1ms)  ROLLBACK TO SAVEPOINT active_record_1
 => #<Usuario id: nil, nombre: "Pepe", email: "juan@miempresa.com", created_at:
nil, updated_at: nil>
1.9.3-p545 :003 > Usuario.all
  Usuario Load (0.3ms)  SELECT "usuarios".* FROM "usuarios"
 => #<ActiveRecord::Relation [#<Usuario id: 1, nombre: "Juan", email: "juan@miem
presa.com", created_at: "2014-07-28 21:48:49", updated_at: "2014-07-28 21:48:49"
>]>
1.9.3-p545 :004 > usuario1.valid?
  Usuario Exists (0.2ms)  SELECT 1 AS one FROM "usuarios" WHERE (LOWER("usuarios
"."email") = LOWER('juan@miempresa.com') AND "usuarios"."id" != 1) LIMIT 1
 => true
1.9.3-p545 :005 > usuario2.valid?
  Usuario Exists (0.2ms)  SELECT 1 AS one FROM "usuarios" WHERE LOWER("usuarios"
."email") = LOWER('juan@miempresa.com') LIMIT 1
 => false
1.9.3-p545 :006 > usuario2.errors.full_messages
 => ["Email has already been taken"]
```

Llegados a este punto, aún no hemos terminado porque no se tiene la unicidad al cien, ¿por qué?, es muy sencillo, si un usuario *Test* (por ponerle nombre) se está registrando y a la hora de confirmar, accidentalmente ejerce dos clic en la confirmación, se tendrán dos peticiones iguales pero que *Rails* validará como correctas, salvando las dos peticiones en la base de datos y con el mismo correo.

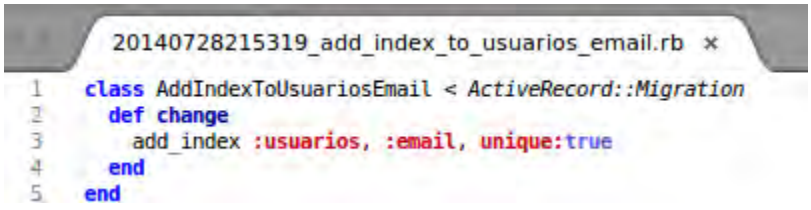
Para poder garantizar que lo anterior no pase, debemos crear una solución a nivel base de datos y esto lo logramos creando un índice (lo dejaremos como *index* dentro del código para evitar confusiones) para la columna *email*, y que este índice sea único.

La inclusión del *indice* a la columna *email*, requiere actualizar los requisitos de modelo, el cuál es manejado por *Rails* a través de las migraciones. En este caso, vamos a añadir una nueva estructura al modelo, así que debemos de realizar una migración directa usando el generador `migration`:

```
$ rails generate migration add_index_to_usuarios_email
```



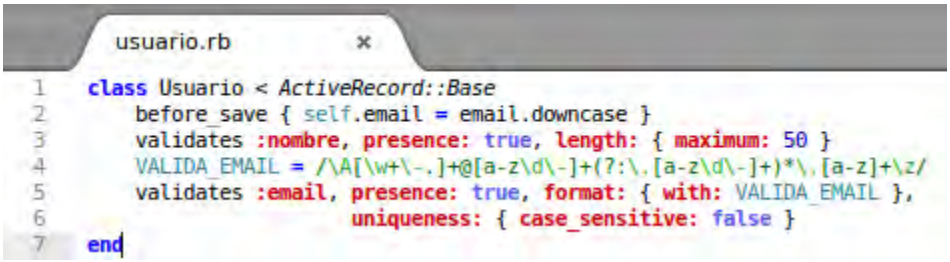
Como la migración de la unicidad de *email* no está predefinida (a diferencia de la de *usuarios*), se necesita llenar con la siguiente información el archivo **[timestamp]_add_index_to_usuarios_email** como se muestra:



Con el método `add_index` se añade un índice a la columna *email*, de la tabla *Usuarios* este índice por sí sólo no tiene unicidad, pero usando la opción `unique: true`, lo hace. Por último, migramos la base de datos:

```
$ bundle exec rake db:migrate
```

Nos falta sólo un paso para asegurar la unicidad, debemos de tener en la base de datos el *email* en minúsculas, esto se debe a que no todos los adaptadores de bases de datos usan índices sensibles a mayúsculas o minúsculas. Para remediar esto, usamos un *callback* que es un método que se invoca en algún punto en particular dentro de la vida de un objeto de *Active Record.* El *callback* que usaremos sera `before_save` para que antes de guardar al usuario en la base, *Rails* pase el correo electrónico a minúsculas, escribimos:



El *callback* `before_save` recibe un bloque como argumento que dentro hace referencia al objeto que se está usando (a través de `self`), en este caso, el correo electrónico que será cambiado a minúsculas usando el método `downcase`.

Con todo lo anterior, ya podemos garantizar la unicidad de nuestros registros en la base de datos.

7.3. Cómo añadir una contraseña segura

En este apartado vamos a añadir otro atributo básico para el usuario, la contraseña (segura). Esto consiste en que el usuario genere su contraseña y que ésta a través de una función *hash* se encripte y después guardar está versión encriptada en la base de datos. También vamos a añadir una forma de autenticar usuarios dada la contraseña.

Para autenticar a nuestros usuarios, tomaremos la contraseña que proporcionen para el inicio de sesión, se aplicará la función *hash* y se comparara con el valor almacenado en la base de datos (que también ya tiene aplicada la función *hash*). Si ambas coinciden, entonces la contraseña que dió el usuario es correcta y el usuario será autenticado. Nos conviene tener las contraseñas con *hash* ya que así, si la base de datos se ve comprometida con algún ataque o virus, sus contraseñas estarán seguras. Mucho de lo que implica la creación de una contraseña segura se relaciona con el uso del método `has_secure_password`, que veremos más adelante.

7.3.1. Aplicando la función hash a una contraseña

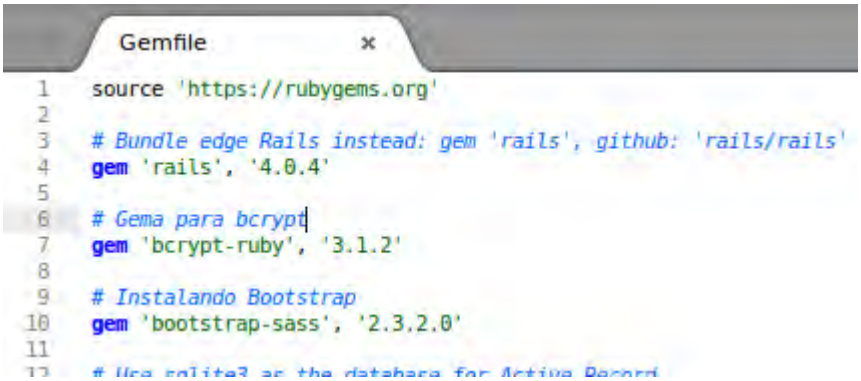
Para empezar, debemos de añadir un cambio al modelo. Vamos a añadir una columna correspondiente para guardar las contraseñas (*password_digest*[55]). Como comentamos anteriormente, aplicar una función *hash* a la contraseña nos asegura que el atacante no tenga posibilidad de autenticarse en el sitio, incluso si obtiene una copia de la base de datos.

[55] Se va a manejar el atributo como "password" debido a que "contraseña" lleva el carácter "ñ" y podría causar problemas en un futuro y `digest` por la función *hash* que se le aplica por eso y que más adelante con la seguridad de las contraseñas, se ocupará ese nombre.

Tabla 7.2. Modelo *Usuarios* actualizado.

Usuarios	
id	integer
nombre	string
email	string
password_digest	string
created_at	datetime
updated_at	datetime

Vamos a usar la función *hash* llamada *bcrypt*[56] para encriptar la contraseña y tenerla con un hash. Para usar *bcrypt* con la aplicación, vamos a añadir la *gema bcrypt-ruby* al archivo *Gemfile*:



Instalamos la *gema*:

```
$ bundle install
```

Como vamos a añadir una nueva columna al modelo, escribimos:

```
$ rails generate migration add_password_digest_to_usuarios
password_digest:string
```

[56] *Bcrypt* este algoritmo resulta ser el más fiable ya que métodos como MD5 o SHA-1 se han vuelto inseguros. *Bcrypt* usa un nuevo proceso matemático que resulta ser 10 más fuerte y que además introduce un factor de trabajo que nos permite determinar que tanto costará el hecho de generar un *hash*

```
pavel@pavel-K61IC ~/lab_rails/base_app $ rails generate migration add_password_d
igest_to_usuarios password_digest:string
      invoke  active_record
      create    db/migrate/20140728223036_add_password_digest_to_usuarios.rb
```

El primer argumento es el nombre de la migración y complementamos con el nombre y el tipo del nuevo atributo (columna en la base) que vamos a añadir. Podemos elegir cualquier nombre de migración, sólo hay que estar seguros de que añadimos al final del nombre `_to_usuarios`, para que *Rails* construya automáticamente la migración que va a añadir la nueva columna a la tabla *usuarios*.

Rails construye la siguiente migración:

20140728223036_add_password_digest_to_usuarios.rb

```
1  class AddPasswordDigestToUsuarios < ActiveRecord::Migration
2    def change
3      add_column :usuarios, :password_digest, :string
4    end
5  end
```

El código anterior usa el método `add_column` para añadir la columna *password_digest* a la tabla *usuario*. Por último, migramos la base:

```
$ bundle exec rake db:migrate
```

7.3.2. Confirmar contraseña

Esperando que los usuarios confirmen sus contraseñas, podemos realizarlo a través de un controlador, pero resulta más conveniente hacerlo en el modelo y usar la clase *Active Record.*

La lógica es añadir los atributos *password* y *password_confirmation*[57] a nuestro modelo *Usuario,* luego tenemos que verificar que los dos atributos coincidan antes de guardar un registro en la base de datos. A diferencia de otros atributos , los atributos de contraseña y su confirmación son temporalmente virtuales, sólo van a existir en memoria y no serán persistentes en la base. Estos atributos virtuales se implementan de forma automática a través del método `has_secure_password` como sigue:

[57] Estos nombres son generados por *Rails* a través del método `has_secure_password`, ahorrándonos tiempo y código.

usuario.rb

```ruby
class Usuario < ActiveRecord::Base
    before_save { self.email = email.downcase }
    validates :nombre, presence: true, length: { maximum: 50 }
    VALIDA_EMAIL = /\A[\w+\-.]+@[a-z\d\-]+(?:\.[a-z\d\-]+)*\.[a-z]+\z/
    validates :email, presence: true, format: { with: VALIDA_EMAIL },
                      uniqueness: { case_sensitive: false }
    has_secure_password
end
```

Podemos probarlo de la siguiente manera:

```
$ rails console --sandbox

>> usuario_contra = Usuario.new(nombre: “Test”, email:
“test@test.com”, password:”testing”, password_confirmation:
“notesting”)
```

```
1.9.3-p545 :002 > usuario_contra = Usuario.new(nombre: "Test", email: "test@test
.com", password: "testing", password_confirmation: "notesting")
 => #<Usuario id: nil, nombre: "Test", email: "test@test.com", created_at: nil,
updated_at: nil, password_digest: "$2a$10$BxG9yM46Et1rFbDcugqwbOR27x5x0iixRNUikp
5jrAXP...">
1.9.3-p545 :003 > usuario_contra.valid?
  Usuario Exists (0.2ms)  SELECT 1 AS one FROM "usuarios" WHERE LOWER("usuarios"
."email") = LOWER('test@test.com') LIMIT 1
 => false
1.9.3-p545 :004 > usuario_contra.errors.full_messages
 => ["Password confirmation doesn't match Password"]
```

```
>> usuario_contra = Usuario.new(nombre: “Test”, email:
“test@test.com”, password:”testting”, password_confirmation:
“testing”)
```

```
1.9.3-p545 :006 > usuario_contra = Usuario.new(nombre: "Test", email: "test@test
.com", password: "testing", password_confirmation: "testing")
 => #<Usuario id: nil, nombre: "Test", email: "test@test.com", created_at: nil,
updated_at: nil, password_digest: "$2a$10$gWY9cr7F4JUiM0pMVJ8R0OLOlwl3PU0pjNn5EK
3CzAXq...">
1.9.3-p545 :007 > usuario_contra.valid?
  Usuario Exists (0.2ms)  SELECT 1 AS one FROM "usuarios" WHERE LOWER("usuarios"
."email") = LOWER('test@test.com') LIMIT 1
 => true
```

7.3.3. Autenticar un usuario

Lo último que nos falta en este mecanismo de contraseñas es un método que nos regrese a los usuarios basados en su email y contraseña. Primero, debemos encontrar al usuario por su email; segundo, autenticar al usuario con una contraseña dada.

Como hemos visto previamente, buscaremos al usuario con ayuda de `find_by`:

```
usuario = Usuario.find_by(email: email)
```

Ahora debemos de usar el método `authenticate` para verificar que el usuario tiene la contraseña dada:

```
usuario_actual = usuario_contra.authenticate(password)
```

Si la contraseña coincide con la contraseña de usuario, deberá de regresar al usuario, en otro caso, regresa `false`.

```
1.9.3-p545 :010 > usuario_actual = usuario_contra.authenticate("testing")
 => #<Usuario id: nil, nombre: "Test", email: "test@test.com", created_at: nil,
updated_at: nil, password_digest: "$2a$10$gWY9cr7F4JUiM0pMVJ8R0OLOlwl3PU0pjNn5EK
3CzAXq...">
1.9.3-p545 :011 > usuario_actual = usuario_contra.authenticate("notesting")
 => false
```

7.3.4. Contraseñas seguras

Para crear una contraseña segura, tenemos que tomar en cuenta algunos requisitos mínimos como por ejemplo, la longitud de la contraseña y la presencia de ésta. Para validar la longitud de la contraseña, usaremos la llave `:minimum` junto con `:maximum` para hacerlo. La presencia se valida a través del método `has_secure_password`[58].

```
validates :password, length: {minimum: 8}
```

Ahora, vamos a añadir los atributos *password* y *password_confirmation*, también requerimos la presencia de la contraseña y que coincidan; y añadir un método que compare la contraseña con *hash* con la contraseña de *password_digest* para autenticar a los usuarios y todo esto ya se incluye en un sólo método: `has_secure_password`. Tanto como exista la columna *password_digest* en la base de datos, añadir el método `has_secure_password` al modelo, nos brinda un forma segura de crear y autenticar a nuevos usuarios.

[58] Para saber más sobre este método y sobretodo leer su código, consultamos el siguiente enlace: https://github.com/rails/rails/blob/master/activemodel/lib/active_model/secure_password.rb

Juntando lo anterior nos queda nuestro archivo **usuario.rb** como sigue:

usuario.rb

```
1  class Usuario < ActiveRecord::Base
2      before_save { self.email = email.downcase }
3      validates :nombre, presence: true, length: { maximum: 50 }
4      VALIDA_EMAIL = /\A[\w+\-.]+@[a-z\d\-]+(?:\.[a-z\d\-]+)*\.[a-z]+\z/
5      validates :email, presence: true, format: { with: VALIDA_EMAIL },
6                        uniqueness: { case_sensitive: false }
7      has_secure_password
8      validates :password, length: { minimum: 8 }
9  end
```

7.3.5. Creando usuarios

Hemos completado el modelo *Usuario*, crearemos un usuario en la base de datos para empezar a diseñar la página que mostrará su respectiva información (y la de los demás usuarios). Todavía no podemos crear usuarios a través de la interfaz web, así que vamos a hacerlo manualmente con la consola de *Rails*. No usaremos el modo *sandbox* de la consola ya que queremos guardar un registro en la base de datos.

```
$ rails console

>> Usuario.create(nombre: “Eliz Heralf”, email:
“elizheralf@company.com”, password: “elizrules”,
password_confirmation: “elizrules”)

>> Usuario.find_by(email: “elizheralf@company.com”)
```

```
pavel@pavel-K61IC ~/lab_rails/base_app $ rails console
Loading development environment (Rails 4.0.4)
1.9.3-p545 :001 > Usuario.create(nombre: "Eliz Heralf", email: "elizheralf@compa
ny.com", password: "elizrules", password_confirmation: "elizrules")
   (0.1ms)  begin transaction
  Usuario Exists (0.3ms)  SELECT 1 AS one FROM "usuarios" WHERE LOWER("usuarios"
."email") = LOWER('elizheralf@company.com') LIMIT 1
Binary data inserted for `string` type on column `password_digest`
  SQL (5.1ms)  INSERT INTO "usuarios" ("created_at", "email", "nombre", "passwor
d_digest", "updated_at") VALUES (?, ?, ?, ?, ?)  [["created_at", Mon, 28 Jul 201
4 22:48:25 UTC +00:00], ["email", "elizheralf@company.com"], ["nombre", "Eliz He
ralf"], ["password_digest", "$2a$10$WsfxpC66vj3AVCVV27zPu.jRMacDayJS9YAfC.4GLjdF
j8iJTNoeK"], ["updated_at", Mon, 28 Jul 2014 22:48:25 UTC +00:00]]
   (126.9ms)  commit transaction
 => #<Usuario id: 1, nombre: "Eliz Heralf", email: "elizheralf@company.com", cre
ated_at: "2014-07-28 22:48:25", updated_at: "2014-07-28 22:48:25", password_dige
st: "$2a$10$WsfxpC66vj3AVCVV27zPu.jRMacDayJS9YAfC.4GLjdF...">
1.9.3-p545 :002 > Usuario.find_by(email: "elizheralf@company.com")
  Usuario Load (0.4ms)  SELECT "usuarios".* FROM "usuarios" WHERE "usuarios"."em
ail" = 'elizheralf@company.com' LIMIT 1
 => #<Usuario id: 1, nombre: "Eliz Heralf", email: "elizheralf@company.com", cre
ated_at: "2014-07-28 22:48:25", updated_at: "2014-07-28 22:48:25", password_dige
st: "$2a$10$WsfxpC66vj3AVCVV27zPu.jRMacDayJS9YAfC.4GLjdF...">
1.9.3-p545 :003 >
```

Podemos ver lo que hace `has_secure_password` echando un vistazo al atributo *password_digest*. Veremos la versión *hash* de la contraseña ("elizrules") que asignamos para inicializar el objeto usuario.

```
>> usuario = Usuario.find_by(email: "elizheralf@company.com")

>> usuario.password_digest
```

```
1.9.3-p545 :006 >   usuario = Usuario.find_by(email: "elizheralf@company.com")
  Usuario Load (0.4ms)  SELECT "usuarios".* FROM "usuarios" WHERE "usuarios"."em
ail" = 'elizheralf@company.com' LIMIT 1
 => #<Usuario id: 1, nombre: "Eliz Heralf", email: "elizheralf@company.com", cre
ated_at: "2014-07-28 22:48:25", updated_at: "2014-07-28 22:48:25", password_dige
st: "$2a$10$WsfxpC66vj3AVCVV27zPu.jRMacDayJS9YAfC.4GLjdF...">
1.9.3-p545 :007 > usuario.password_digest
 => "$2a$10$WsfxpC66vj3AVCVV27zPu.jRMacDayJS9YAfC.4GLjdFj8iJTNoeK"
```

Podemos verificar también que el método `authenticate` esté funcionando correctamente si nos tratamos de iniciar sesión usando una contraseña incorrecta y después la correcta:

```
>> usuario.authenticate("estanoeslacontra")
```

```
>> usuario.authenticate(“elizrules”)
```

```
1.9.3-p545 :009 > usuario.authenticate("estanoeslacontra")
 => false
1.9.3-p545 :010 > usuario.authenticate("elizrules")
 => #<Usuario id: 1, nombre: "Eliz Heralf", email: "elizheralf@company.com", cre
ated_at: "2014-07-28 22:48:25", updated_at: "2014-07-28 22:48:25", password_dige
st: "$2a$10$WsfxpC66vj3AVCVV27zPu.jRMacDayJS9YAfC.4GLjdF...">
```

7.5. Guardando cambios

Hemos acabado con el modelo de Usuarios que es ahora funcional con la creación, manipulación y autenticación de usuarios; así como la creación, confirmación y seguridad de las contraseñas de los usuarios. En el siguiente capítulo se podrán registrar usuarios a través de un formulario y mostrar su información en una página.

Salvemos los cambios creados, hagamos un *merge* y subamos nuestro trabajo al repositorio remoto.

```
$ git add .

$ git commit -m ‘Modelo de usuarios completo’

$ git checkout master

$ git merge modelo_usuarios

$ git push origin master
```

Capítulo 8: Registro de Usuarios

8. Desarrollo

Anteriormente habíamos concluido el modelo *Usuario*, es tiempo de que los usuarios se registren en el sitio. Para el registro de los usuarios, vamos a implementar un formulario *HTML* que enviará la información del registro a nuestra aplicación, con esta información crearemos un nuevo usuario y salvamos sus atributos en la base de datos. Al final de este proceso, es importante hacer una página de perfil, con la información del usuario nuevo, es por eso que se hará una página para mostrar a los usuarios.

Como ya contamos con un usuario en la base (capítulo 7), empezaremos por crear la página de perfil de usuario. Creamos la rama correspondiente.

```
$ git checkout master

$ git checkout -b registro_usuarios
```

8.1. Mostrando a los usuarios

En este apartado empezaremos el desarrollo en torno a la página de perfil con la creación de una página que muestre el nombre del usuario y una foto del mismo. El objetivo es que en la página de perfil tengamos, la imagen del usuario, su información básica y sus mensajes. Todo lo anterior se muestra como maqueta en las Figuras 8.1 y la Figura 8.2.

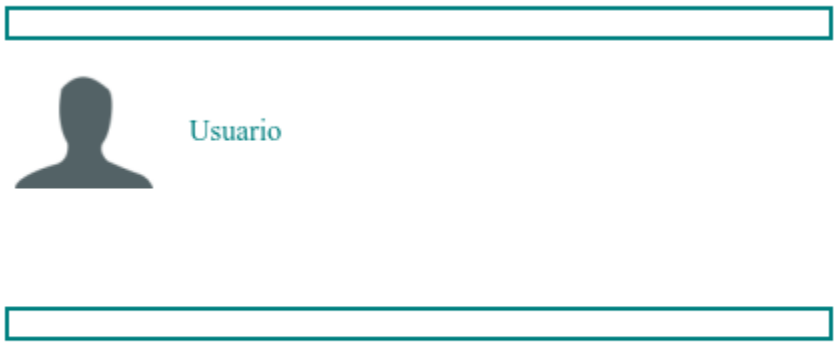


Figura 8.1. Maqueta de la página de perfil.

Figura 8.2. Maqueta de página de perfil detallado.

8.1.1. Modo debug y ambientes en Rails

Los perfiles que se harán en esta sección, serán las primeras páginas dinámicas que manejaremos. La vista va a existir como una página de código simple, cada perfil será personalizado usando la información que tomemos de la base de datos.

Es importante contar con la información del modo *debug* en el *layout* de nuestro sitio ya que esto mostrará en pantalla información importante sobre cada página que esté utilizando el método `debug` y la variable `params` (que veremos más a detalle en el desarrollo de este capítulo).

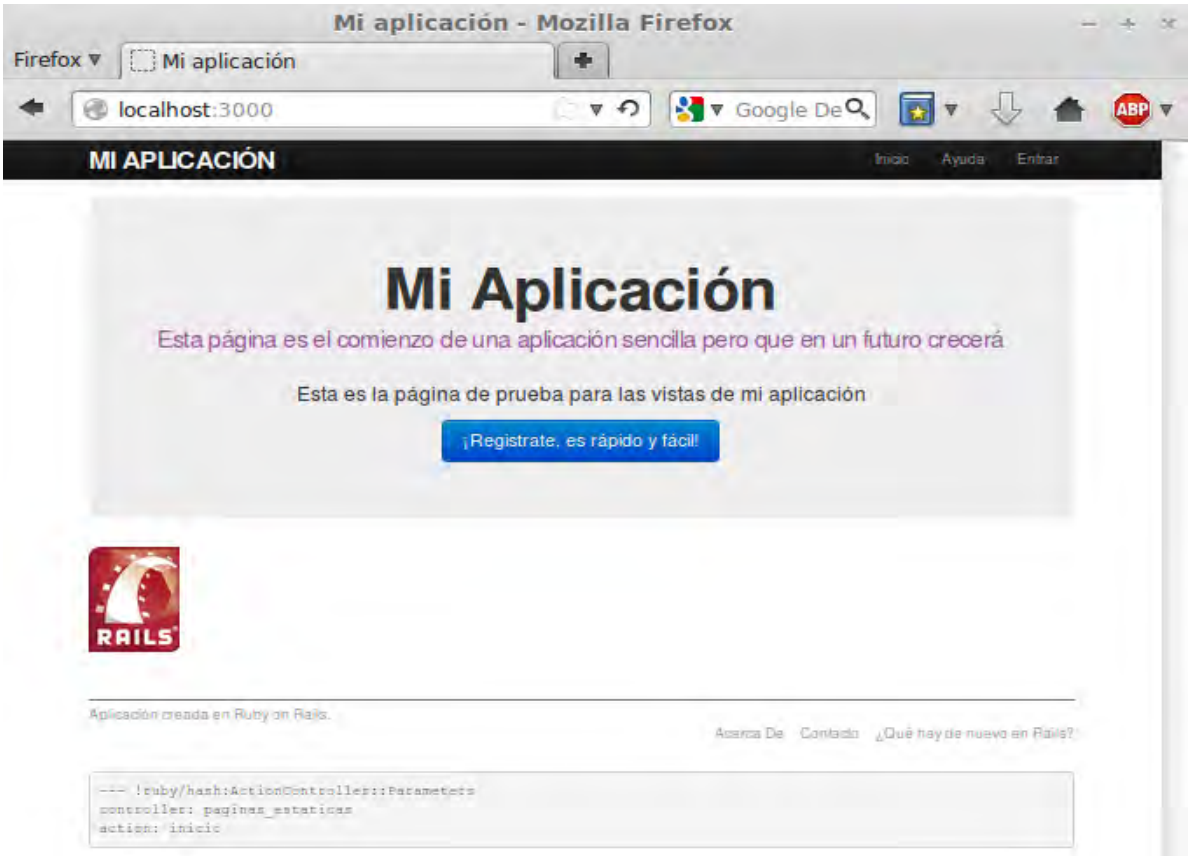
application.html.erb

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title><%= titulo_base(yield(:titulo)) %></title>
5    <%= stylesheet_link_tag    "application", media: "all", "data-turbolinks-track" => true
       %>
6    <%= javascript_include_tag "application", "data-turbolinks-track" => true %>
7    <%= csrf_meta_tags %>
8  </head>
9  <body>
10     <%= render 'layouts/etiquetaHeader'%>
11     <div class="container">
12       <%= yield %>
13       <%= render 'layouts/etiquetaFooter' %>
14       <%= debug(params) if Rails.env.development? %>
15     </div>
16   </body>
17 </html>
```

Figura 8.3. Muestra del modo *debug*.

Cambiamos nuestra hoja de estilo **estilo.css.scss** para mostrar de mejor manera la salida del modo *debug*.

```
107  /*Otros estilos*/
108
109  @mixin box_sizing {
110    -moz-box-sizing: border-box;
111    -webkit-box-sizing: border-box;
112    box-sizing: border-box;
113  }
114
115  .debug_dump {
116    clear: both;
117    float: left;
118    width: 100%;
119    margin-top: 20px;
120    @include box_sizing;
121  }
```



8.4. Modo *debug* con estilo.

Se hizo uso de las facilidades de *Sass* con los *mixin*, que son un conjunto de reglas *CSS* que agrupan y se pueden usar en múltiples elementos. Queda más claro con un ejemplo:

Si tenemos:

```
.debug_dump {
  clear: both;
  float: left;
  width: 100%;
  margin-top: 20px;
  @include box_sizing;
}
```

Se convierte en esto:

```
.debug_dump {
  clear: both;
  float: left;
  width: 100%;
  margin-top: 20px;
  -moz-box-sizing: border-box;
  -webkit-box-sizing: border-box;
  box-sizing: border-box;
}
```

La salida de la Figura 8.4 nos brinda información importante sobre la página que está siendo desplegada en el navegador.

La variable `params` es básicamente un *hash* y se representa como un *YAML*[59], en este caso nos dice quién es el controlador y la acción correspondiente a la página. Ahora, esta información sólo debe de estar disponible al desarrollador y no a los usuarios finales, es por eso que se restringe el acceso con la línea:

```
if Rails.env.developmet?
```

Lo anterior, restringe la información de *debug* para el ambiente de desarrollo (*development environment*), que es uno de los tres ambientes que se definen por defecto en *Rails* (que vimos también en la configuración de la base de datos en el capítulo 2). Por ejemplo, todo el código embebido de *Ruby* en el ambiente de desarrollo, no será insertado en el ambiente de producción (*production development*) o en el ambiente de pruebas (*test development*) y viceversa. El método `Rails.env.development?` regresa true sólo en un ambiente de desarrollo

8.1.2. Usuarios como recurso

Hemos estado siguiendo las convenciones de la arquitectura *REST* en nuestra aplicación y esto significa representar datos como recursos que pueden ser creados, mostrados, actualizados o destruidos. Los recursos son típicamente referenciados usando el nombre del recurso en sí, y un identificador único. Aplicado a los usuarios (*Usuarios resource*, como recurso) debemos poder visualizar la información del usuario con *id* x, si hacemos una petición *GET* a través de la *URL* /usuarios/x.

Anteriormente creamos un usuario en la base de datos, si deseamos acceder a su información con la *URL* /usuarios/1, deberíamos poder visualizar la información pero nos dará un error (Figura 8.5.).

[59] *YAML*(*Ain't Another Markup Language*, en inglés) es un lenguaje de marcado ligero, inspirado en *XML*. Más información en su página oficial: http://www.yaml.org/

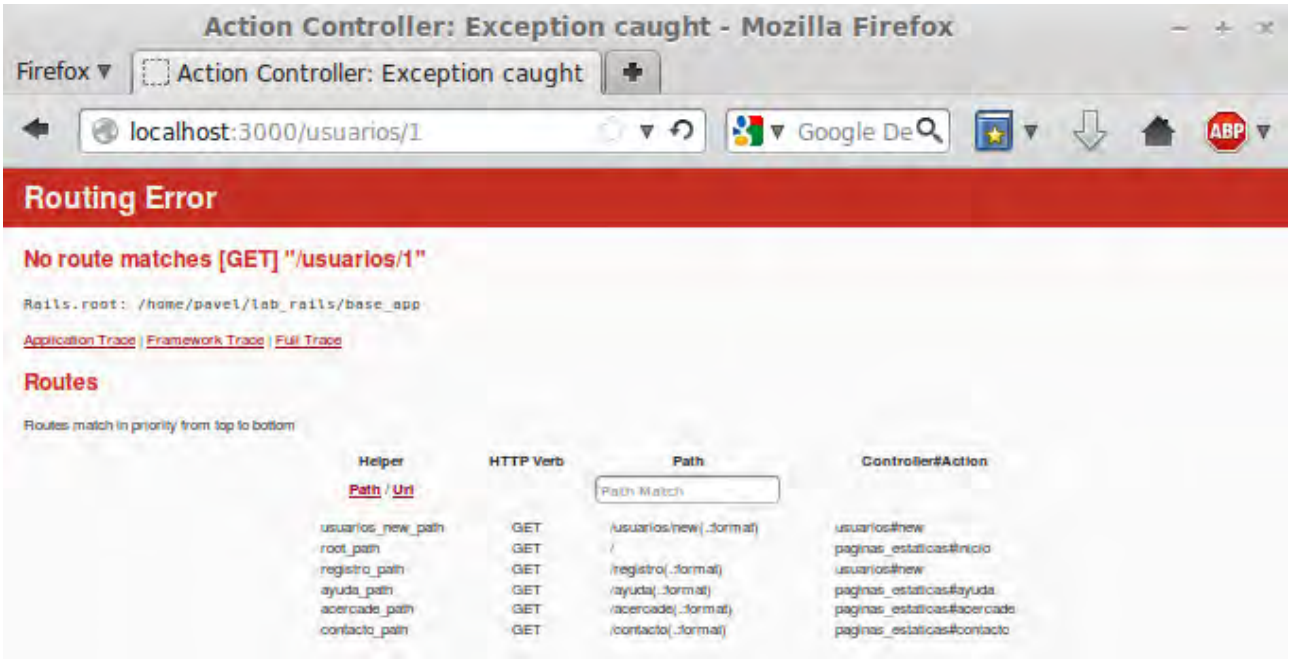


Figura 8.5. Error de ruta.

Para tener el estilo de *URL* que usa *REST* añadimos la siguiente línea al archivo **routes.rb**:

```
resources :usuarios
```

routes.rb

```
1  BaseApp::Application.routes.draw do
2    resources :usuarios
3    root 'paginas_estaticas#inicio'
4    match '/registro', to: 'usuarios#new', via: 'get'
5    match '/ayuda', to: 'paginas_estaticas#ayuda', via: 'get'
6    match '/acercade', to: 'paginas_estaticas#acercade', via: 'get'
7    match '/contacto', to: 'paginas_estaticas#contacto', via: 'get'
```

Hacemos notar que la línea `get “usuarios/new”` se ha eliminado. Esto es así porque al agregar `resources :usuarios`, estamos no sólo añadiendo una ruta funcional /usuarios/x; estamos añadiendo también a nuestra aplicación todas las acciones *REST* requeridas para el recurso *Usuarios*, además de un montón de rutas personalizadas para la generación de las *URL* de usuario, podemos ver las *URLs*, acciones y las rutas personalizadas en la tabla 8,1.

Tabla 8.1. Rutas *REST* para el recurso *Usuarios*.

URL	Petición HTTP	Acción Correspondiente	Ruta Personalizada	Función
/usuarios	GET	index	usuarios_path	Lista a todos los usuarios
/usuarios/x	GET	show	usuario_path(usuario)	Muestra un usuario
/usuarios/new	GET	new	new_usuario_path	Página para haccer un nuevo usuario
/usuarios	POST	create	usuarios_path	Crea un nuevo usuario
/usuarios/x/edit	GET	edit	edit_usuario_path(usuario)	Página para editar al usuario con id *x*
/usuarios/x	PATCH	update	usuario_path(usuario)	Actualiza al usuario
/usuarios/x	DELETE	destroy	usuario_path(usuario)	Elimina al usuario

Con la adición del recurso, la ruta /usuarios/1 abrirá pero no tendrá una página correspondiente. Esto lo vamos a solucionar añadiendo la página de perfil.

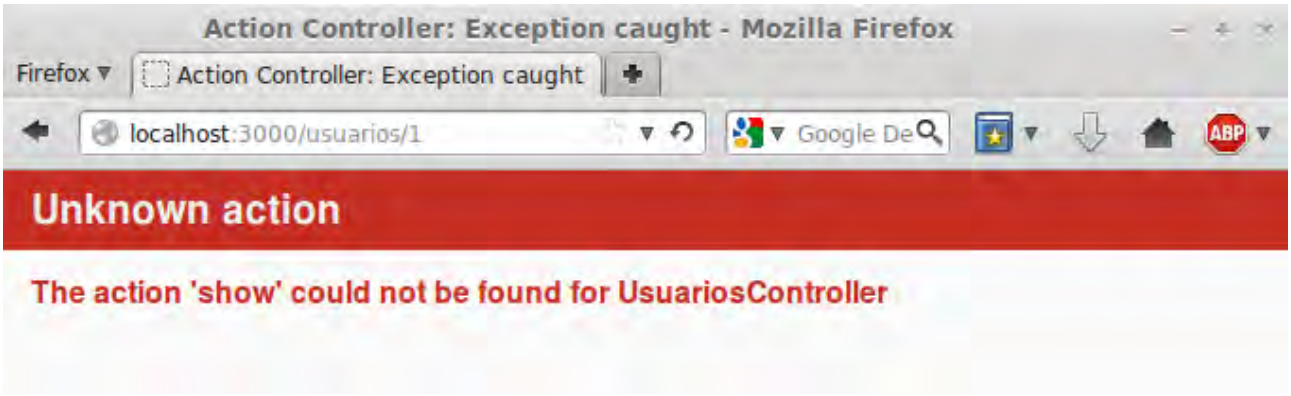


Figura 8.6. Vista no encontrada.

Usando la ubicación estándar para mostrar un usuario tendremos la ruta app/views/usuarios/show.html.erb. El archivo **show.html.erb** no existe, así que debemos crearlo, añadiendo lo siguiente:

show.html.erb

```
1  <%= @usuario.nombre %>, <%= @usuario.email %>
```

El código anterior muestra el nombre de usuario y su email, asumiendo que exista una variable de instancia `@usuario`. El email será omitido al usuario final, lo agregamos por el momento sólo para verificar que retorne la información.

Para que la viste funcione, debemos definir la variable de instancia `@usuario` dentro de la acción `show` del controlador *Usuarios*.

usuarios_controller.rb

```
1  class UsuariosController < ApplicationController
2
3    def show
4      @usuario = Usuario.find(params[:id])
5    end
6
7    def new
8    end
9  end
```

Con `params` recuperamos el *id* del usuario, `params[:id]` será el usuario con el *id* 1, que es equivalente a usar `Usuario.find(1)`. *Rails* se encarga de convertir la cadena `"1"` en un entero para que lo anterior funcione.

Ya definidos tanto la vista, como la acción, la *URL* /usuarios1/ debe de funcionar correctamente, mostrando también la acción, el controlador y el id correspondientes al usuario en el modo *debug* (Figura 8.7).

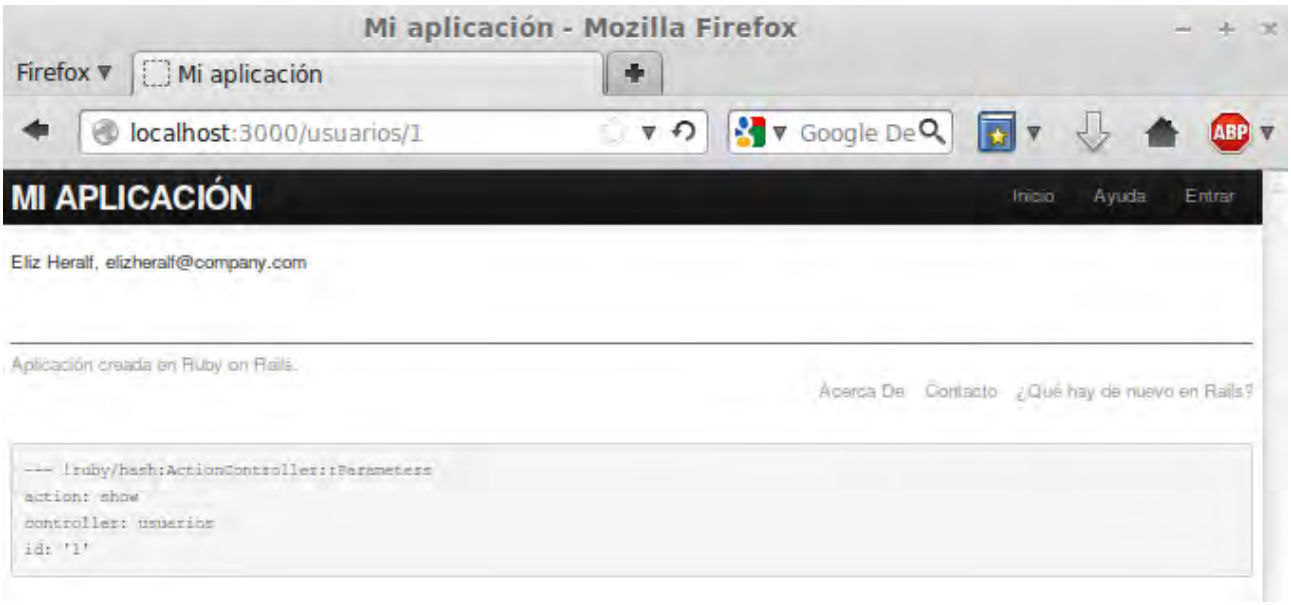
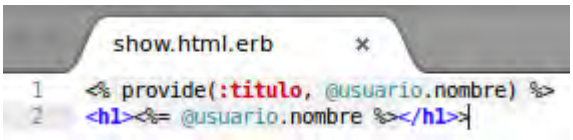


Figura 8.7. Vista de usuarios.

```
--- !ruby/hash:ActionController::Parameters
action: show
controller: usuarios
id: '1'
```

Sólo para finalizar, vamos a modificar el archivo **show.html.erb** para añadir el nombre del usuario como título de la página (Figura 8.8).



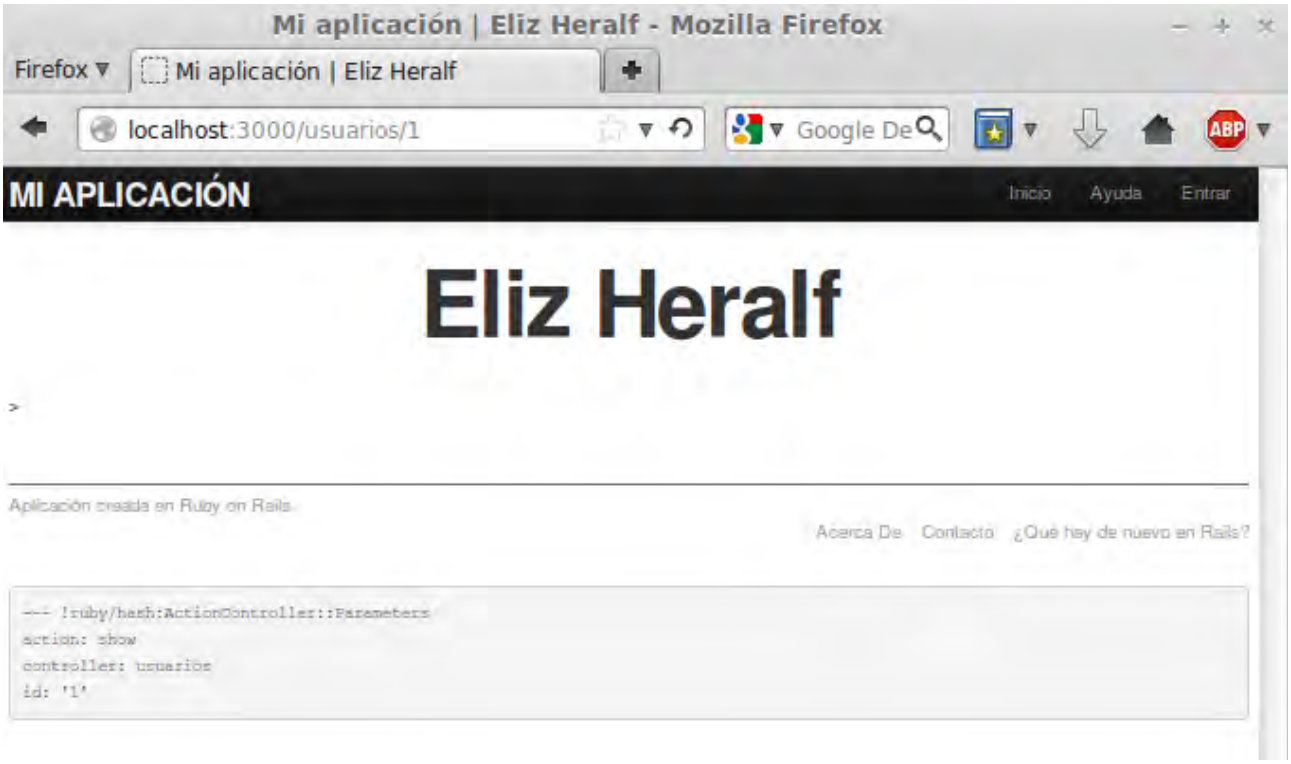


Figura 8.8. Muestra del nuevo título en la vista.

8.1.3. Imagen de perfil

Un elemento importante para la aplicación que vamos a desarrollar, es que nuestro usuario tenga una imagen personal. Vamos a hacer uso de una herramienta llamada *Gravatar*[60] (*Globally Recognized Avatar* en inglés y creado por *Tom Preston*), esta herramienta nos ahorrará los problemas de subir una imagen, cortarla y almacenarla; lo único que debemos hacer es construir una *URL* con la imagen en *Gravatar* usando nuestro email y obviamente, la imagen que queremos tener de perfil.

Preliminarmente tendríamos un *helper* con un método `gravatar_de` que retorne la imagen dado un usuario. El archivo **show.html.erb** tendría la siguiente estructura (no funcional por el momento):

show.html.erb

```
1  <% provide(:titulo, @usuario.nombre) %>
2  <h1>
3     <%= gravatar_de @usuario%>
4     <%= @usuario.nombre %>
5  </h1>
```

[60] *Gravatar* es un servicio que asocia una imagen a un correo electrónico, así, cuando te registras en un sitio que usa *Gravatar* ya no te preocupas de subir la imagen, de hecho, se asigna automáticamente ahorrando tiempo. Para más información: https://es.gravatar.com/

No hemos definido el método `gravatar_de` aún. Por defecto los métodos que definimos en cualquier *helper* se encuentran disponibles para cualesquiera de las vistas asociadas al controlador `Usuarios`. Consultando la página de *Gravatar*, nos enteramos de que sus *URLs* son basadas en una función *hash MD5* sobre el email del usuario. Podemos implementar el algorimo *MD5* en *Ruby* con el método `hexdigest` de la clase `Digest`[61] (esta clase ya viene en el paquete de instalación de Ruby):

```
>> email = “testing@portal.com”

>> Digest::MD5::hexdigest(email.downcase)
```

```
1.9.3-p545 :001 > email = "testing@portal.com"
 => "testing@portal.com"
1.9.3-p545 :002 > Digest::MD5::hexdigest(email.downcase)
 => "dc8e4fba7e5319af0fef07927515dc79"
```

Ahora, ya implementando el método `hexdigest` al archivo **usuarios_helper.rb**, nos queda de la siguiente manera:

usuarios_helper.rb

```
1  module UsuariosHelper
2
3      #Implementación de la imagen de perfil, usando Gravatar
4      def gravatar_de(usuario)
5          gravatar_id = Digest::MD5::hexdigest(usuario.email.downcase)
6          gravatar_url = "https://secure.gravatar.com/avatar/#{gravatar_id}"
7          image_tag(gravatar_url, alt: usuario.nombre, class: "gravatar_imagen")
8      end
9  end
```

El código anterior es la forma en la que retornamos la etiqueta de una imagen para *Gravatar*, junto con una clase “`gravatar`” y una un texto `alt` con el nombre de usuario. La *URL* que se especifica es parte del uso de *Gravatar*, para más información consultar la página oficial.

El resultado de modificar **usuarios_helper.rb** se muestra en la Figura 8.9:

[61] Para más información sobre *Digest*, consultar el *API* en: http://ruby-doc.org/stdlib-2.1.0/libdoc/digest/rdoc/Digest.html

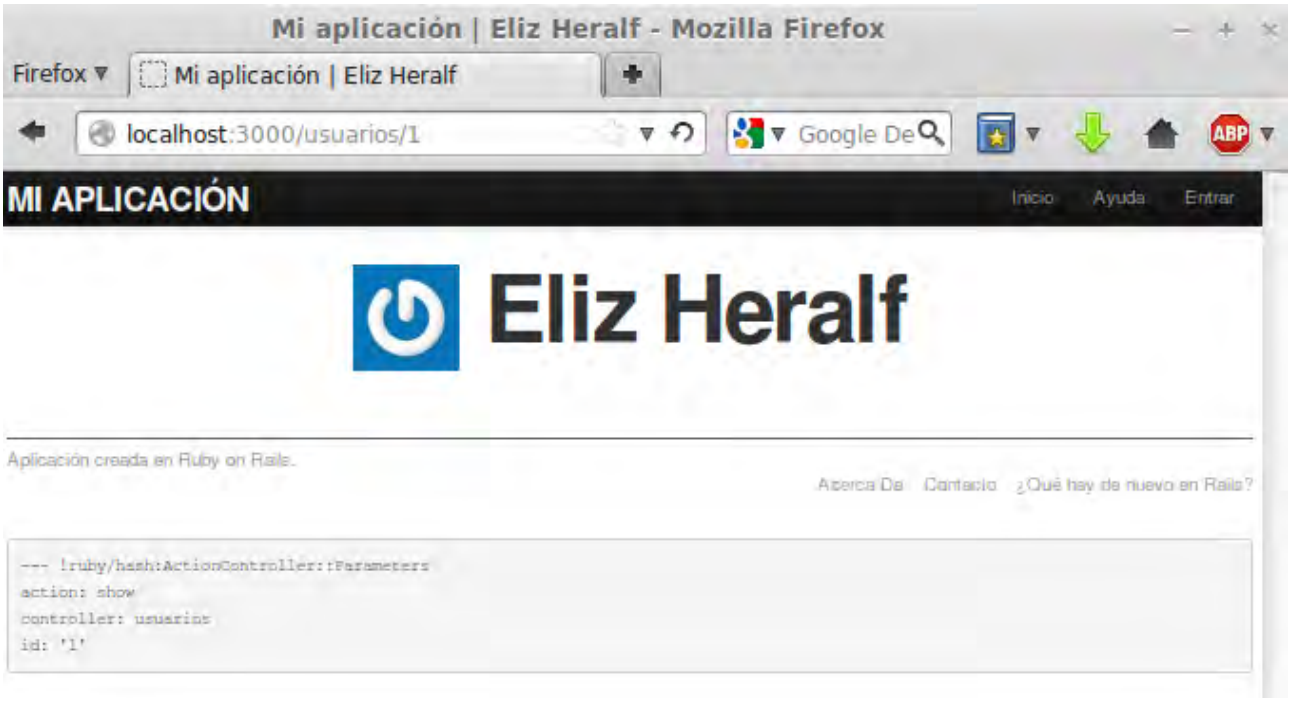


Figura 8.9. Prueba de *Gravatar*.

El avatar corresponde a la *URL* en el *helper*, esta imagen se muestra cuando no existe un correo registrado en la base de *Gravatar*. Modificaremos el correo de nuestro usuario por uno que si esté en la base de datos de *Gravatar*, o podemos crear un nuevo usuario en el que su correo este en la base. También podemos usar el correo "*laboratorio.rubyrails@gmail.com*" y sólo modificar el original de nuestro usuario.

En este caso, optamos por sólo modificar el correo del usuario de la base con el correo anterior, el resultado es el siguiente:

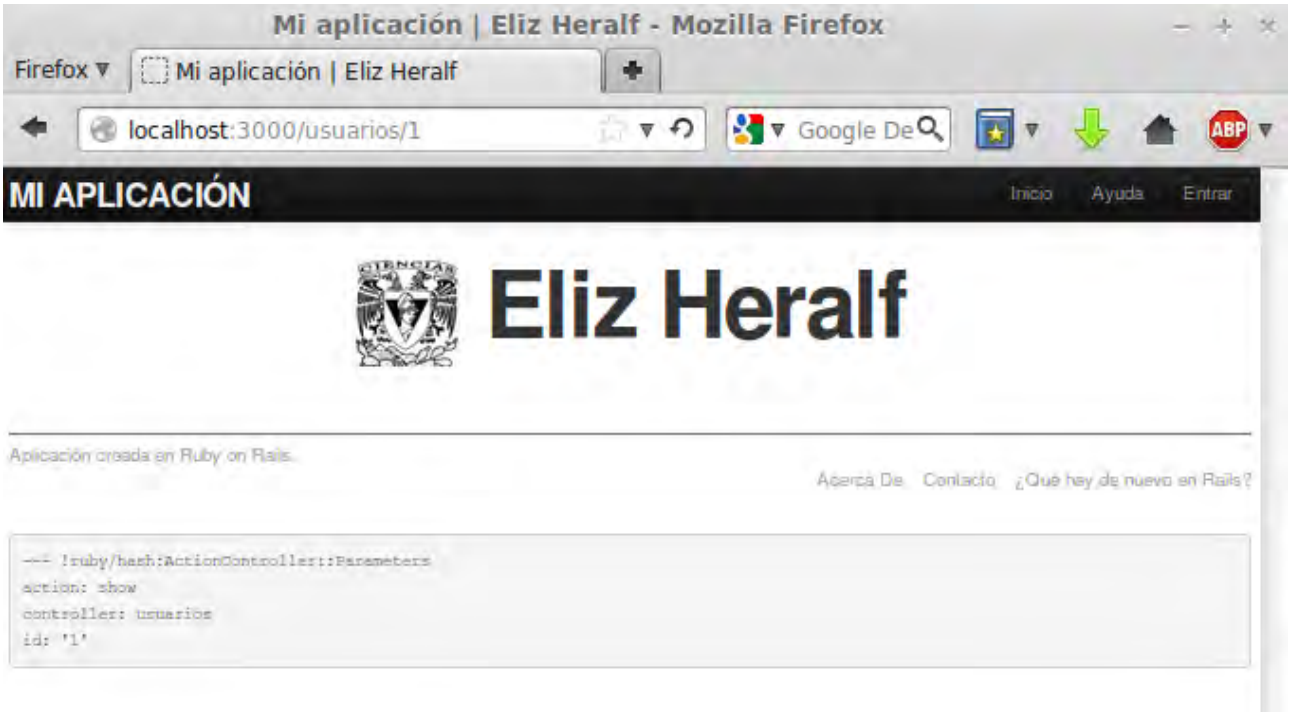


Figura 8.10. *Gravatar* funcionando.

Ahora, para dar un estilo más profesional y apegado a nuestra aplicación, vamos a insertar una barra lateral (*sidebar*). Usamos la etiqueta `aside` que es usada para el contenido que complementa nuestra página pero puede usarse de manera independiente. Incluimos también las clases `row` y `span4`.

show.html.erb

```
1  <% provide(:titulo, @usuario.nombre) %>
2
3  <div class = "row">
4      <aside class = "span4">
5          <section>
6              <h1>
7                  <%= gravatar_de @usuario %>
8                  <%= @usuario.nombre %>
9              </h1>
10         </section>
11     </aside>
12 </div>
```

Añadiendo estilo a la barra lateral y al elemento *Gravatar*, el archivo **estilo.css.scss** nos queda de la siguiente manera:

```
123  /*Barra lateral*/
124
125  aside {
126    section {
127      padding: 15px 0;
128      border-top: 1.9px solid $color_gris;
129      &:first-child {
130        border: 0;
131        padding-top: 0;
132      }
133      span {
134        display: block;
135        margin-bottom: 3px;
136        line-height: 1;
137      }
138      h1 {
139        font-size: 1.45em;
140        text-align: left;
141        letter-spacing: -1.5px;
142        margin-bottom: 3px;
143        margin-top: 0px;
144      }
145    }
146  }
147
148  .gravatar_imagen {
149    margin-right: 10px;
150    float: left;
151  }
```

Quedando el avatar más al estilo de un perfil de una red social como la maqueta inicial (Figura 8.1) y que se muestra en la Figura 8.11.

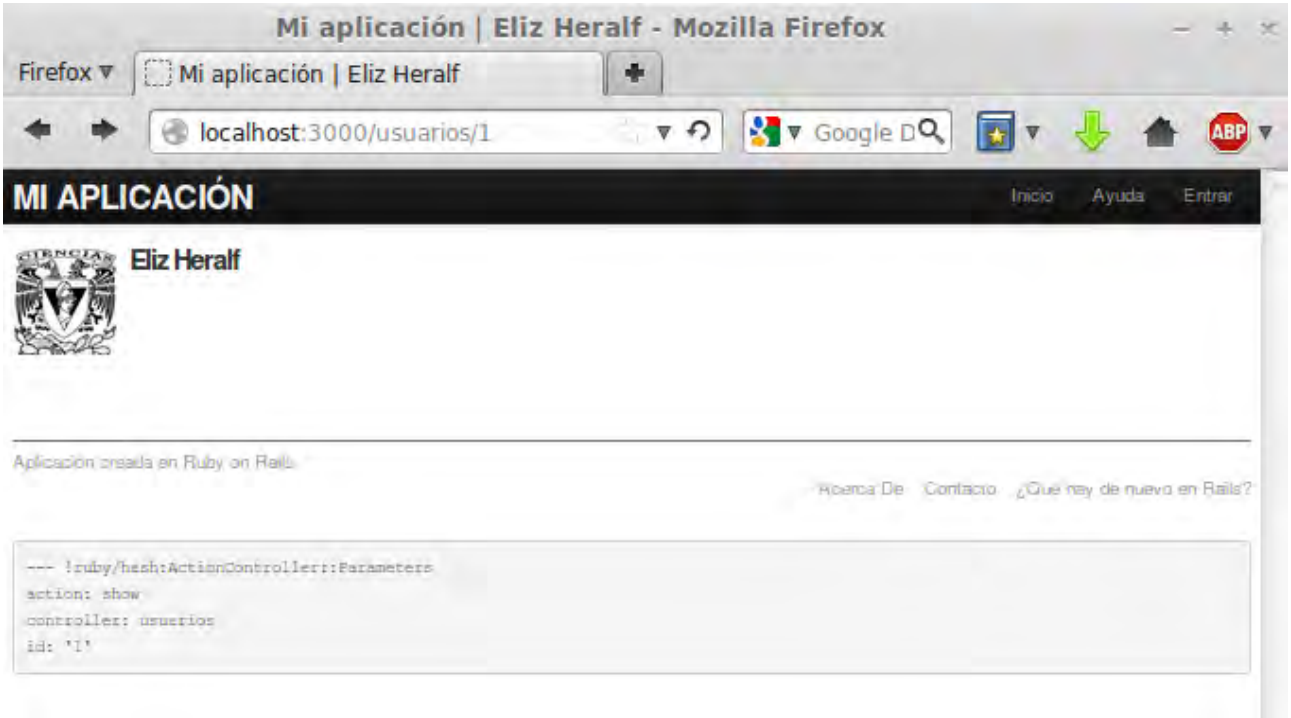


Figura 8.11. Avatar del perfil de usuario.

8.2. Formulario de registro para usuarios

Ahora que tenemos una interfaz funcional -más no completa- para el perfil de usuario, podemos empezar un formulario web, que permita el registro de usuarios vía web. Recordemos que nuestra vista para el registro está vacía, el objetivo de este apartado es cambiar dicha vista por una que contenga un formulario de registro como en la maqueta de la Figura 8.12.:

Figura. 8.12. Registro previo.

Figura 8.13. Maqueta Registro.

8.2.1. Uso de form_for

Para hacer el formulario correspondiente al registro, usaremos el método `form_for`, el cual toma lugar en el objeto de *Active Record* y automáticamente construye un formulario basado en los atributos de dicho objeto. Lo anterior se muestra en el archivo **new.html.erb**:

new.html.erb

```
<% provide(:title, 'Registro') %>
<h1>Registro</h1>

<div class = "row">
  <div class = "span6 offset3">
    <%= form_for(@usuario) do |atributo| %>

      <%= atributo.label :nombre %>
      <%= atributo.text_field :nombre %>

      <%= atributo.label :email %>
      <%= atributo.text_field :email %>

      <%= atributo.label :password %>
      <%= atributo.password_field :password %>

      <%= atributo.label :password_confirmation, "Confirmación" %>
      <%= atributo.password_field :password_confirmation %>

      <%= atributo.submit "¡Crear tu cuenta!", class: "btn btn-large btn-primary" %>
    <% end %>
  </div>
</div>
```

En el código anterior tenemos el ciclo `do` en el que nuestro método `form_for` toma un bloque con una variable llamada `atributo`. Cuando usamos *helpers* no es necesario saber muchos detalles sobre la implementación pero en este caso cuando llamamos a `atributo` con algún método que corresponda a un elemento de un formulario *HTML*, este retorna el código específico para asignar un atributo del objeto tipo `@usuario`, es decir, el código,

```
<%= atributo.label :nombre %>
<%= atributo.text_field :nombre %>
```

crea el código *HTML* necesario para generar el campo de texto con la etiqueta apropiada para asignar el atributo `nombre` del modelo *Usuario*.

Si abrimos el navegador con la *URL* de registro, nos despliega un error ya que no hemos definido a la variable `@usuario` (la variable es `nil`). Vamos a definir la variable en la acción en el controlador que corresponda en **new.html.erb**, osease, la acción `new` en el controlador `Usuario`. El método `form_for` espera un objeto de tipo `Usuario` y como queremos crear un nuevo usuario, simplemente complementamos con `User.new`, asignándolo a la variable `@usuario`, como se muestra:

usuarios_controller.rb

```
1  class UsuariosController < ApplicationController
2
3    def show
4      @usuario = Usuario.find(params[:id])
5    end
6
7    def new
8      @usuario = Usuario.new
9    end
10 end
```

El resultado de todo lo anterior es el siguiente:

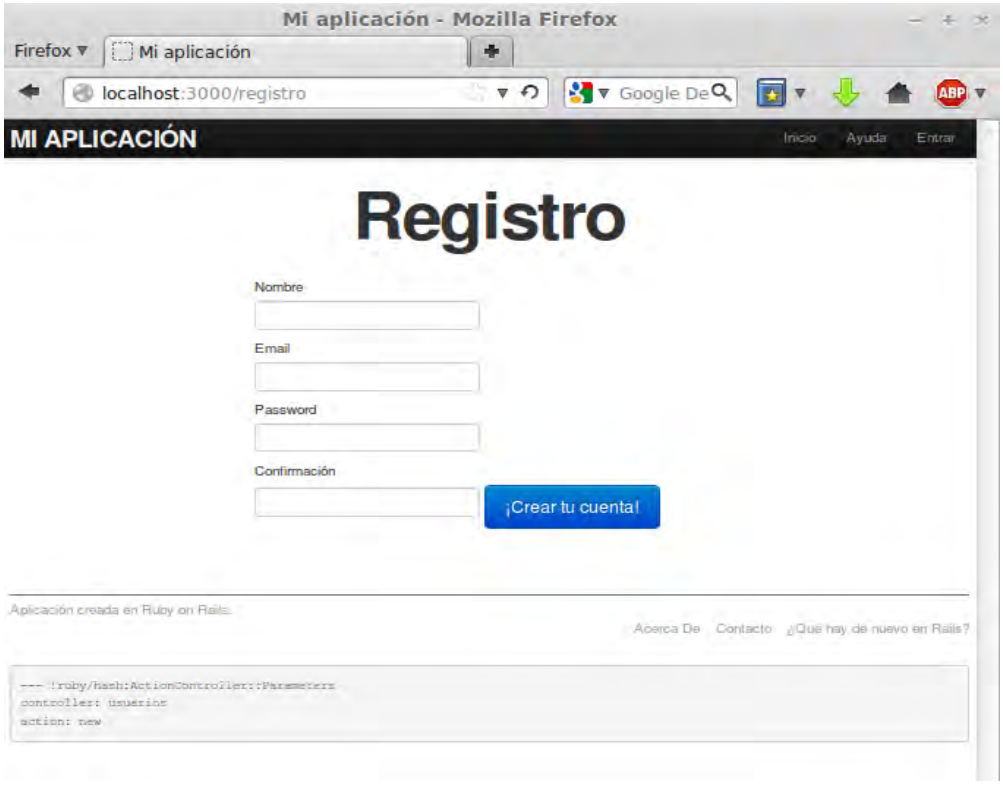


Figura 8.14. Página de registro sin estilo.

Observamos que aparecen los campos correspondientes a la base de datos gracias a `@user`, pero el botón no aparece del todo bien, agreguemos el estilo para el botón a nuestro archivo **estilo.css.scss**:

```
152
153  /*Estilo del botón para el formulario de registro*/
154
155  input, textarea, select, .uneditable-input{
156    border: 1px, solid #bbb;
157    width: 100%;
158    margin-bottom: 15px;
159    @include box_sizing;
160  }
161
162  input{
163    height: auto !important;
164  }
```

El resultado final se muestra en la Figura 8.15.

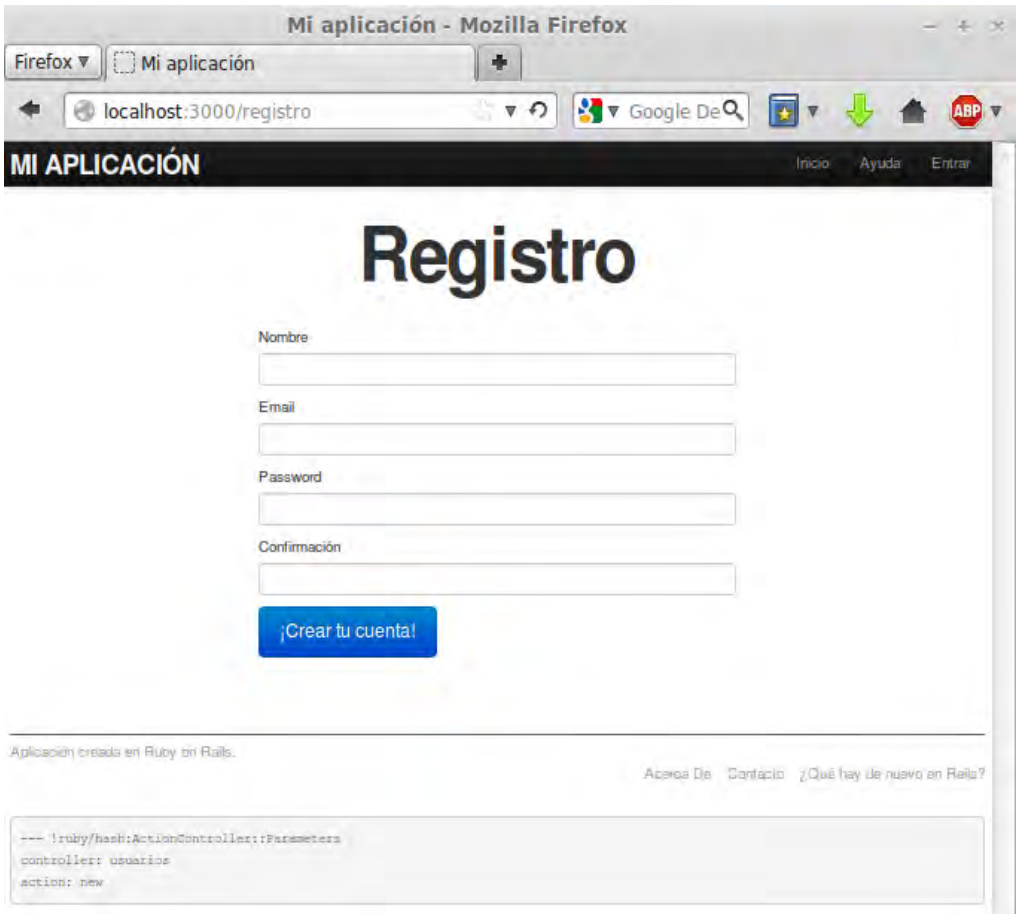


Figura 8.15. Página de registro con estilo.

8.2.2. Formulario con HTML

En el apartado anterior generamos la vista correspondiente al formulario de registro, resultando un código *HTML* generado automáticamente. Veamos parte de este código que `form_for` generó.

```
<form accept-charset="UTF-8" action="/usuarios" class="new_usuario" id="new_usuario" method="post"><div

    <label for="usuario_nombre">Nombre</label>
    <input id="usuario_nombre" name="usuario[nombre]" type="text" />

    <label for="usuario_email">Email</label>
    <input id="usuario_email" name="usuario[email]" type="text" />

    <label for="usuario_password">Password</label>
    <input id="usuario_password" name="usuario[password]" type="password" />

    <label for="usuario_password_confirmation">Confirmación</label>
    <input id="usuario_password_confirmation" name="usuario[password_confirmation]" type="password" />

    <input class="btn btn-large btn-primary" name="commit" type="submit" value="¡Crear tu cuenta!" />
</form>  </div>
```

Omitimos una parte del código generado que es el token de autenticidad (*token authenticity*) que automáticamente *Rails* añade para impedir un ataque de tipo *CSRF*[62] (C*ross-Site Request Forgery*, por sus siglas en inglés). Veamos la estructura interna del documento:

El código embebido:

```
<%= atributo.label :nombre %>
<%= atributo.text_field :nombre %>
```

Genera el siguiente código en *HTML*:

```
<label for="usuario_nombre">Nombre</label>
<input id="usuario_nombre" name="usuario[nombre]" type="text" />
```

Y este código:

```
<%= atributo.label :password %>
<%= atributo.password_field :password %>
```

Genera:

[62] Para más información consultar el siguiente enlace: http://stackoverflow.com/questions/941594/understand-rails-authenticity-token

```
<label for="usuario_password">Password</label>
<input id="usuario_password" name="usuario[password]" type="password" />
```

Así por ejemplo vemos que `type="text"` genera un campo de texto y muestra su contenido, parecido al campo de contraseña generado por `type:"password"` sólo que éste no muestra los caracteres ingresados por cuestiones de seguridad (Figura 8.16.)

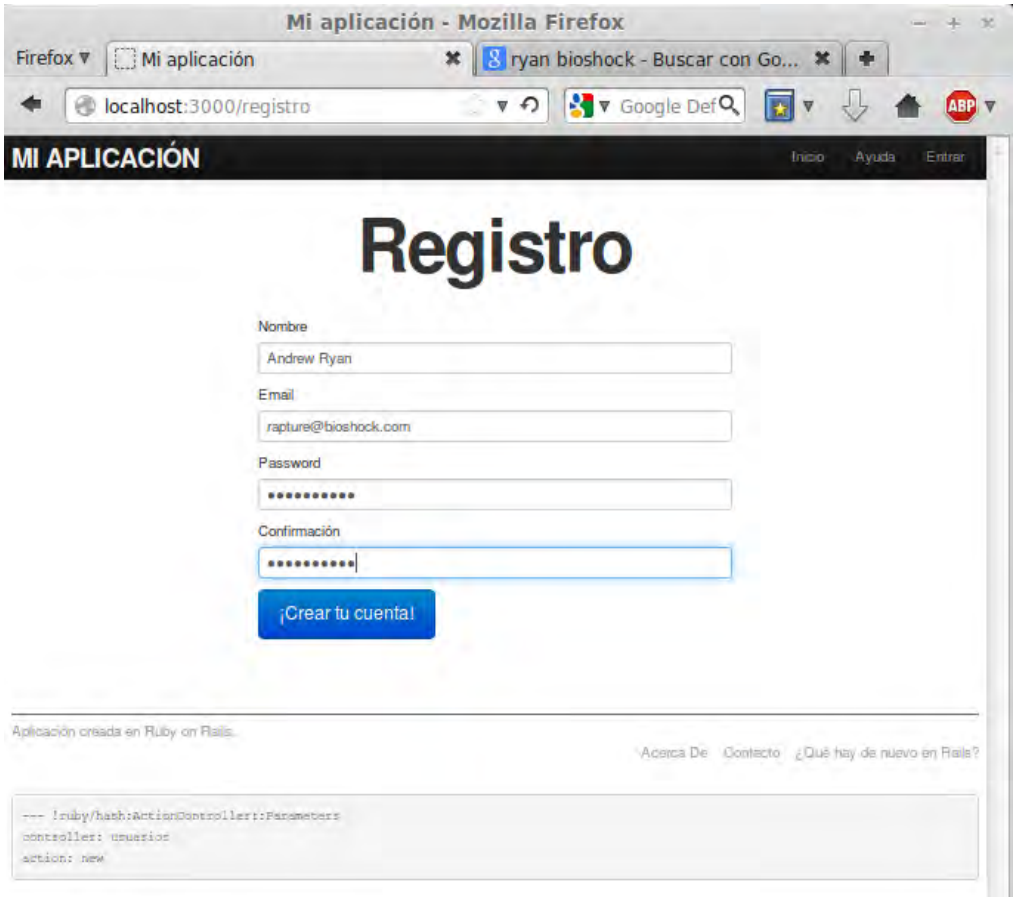


Figura 8.16. Formulario y campos.

La clave para la creación de nuevos usuarios se debe al atributo *nombre* en cada `input` del código *HTML* generado, los valores que tiene este atributo permiten a *Rails* construir y a la vez inicializar un *hash* (a través de la variable `params`) que almacena los datos necesarios para los usuarios nuevos.

Otro elemento importante es la etiqueta `form`, creada por *Rails* a través del objeto `@usuario`, esto es porque cada objeto en *Ruby* conoce su propia clase, *Rails* entonces intuye que `@usuario` es un objeto de la clase `Usuario`; además, *Rails* también construye un formulario con un método de tipo `post`.

8.3. Problemas con el registro de usuarios

Para este punto ya tenemos un formulario casi completo, en este apartado haremos que el formulario al recibir datos inválidos, recargue la página y muestre el registro previo pero con los errores encontrados. Algo como la maqueta de la Figura 8.17.

Figura 8.17. Maqueta de errores en inicio de sesión.

Cuando añadimos `resources :usuarios` en la sección 8.1.2. al archivo **routes.rb** automáticamente nuestra aplicación responde a las *URLs REST* de la Tabla 8.1. La idea aquí es que la acción `create` (que maneja peticiones tipo *POST*) use la petición del formulario para crear un nuevo usuario, utilizando `Usuario.new`, probar (sin éxito) y salvar al usuario, luego, recargar la página mostrando los problemas encontrados.

Vamos a modificar el archivo **usuarios_controller.rb** añadiendo el método `create` y dentro de éste, una condición `if-else` que nos permite manejar los casos en los que no

tengamos éxito en el registro y por otro lado cuando sí se cumplan, todo esto basado en el valor que retorne `@usuarios.save` (verdadero o falso).

usuarios_controller.rb

```
4     @usuario = Usuario.find(params[:id])
5   end
6
7   def new
8     @usuario = Usuario.new
9   end
10
11  def create
12    @usuario = Usuario.new(params[:usuario])
13    if @usuario.save
14        #Pasamos aquí si el salvar nos retorna true
15    else
16        render 'new'
17    end
18  end
```

El código no está completo pero nos sirve para empezar, primero vamos a añadir parámetros que no cumplan con los campos requeridos, nos despliega la siguiente pantalla indicando los errores (Figura 8.18 y Figura 8.19):

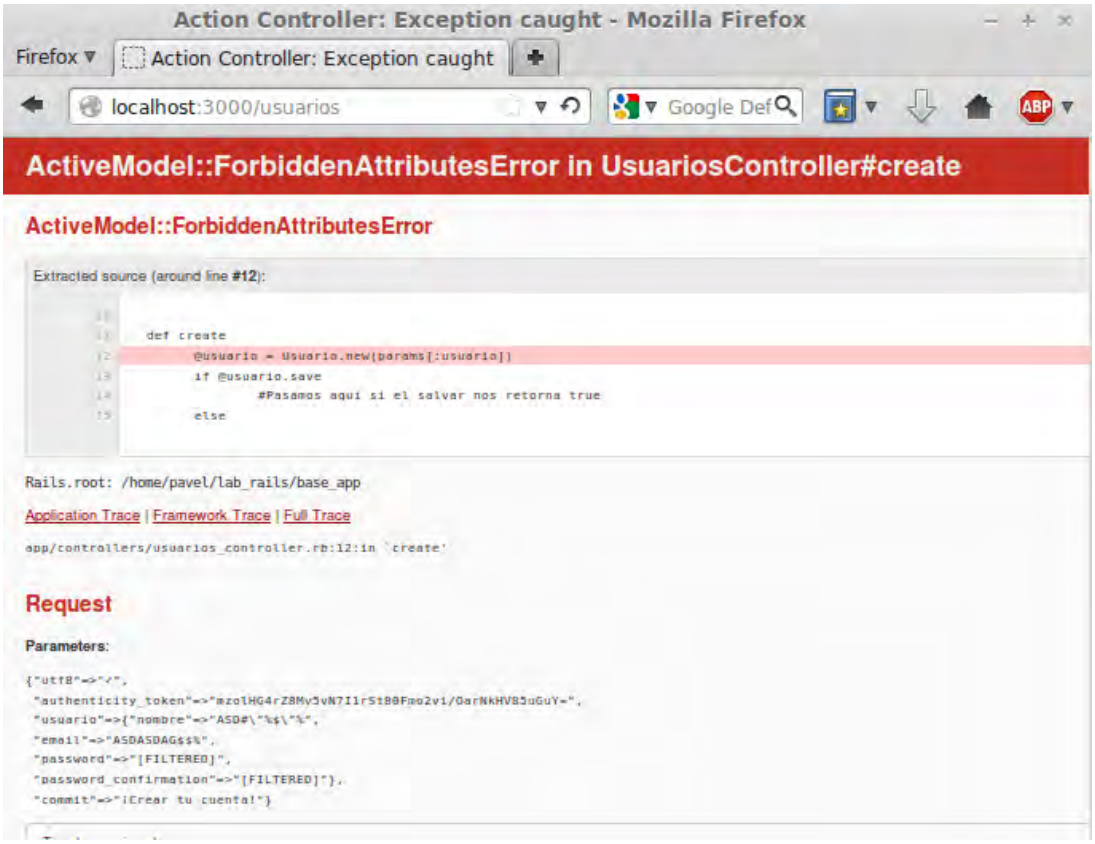


Figura 8.18. Página de error.

Figura 8.19. Parámetros inválidos.

En la Figura 8.19, se aprecia un *hash* con los parámetros de `usuario` dados en el formulario, éste *hash* se le pasa al controlador `Usuario` como el argumento `params` y que éstos contienen la información sobre cada una de la peticiones que se hacen. Cuando enviamos la petición, los parámetros se muestran como un *hash* de hashes en el que se introduce la variable llamada `params` (a propósito) en la sesión de la consola. La información de *debug* muestra los atributos que se pasaron al enviar la petición del formulario, donde las claves vienen de los nombres de atributos dentro de las etiquetas `input`, es decir, el valor de:

```
<input id="usuario_nombre" name="usuario[nombre]" type="text" />
```

en el que el *nombre* con "usuario[nombre]" es el atributo `nombre` en el *hash* `usuario`.

8.3.1. Asegurando parámetros

Lo que hemos estado haciendo es una asignación masiva, es decir, inicializar una variable de *Ruby* con un *hash* de valores, como por ejemplo en:

```
@usuario = Usuario.new(params[:usuario])
```

En la línea anterior tomada del archivo **usuarios_controller.rb** resulta que inicializar todo el *hash* `params` es extremadamente peligroso ya que estamos pasando TODOS los datos a `User.new` por cada usuario. Supongamos que además de los atributos con los que ya contamos, nuestro modelo *Usuario* incluye un atributo extra *admin* que se usa para identificar a los usuarios con niveles de administrador en el sistema. La forma de inicializar el atributo *admin* con un valor `true`, es a través del valor booleano `admin = '1'` - y que sería parte de los parámetros `params[usuario]`- pero al hacer esto, abrimos la

posibilidad de que cualquier usuario del sitio pueda tener privilegios de administrador incluyendo `admin = '1'` en la petición web.

En *Rails 4* para evitar el riesgo anterior, se usa una técnica llamada *strong parameters* que actúa sobre el controlador. Esto nos permite especificar los parámetros requeridos y qué parámetros serán los únicos accesibles. A continuación, vamos a añadir un símbolo `:usuario` al *hash* `params` y vamos a permitir los atributos *nombre*, *email*, *password* y *password_confirmation*. Realizamos lo siguiente:

```
params.require(:usuario).permit(:nombre, :email,
:password, :password_confirmation)
```

El código de arriba retorna una versión del *hash* `params` sólo con esos atributos y levantando un error si `:usuario` falta. Para hacer más fácil el uso de éstos parámetros, introducimos el método auxiliar `usuario_params` que retorna un *hash* con las asignaciones apropiadas y lo usa como argumento, sustituyendo a `params[:usuario]`:

```
@usuario = Usuario.new(usuario_params)
```

Lo anterior sólo se usa internamente en el controlador `Usuarios` y no necesita no ser expuesto a usuarios externos en la web, para esto debemos usar la clave de *Ruby* `private` como sigue:

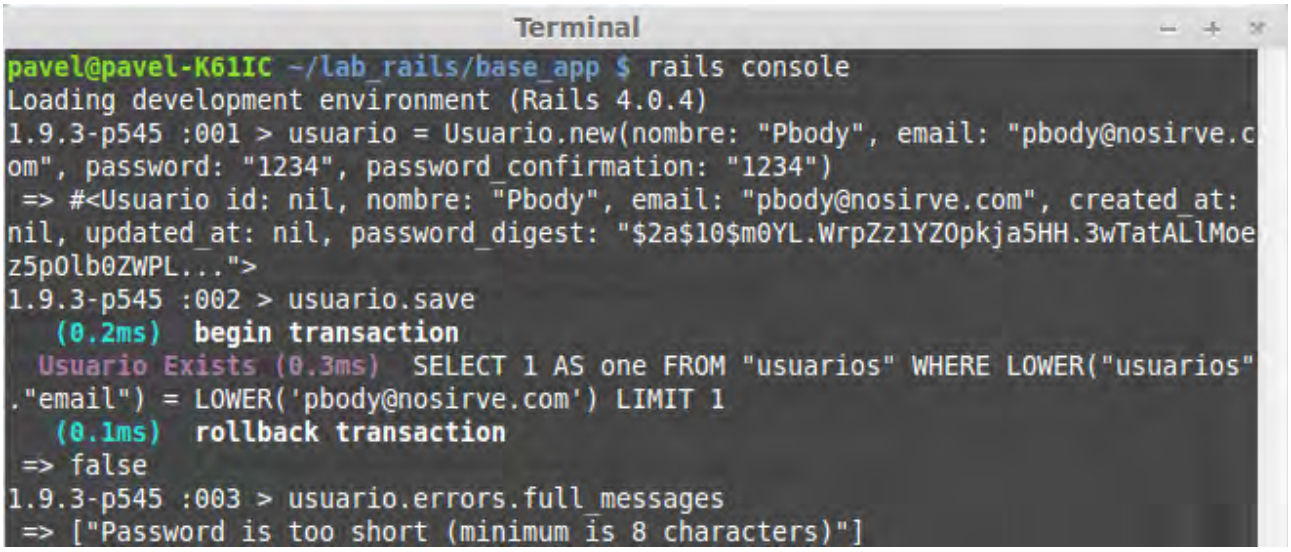
usuarios_controller.rb

```
1  class UsuariosController < ApplicationController
2
3    def show
4      @usuario = Usuario.find(params[:id])
5    end
6
7    def new
8      @usuario = Usuario.new
9    end
10
11   def create
12     @usuario = Usuario.new(usuario_params)
13     if @usuario.save
14       #Pasamos aquí si el salvar nos retorna true
15     else
16       render 'new'
17     end
18   end
19
20   private
21
22   def usuario_params
23     params.require(:usuario).permit(:nombre, :email, :password, :password_confirmation)
24   end
25 end
```

8.3.2. Manejo de errores para el registro de usuario

Para el manejo de errores cuando un usuario no puede crearse, mostraremos mensajes de error que indiquen los problemas que afectaron al registro. *Rails* automáticamente provee de estos mensajes a las validaciones en el modelo *Usuario*. Por ejemplo:

```
$ rails console

>> usuario = Usuario.new(nombre: “Pbody”, email:
“pbody@nosirve.com”, password: “1234”, password_confirmation:
“1234”)

>> usuario.save
=> false

>> usuario.errors.full_messages
```



Cuando intentamos salvar al usuario y falla (como se muestra en la sesión de consola anterior), se genera una lista de mensajes de error que se asocian al objeto `@usuario`. Para mostrar estos errores en el navegador, debemos hacer un *render* de un parcial con todos estos errores dentro la página `new`.

new.html.erb

```
<% provide(:title, 'Registro') %>
<h1>Registro</h1>

<div class = "row">
  <div class = "span6 offset3">
    <%= form_for(@usuario) do |atributo| %>
      <%= render 'shared/mensajes_error' %>

        <%= atributo.label :nombre %>
```

Notemos que estamos haciendo un *render* a un parcial (*mensajes_error*), éste se encuentra sobre el directorio *shared* en el que guardamos los parciales que vamos a usar en distintas vistas de múltiples controladores. Entonces creamos el directorio y el respectivo parcial (Código 8.1):

```
$ mkdir app/views/shared
```

app/views/shared/_mensajes_error.html.erb

_mensajes_error.html.erb

```
<% if @usuario.errors.any? %>
  <div id="errores_estilo">
    <div class="alert alert-error">
      <% if @usuario.errors.count == 1 %>
        Existe 1 error
      <% else %>
        Existen <%= @usuario.errors.count %> errores
      <% end %>
    </div>
    <ul>
    <% @usuario.errors.full_messages.each do |msg| %>
        <% if msg == "Nombre can't be blank" %>
            <li>* <%= "Debes llenar el campo con tu nombre" %></li>
        <% elsif msg == "Email can't be blank" %>
            <li>* <%= "Debes llenar el campo con tu correo electrónico (email)" %></li>
        <% elsif msg == "Email is invalid" %>
            <li>* <%= "El correo electrónico que proporcionaste no es válido" %></li>
        <% elsif msg == "Password can't be blank" %>
            <li>* <%= "La contraseña no puede ser vacía" %></li>
        <% elsif msg == "Password is too short (minimum is 8 characters)" %>
            <li>* <%= "La longitud mínima para la contraseña es de 8 caracteres" %></li>
        <% elsif msg == "Password confirmation doesn't match Password" %>
            <li>* <%= "La contraseña no coincide" %></li>
        <% elsif msg == "Email has already been taken" %>
            <li>* <%= "El correo ya está registrado" %></li>
        <% end %>
    <% end %>
    </ul>
  </div>
<% end %>
```

Código 8.1. **_mensajes_error.html.erb**

Este parcial hace uso del método `count` que retorna el número de errores. Otro método es `any?`, el cual retorna un valor booleano `true` si hay cualesquiera elementos presentes. El código genera los mensajes de error en español, ya que, como se ha visto anteriormente a través de la consola, los mensajes se muestran en inglés. Aquí simplemente

tomamos el error en inglés y lo traducimos acorde a nuestra aplicación como se ve en la Figura 8.20.

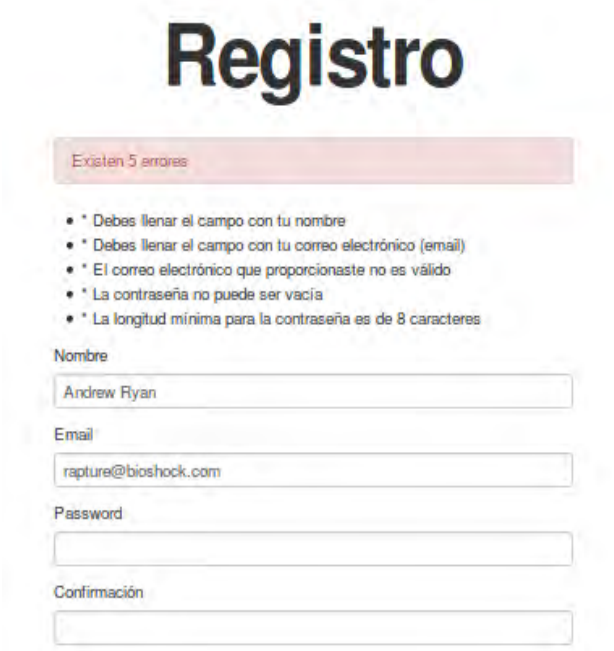


Figura 8.20. Errores de registro.

Para añadir estilo a los mensajes de error, añadimos el *id* `error_estilo`. *Rails* agrupa los errores en etiquetas `div` con la clase *SCSS* `field_with_errors`. Estas etiquetas nos permiten dar estilo a los mensajes de error, esto lo logramos incluyendo dos clases de *Bootstrap*: `control-group` y `error`; haciendo uso de la función `@extend`. Todo lo anterior hace que los campos con errores, aparezcan en rojo como se muestra en la Figura 8.21.

```
163   height: auto !important;
164 }
165
166 /* Estilo para errores */
167
168 #errores_estilo{
169     color: #f00;
170     ul{
171         list-style: none;
172         margin: 0 0 30px 0;
173     }
174 }
175
176 .field_with_errors{
177     @extend .control-group;
178     @extend .error;
179 }
```

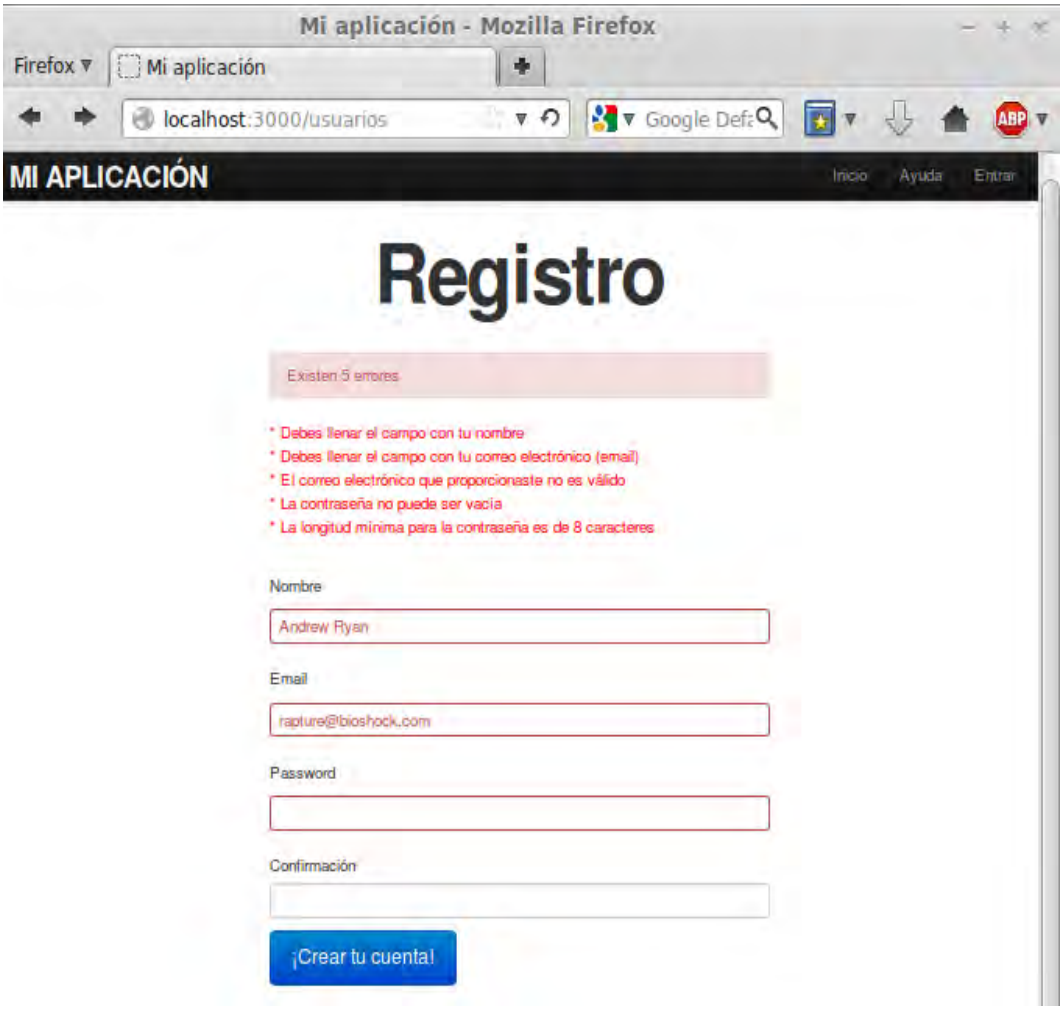


Figura 8.21. Errores con estilo.

8.4. Registro de usuarios correcto

Ya que hemos manejado los errores que se pueden presentar en el registro de usuarios, ahora toca turno al caso en que no existan errores en el registro y pasemos a guarda al nuevo usuario en la base de datos. Lo que haremos será salvar al usuario, si pasa, la información de dicho usuario se escribe en la base y redirigimos al usuario a la página de perfil – que tendrá un saludo- como en la maqueta siguiente:

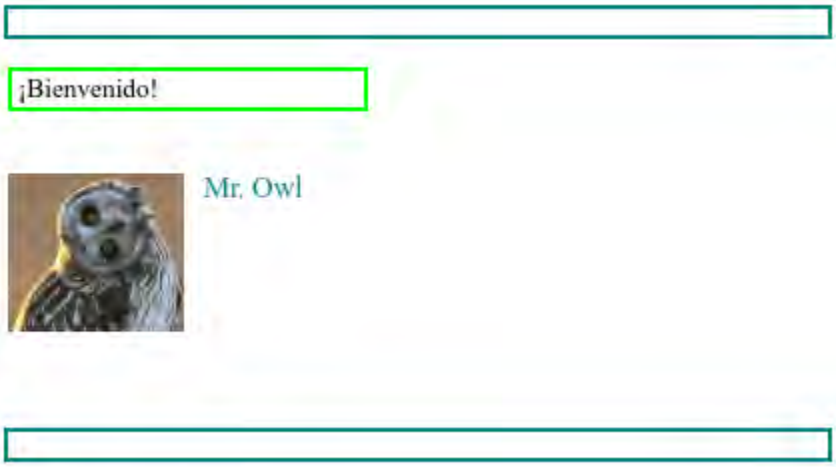


Figura 8.22. Maqueta de registro exitoso.

Para salvar al usuario usaremos una condicional rellenando el código del controlador **usuarios_controller.rb**, si puede salvarse el usuario entonces redirigimos a la página de perfil, el caso contrario ya lo habíamos resuelto y era recargar la página con los errores.

```
11  def create
12    @usuario = Usuario.new(usuario_params)
13    if @usuario.save
14      redirect_to @usuario
15    else
16      render 'new'
17    end
18  end
```

8.4.1. La herramienta flash

Antes de enviar un registro valido, debemos de dar una alerta o un mensaje a modo de notificación que de la bienvenida y entonces desaparezca si accedemos a otra página del sitio o si recargamos la misma página donde se notifica. Para tal fin, *Rails* cuenta con una herramienta llamada *flash*, que podemos trabajarla como un *hash*. Veamos un ejemplo de su funcionamiento en consola:

```
Terminal
pavel@pavel-K61IC ~/lab_rails/base_app $ rails console
Loading development environment (Rails 4.0.4)
1.9.3-p545 :001 > flash = {exito: "¡Funcionó!", error: "Hubo un problema"}
 => {:exito=>"¡Funcionó!", :error=>"Hubo un problema"}
1.9.3-p545 :002 > flash.each do |clave, valor|
1.9.3-p545 :003 >     puts "#{clave}"
1.9.3-p545 :004?>     puts "#{valor}"
1.9.3-p545 :005?>   end
exito
¡Funcionó!
error
Hubo un problema
 => {:exito=>"¡Funcionó!", :error=>"Hubo un problema"}
```

Podemos añadir dicha notificación a nuestra aplicación, a través del *layout* **application.html.erb** como sigue:

application.html.erb

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title><%= titulo_base(yield(:titulo)) %></title>
5    <%= stylesheet_link_tag    "application", media: "all", "data-turbolinks-track" => true %>
6    <%= javascript_include_tag "application", "data-turbolinks-track" => true %>
7    <%= csrf_meta_tags %>
8  </head>
9  <body>
10     <%= render 'layouts/etiquetaHeader'%>
11     <div class="container">
12     <% flash.each do |clave, valor| %>
13         <div class="alert alert-<%= clave %>"><%= valor %></div>
14     <% end %>
15       <%= yield %>
16       <%= render 'layouts/etiquetaFooter' %>
17       <%= debug(params) if Rails.env.development? %>
18     </div>
19   </body>
20 </html>
```

El código anterior inserta etiquetas `div` a cada elemento perteneciente al *flash* junto con una clase *CSS* que indica el tipo de mensajes que van a mostrarse. Por ejemplo, `flash[:exito] = "Hola Mundo"` entonces las líneas:

```
<% flash.each do |clave, valor| %>
    <div class="alert alert-<%= clave %>"><%= valor %></div>
<% end %>
```

Producen la salida:

```
<div class="alert alert-success">Hola Mundo</div>
```

Ahora, para que el mensaje se despliegue después de rellenar los campos correctamente. modificamos el archivo **usuarios_controller.rb**. Vamos a usar el símbolo `:success` que para *flash* es una palabra que indica un registro correcto y mostrará una notificación de color verde, por otro lado, si usamos el símbolo `:error` mostrará la notificación en rojo, cualquier otra cosa mostrará la notificación en amarillo.

```
12    def create
13      @usuario = Usuario.new(usuario_params)
14      if @usuario.save
15        flash[:success] = "Te has registrado con éxito: #{@usuario.nombre}!"
16        redirect_to @usuario
17      else
18        render 'new'
19      end
20    end
21
22    private
23
24    def usuario_params
25      params.require(:usuario).permit(:nombre, :email, :password, :password_confirmation)
26    end
27  end
28
```

8.4.2. Registrando un usuario

Ahora que ya tenemos todo correctamente para el registro de usuarios, llenemos los campos con información válida. Si todo sale bien, veremos la pantalla de la Figura 8.23, mostrando la notificación de un registro exitoso (en verde) y el nombre de usuario. Si reiniciamos la misma página, el mensaje de notificación ya no se mostrará (Figura 8.24)[63].

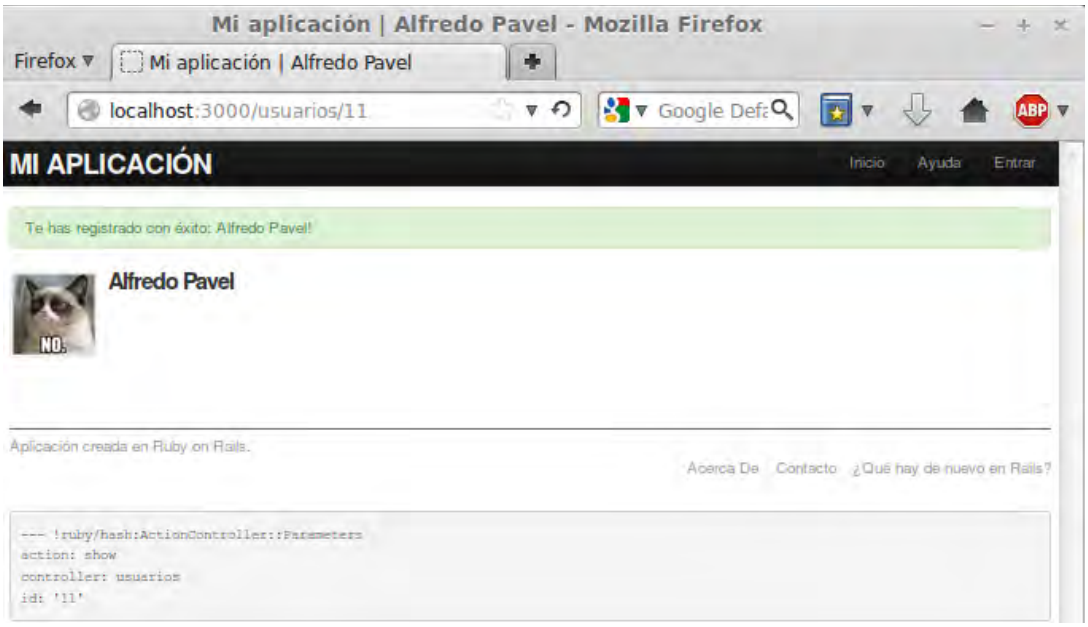


Figura 8.23. Notificación de registro exitoso.

[63] En este caso, el id de usuario es 11 ya que previamente se hicieron algunos cambios en la base pero esto no afecta en nada el resultado, sólo cambia el id de usuario.

Figura 8.24. Actualizando la página de perfil.

Por último, verificamos que el usuario creado se encuentre en la base de datos:

```
1.9.3-p545 :023 > Usuario.find_by(nombre: "Alfredo Pavel")
  Usuario Load (0.4ms)  SELECT "usuarios".* FROM "usuarios" WHERE "usuarios"."no
mbre" = 'Alfredo Pavel' LIMIT 1
 => #<Usuario id: 11, nombre: "Alfredo Pavel", email: "atomsk2.0@gmail.com", cre
ated_at: "2014-08-06 16:40:01", updated_at: "2014-08-06 16:40:01", password_dige
st: "$2a$10$upp952BltfnBN28L1nIREOu8.KJZaXA9LRnkDyjdJpAx...">
```

8.5. Guardando cambios

Como en los capítulos anteriores, vamos a guardar nuestros cambios y añadirlos a la rama principal. Hasta aquí hemos acabado el modelo y el registro de usuarios, en el siguiente capítulo veremos el inicio de sesión de usuarios, manejo de sesiones y cierre de sesión.

```
$ git add .

$ git commit -m ‘Registro de usuarios finalizado’

$ git checkout master

$ git merge registro_usuarios

$ git push origin master
```

Capítulo 9: Inicio y Cierre de Sesión

9. Desarrollo

Es hora de que los alumnos que se hayan registrado satisfactoriamente, sean capaces de iniciar y cerrar su sesión. Esto nos brinda tener una personalización basada en el estado de la sesión y en la identidad del usuario que inicio sesión.

Haciendo uso de las sesiones también estamos añadiendo un modelo de seguridad, en el que estamos restringiendo el acceso a ciertas páginas, dependiendo de la identidad del usuario. Sólo usuarios que inician sesión tienen permitido modificar sus datos. El inicio de sesión también incluirá ciertos privilegios a los usuarios que sean administradores, como por ejemplo, borrar usuarios de la base de datos.

Creamos y nos pasamos a una rama dedicada al desarrollo de esta sección:

```
$ git checkout master

$ git checkout -b iniciar_cerrar_sesion
```

9.1. Sesiones

Primeramente una sesión, es una conexión semi permanente entre dos computadoras (como una computadora cliente ejecutando una navegador web y un servidor ejecutando *Rails*). Al tener como opción el inicio de sesión nos podemos enfrentar a distintas situaciones el manejo de las mismas: "olvidar" la sesión cuando cerramos nuestro navegador web, usar un *checkbox* para sesiones permanentes, y automáticamente recordar las sesiones hasta que el usuario las cierre de forma manual. Nosotros utilizaremos la última opción: cuando uno de los usuarios inicie sesión, ésta se va a recordar hasta que el usuario la cierre de forma explicita (que será parte de la interfaz, el cierre de sesión).

Es conveniente manejar las sesiones como recursos de tipo *REST*, las sesiones tendrán una página de inicio de sesión para nuevas sesiones (función `new` dentro de *REST*), si el inicio de sesión fue exitoso, la sesión se crea (`create`); cuando el usuario cierra sesión ésta se destruirá (`destroy`) . Como recurso de *REST* (que también tendrá una persistencia de datos vía el modelo *Usuarios*), se usará una *cookie*, que es un archivo de texto plano que se sitúa en el navegador del usuario. Con la *cookie* tendremos un método de autenticación más seguro.

En este capítulo vamos a construir el controlador *Sesiones*, así como las acciones correspondientes a él; también un formulario para el inicio de sesión de los usuarios.

9.1.1. Controlador para sesiones

Como se dijo previamente, los elementos para el inicio de sesión en nuestra aplicación, contienen algunas acciones particulares de *REST*, el formulario para iniciar sesión es parte de la acción `new`, cuando iniciamos sesión haremos uso de una petición *POST* haciendo uso de la acción `create`, por último, cuando cerremos la sesión en curso se tendrá una petición DELETE que corresponde a una acción `destroy`.

Generamos nuestro controlador *Sesiones*:

```
$ rails generate controller Sesiones
```

Parecido al formulario de registro del capítulo anterior, modelamos la siguiente maqueta (Figura 9.1) para el inicio de sesión:

Figura 9.1. Maqueta de inicio de sesión.

Para empezar vamos a definir las rutas para nuestro recurso *Sesiones*, también vamos a añadir la ruta personalizada para la página de inicio de sesión. Usamos nuevamente el método `resources` para definir las rutas. Como las sesiones no se vana mostrar o a editar, restringimos las acciones a las ya comentadas anteriormente: `new`, `create`, `destroy`. Modificamos nuestro archivo **routes.rb**:

routes.rb

```
BaseApp::Application.routes.draw do
  resources :usuarios
  resources :sesiones, only: [:new, :create, :destroy]
  root 'paginas_estaticas#inicio'
  match '/registro',  to: 'usuarios#new', via: 'get'
  match '/iniciases', to: 'sesiones#new', via: 'get'
  match '/cierrases', to: 'sesiones#destroy', via: 'delete'
  match '/ayuda', to: 'paginas_estaticas#ayuda', via: 'get'
  match '/acercade', to: 'paginas_estaticas#acercade', via: 'get'
  match '/contacto', to: 'paginas_estaticas#contacto', via: 'get'


  # The priority is based upon order of creation: first created -> highest priou
```

Ya con el recurso definido, las rutas generadas se muestran en la Tabla 9.1. Notemos que las rutas para el cierre e inicio de sesión son personalizadas pero la ruta para crear una sesión es la que se genera por default (`nombreRecurso_path`).

Tabla 9.1. Rutas *REST*.

Petición HTTP	URL	Ruta Personalizada	Acción	Propósito
GET	/iniciases	iniciases_path	new	Página de niciao de sesión
POST	/sesiones	sesiones_path	create	Crea la nueva sesión
DELETE	/cierrases	cierrases_path	destroy	Destruye una sesión

Ahora vamos a añadir las acciones correspondieres a nuestro controlador (Código 9.1):

app/controllers/sesiones_controller.rb

sesiones_controller.rb

```
class SesionesController < ApplicationController

    def new
    end

    def create
    end

    def destroy
    end
end
```

Código 9.1. **sesiones_controller.rb**

Por último, toca turno a definir la versión inicial de la página de inicio de sesión, está página va a existir en la ruta **app/views/sesiones/new.html.erb** que debemos crear. Sólo vamos a definir el titulo de la página y un encabezado sencillo:

```
new.html.erb
1  <% provide(:title, 'Inicio de sesión') %>
2  <h1>Inicia Sesión</h1>
3
```

9.1.2. Formulario

La maqueta de nuestro formulario quedaría como en la Figura 9.2, en la que ya tomamos incluso un error en el inicio de sesión y el despliegue del mensaje de error:

Figura 9.2. Maqueta

De la misma manera que en el formulario de registro del capítulo anterior, vamos a utilizar el *helper* `form_for`, sólo que esta vez no pasaremos como argumento la variable de instancia `@usuario`, si no que al querer hacer uso de sesiones, le pasamos como argumentos el nombre del recurso (*sesiones* en este caso) y su ruta correspondiente.

new.html.erb

```
1  <% provide(:title, 'Inicio de sesión') %>
2  <h1>Iniciar sesión</h1>
3
4  <div class="row">
5    <div class="span6 offset3">
6
7      <%= form_for(:sesiones, url: sesiones_path) do |atributo| %>
8
9        <%= atributo.label :email %>
10       <%= atributo.text_field :email %>
11
12       <%= atributo.label :password %>
13       <%= atributo.password_field :password %>
14
15       <%= atributo.submit "Inicia sesión", class: "btn btn-large btn-primary" %>
16     <% end %>
17
18     <p>¿No tienes cuenta?
19       <br><%= link_to "¡Registrate ahora!", registro_path %></br>
20     </p>
21   </div>
22 </div>
```

Quedando nuestra página para iniciar sesión como sigue:

Figura 9.3. Página de inicio sesión.

Si observamos el código *HTML* que se genera en la interfaz, veremos que también genera un *hash* `params` con los campos `params[:sesiones][:email]` y `params[:sesiones][:password]`.

```
<form accept-charset="UTF-8" action="/sesiones" method="post"><div style="margin:0;padding:0;di

  <label for="sesiones_email">Email</label>
  <input id="sesiones_email" name="sesiones[email]" type="text" />

  <label for="sesiones_password">Password</label>
  <input id="sesiones_password" name="sesiones[password]" type="password" />

  <input class="btn btn-large btn-primary" name="commit" type="submit" value="Inicia sesión" />
</form>
```

9.1.3. Usando la información de una formulario

Como en el caso de registro, vamos a empezar con ver qué pasa cuando al llenar un formulario y enviar la petición, y entonces en caso de error, notificar con un mensaje. Luego, veremos el caso contrario en el que no existan errores validando de manera correcta, el email y la contraseña.

sesiones_controller.rb

```
1  class SesionesController < ApplicationController
2
3      def new
4      end
5
6      def create
7        render 'new'
8      end
9
10     def destroy
11     end
12 end
```

Vamos a rellenar la acción `create` de nuestro controlador Sesiones para que haga un *render* de la vista `new`. Si enviamos el formulario con los campos en blanco nos mostrará como resultado la Figura 9.4.

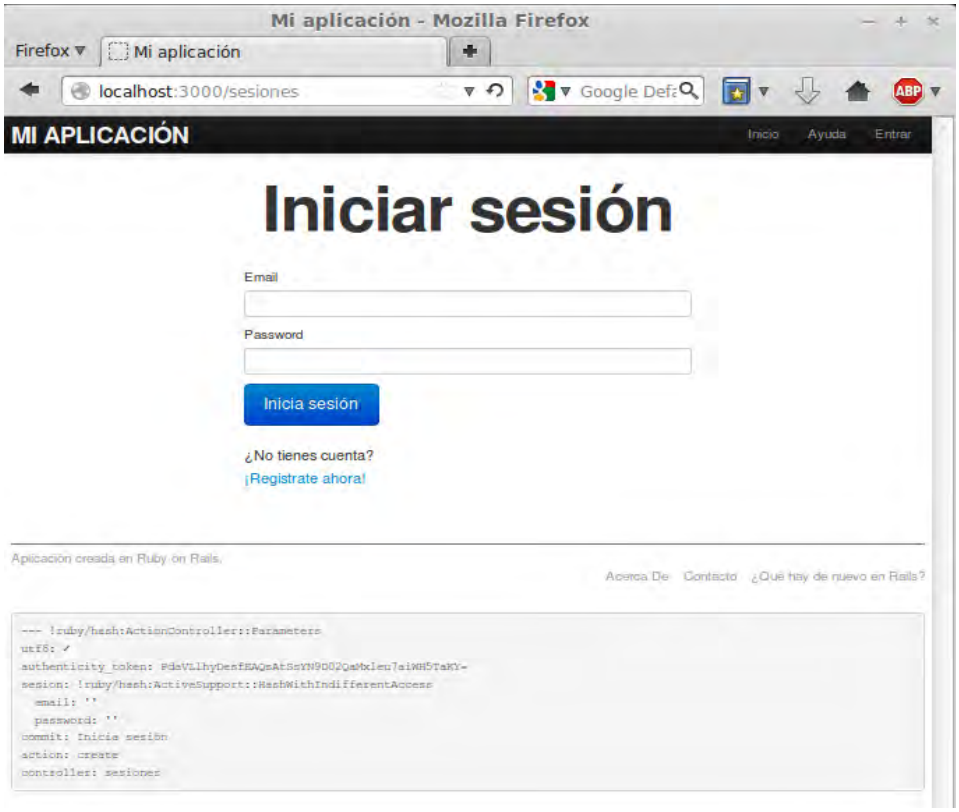


Figura 9.4. Resultado de un formulario vacío.

Si observamos la información del modo *debug*, los resultados del envío del formulario (email, password) se encuentran bajo la clave `:sesiones`.

```
--- !ruby/hash:ActionController::Parameters
utf8: ✓
authenticity_token: PdaVLlhyDesfEAQsAtSsYN9D02QaMxleu7aiWH5TaKY=
sesion: !ruby/hash:ActiveSupport::HashWithIndifferentAccess
  email: ''
  password: ''
commit: Inicia sesión
action: create
controller: sesiones
```

Figura 9.5. Información del modo *debug*.

Estos parámetros formulan un *hash anidado*, en particular, `params` contiene un *hash anidado* de la forma:

```
{ sesion: { email: "", password: "" } }
```

Con lo anterior, la acción `create` tendrá en su interior el *hash* `params` con toda la información necesaria para que el usuario sea autenticado a través de su email y su contraseña. Usaremos los métodos `find_by` de *Active Record* y el método `authenticate` que nos provee `has_secure_password`. La lógica que usaremos será el de comparar (a través de una sentencia `if`) los atributos de la sesión con los atributos de la base de datos, tomando en cuenta que al atributo *email* de la sesión lo debemos de pasar a minúsculas.

De lo anterior, cuando falle el inicio de sesión, mostraremos un mensaje de error pero que no se puede manejar de la misma forma que con el registro de usuarios. En este caso los errores no se asocian con un objeto de tipo *Active Record* ya que estamos trabajando con sesiones y éstas no pertenecen a él. En su lugar, hagamos uso de la herramienta *flash* para mostrar los mensajes de un inicio de sesión fallido. Con todo esto en cuenta, modificamos nuestro controlador *Sesiones* como sigue y vemos los resultados (Figura 9.6):

sesiones_controller.rb

```
1  #encoding: utf-8
2  class SesionesController < ApplicationController
3
4      def new
5      end
6
7      def create
8          usuario = Usuario.find_by(email: params[:sesion][:email].downcase)
9          if usuario && usuario.authenticate(params[:sesion][:password])
10             #Logueamos
11         else
12             flash[:error] = "Revisa que tu email y contraseña sean correctos"
13             render 'new'
14         end
15     end
16
17     def destroy
18     end
19 end
```

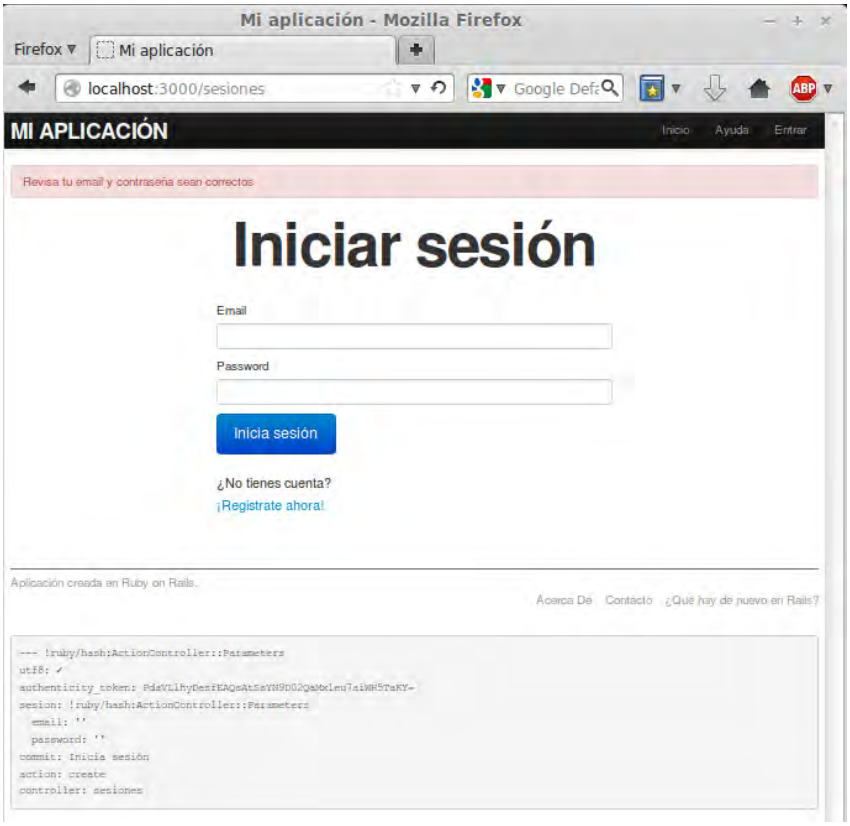


Figura 9.6. Notificación flash de error.

Pero, ¿qué pasa si enseguida de que nos notifica un error de sesión, damos clic al enlace que nos lleva al inicio? Si lo hacemos veremos que el mensaje *flash* aparece de nuevo (Figura 9.7), notamos entonces que el mensaje se muestra si hacemos una petición extra pero, cuando usamos la función `render` para mostrar una plantilla, no se cuenta como petición. En nuestro caso por ejemplo, que al pasar de la página de inicio de sesión (ya con un error) a la página de inicio, ésta última vendría a ser la primer petición que haríamos desde que tratamos de iniciar sesión y el mensaje *flash* aparece de nuevo.

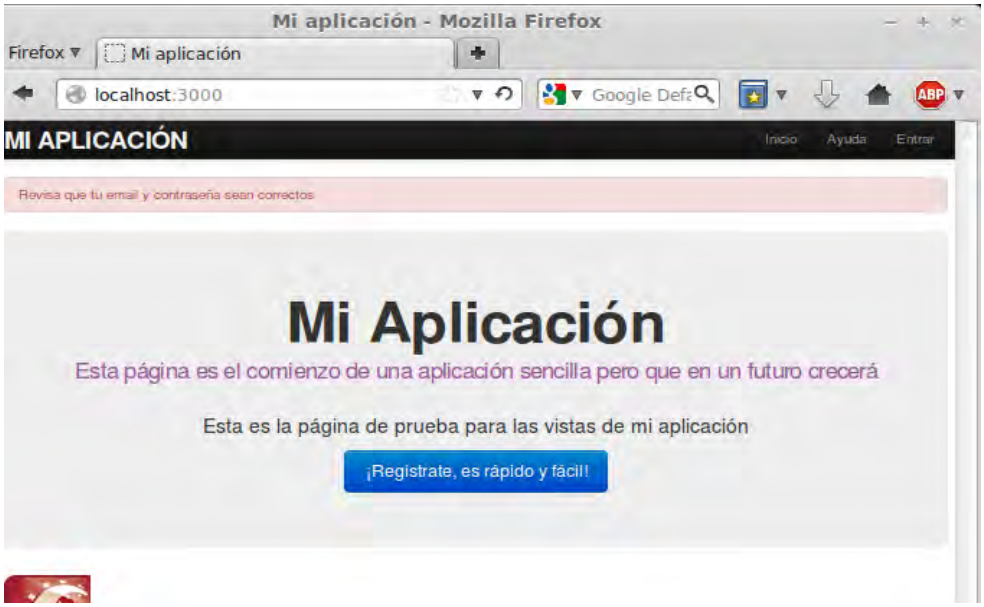


Figura 9.7. Error de notificación flash.

Este pequeño problema se reduce a un *bug* en nuestra aplicación y para corregirlo, haremos uso de `flash.now` que se creó para mostrar los mensajes *flash* en páginas que mostramos (con el método `render`). Con esto, al aparecer una vez el mensaje de notificación, ya no aparecerá de nuevo al hacer otra petición. El código del método `create` queda como sigue:

sesiones_controller.rb

```
1  #encoding: utf-8
2  class SesionesController < ApplicationController
3
4      def new
5      end
6
7      def create
8          usuario = Usuario.find_by(email: params[:sesion][:email].downcase)
9          if usuario && usuario.authenticate(params[:sesion][:password])
10             #Logueamos
11         else
12             flash.now[:error] = "Revisa que tu email y contraseña sean correctos"
13             render 'new'
14         end
15     end
16
17     def destroy
18     end
```

9.2. Inicio de sesión correcto

Igual que en secciones anteriores en los que primero nos enfocamos en el peor caso y luego en el caso esperado, vamos a tomar ahora el caso en que el alumno inicie sesión correctamente. Para tal fin, primeramente vamos a completar el método create del controlador *Sesione*, iniciando sesión con ayuda del método `inicia_sesion` y redigirigr al usuario hacia la página de perfil. El método `inicia_sesion` no está todavía definido pero será lo que haremos en esta sección.

sesiones_controller.rb

```
#encoding: utf-8
class SesionesController < ApplicationController

    def new
    end

    def create
        usuario = Usuario.find_by(email: params[:sesion][:email].downcase)
        if usuario && usuario.authenticate(params[:sesion][:password])
            inicia_sesion usuario
            redirect_to usuario
        else
            flash.now[:error] = "Revisa que tu email y contraseña sean correctos"
            render 'new'
        end
    end

    def destroy
    end
end
```

Ahora podemos implementar las sesiones, recordando al usuario cuando inicie sesión de manera "permanente" y terminar la sesión cuando el usuario explícitamente cierre la sesión. Las funciones que correspondan al inicio de sesión estarán tanto en las vistas como en un controlador. *Ruby* provee de un *módulo* para poder juntar dichas funciones y colocarlas en distintos lugares que es lo planeado para las funciones correspondientes a la autenticación. Podríamos tener un módulo completo para autenticar pero nuestro controlador *Sesiones* ya tiene consigo un módulo llamado `SesionesHelper`. Además, los *helpers* se integran a las vistas, lo único que debemos de hacer para usar las funciones del *helper* dentro del controlador, es incluir el módulo (`SesionesHelper`) dentro del controlador de la aplicación (**application_controller.rb**).

application_controller.rb

```
class ApplicationController < ActionController::Base
  # Prevent CSRF attacks by raising an exception.
  # For APIs, you may want to use :null_session instead.
  protect_from_forgery with: :exception
  include SesionesHelper
end
```

Dado que el protocolo *HTTP* no tiene estados, las aplicaciones web en las que requerimos un inicio de sesión, deben de tener una manera de dar seguimiento al progreso que tenga el usuario página por página. Una forma de dar seguimiento al estatus de un usuario es usar una sesión de *Rails* (a través de la función `session`)para guardar un *token_recuerdo* que sea igual al *id* del usuario. Algo como:

```
session[:token_recuerdo] = usuario.id
```

El objeto `session` hace disponible al *id* de usuario en cada página que visita, almacenando dicha información en una *cookie* que expira cuando el navegador se cierra. En cada página, la aplicación puede llamar a

```
Usuario.find(session[:token_recuerdo])
```

para retornar al usuario. De la forma en que *Rails* maneja las sesiones, hace al proceso seguro; si por ejemplo un usuario malicioso intenta burlar el *id* de usuario, *Rails* detecta que hay problema de coincidencia basado en un *id sesion* que se genera por cada sesión. Para nuestro diseño de la aplicación que implica sesiones persistentes, necesitamos un identificador permanente para cada usuario que se loguee. Debemos generar un *token_recuerdo* seguro y único para cada usuario y almacenarlo, como se dijo en líneas anteriores, en una *cookie* permanente (que permanece el estatus de sesión, incluso si se cierra el navegador).

El *token_recuerdo* que requerimos almacenar, debe de estar ligado con el usuario, por lo tanto debemos de añadirlo como un atributo al modelo *Usuario* quedando como se muestra en la Tabla 9.2.

Tabla 9.2. Modelo `Usuario` actualizado.

Usuario	
id	integer
nombre	string
email	string
password_digest	string
token_recuerdo	string
created_at	datetime
updated_at	datetime

Añadimos el *token* a nuestro modelo:

```
$ rails generate migration add_token_recuerdo_to_usuarios
```

Completamos la migración añadiendo el código que se muestra. Como queremos retornar usuarios a través del *token_recuerdo*, añadimos un índice *index* a la columna *token_recuerdo*.

20140820162758_add_token_recuerdo_to_usuarios.rb ×

```
1  class AddTokenRecuerdoToUsuarios < ActiveRecord::Migration
2    def change
3      add_column :usuarios, :token_recuerdo, :string
4      add_index :usuarios, :token_recuerdo
5    end
6  end
```

Luego, ejecutamos los cambios en la base con la migración de la base:

```
$ bundle exec rake db:migrate
```

Ya que hemos optado por un *token_recuerdo* usaremos el método `urlsafe_base64` del módulo `SecureRandom`[64] de la biblioteca de *Ruby*, este método regresa una cadena al azar de 16 caracteres que incluye 0-9, A-Z, a-z, "-", y "_" (64 posibilidades, *base64*). Esto evita que que la probabilidad de una colisión entre dos *tokens* sea muy poco probable.

La lógica a seguir es: guardar el *token_recuerdo* en el navegador y entonces almacenar el resultado de aplicarle una función *hash digest* en la base de datos. Entonces podemos loguear usuarios sólo con obtener el *token_recuerdo* de la *cookie*, calcular el *hash digest* y buscar entonces un *token* que recuerde al usuario y que coincida con el valor *digest* en la base de datos. Almacenamos sólo el *token_recuerdo* con función *hash* porque si nuestra base de datos se ve comprometida con un ataque, el atacante no podrá usar los *tokens* para loguearse.

Para que un *token* sea más seguro, debemos de cambiarlo cada vez que un usuario cree una nueva sesión, lo que significa que si un usuario roba una *cookie*, ésta expira al siguiente inicio de sesión, evitando lo que se conoce como *hijacked session*. Debemos asegurarnos de que al registrarse un nuevo usuario, éste será logueado automáticamente y debe tener por lo tanto, un *token* válido desde el inicio en que se loguea. Para lograr esto, debemos de crear un *token* inicial usando un *callback* (como con el email en capítulos anteriores), en este caso usaremos el *callback* `before_create` para asignar el *token* cuando el usuario se crea.

[64] Para más información sobre el módulo SecureRandom, visitar: http://www.ruby-doc.org/stdlib-1.9.3/libdoc/securerandom/rdoc/SecureRandom.html

Ahora en nuestra aplicación, el código que incluimos en el archivo **usuario.rb**:

usuario.rb

```
1  class Usuario < ActiveRecord::Base
2      before_save { self.email = email.downcase }
3      validates :nombre, presence: true, length: { maximum: 50 }
4      VALIDA_EMAIL = /\A[\w+\-.]+@[a-z\d\-]+(?:\.[a-z\d\-]+)*\.[a-z]+\z/
5      validates :email, presence: true, format: { with: VALIDA_EMAIL },
6                        uniqueness: { case_sensitive: false }
7      has_secure_password
8      validates :password, length: { minimum: 8 }
9
10     def Usuario.new_token_recuerdo
11       SecureRandom.urlsafe_base64
12     end
13
14     def Usuario.digest(token)
15       Digest::SHA1.hexdigest(token.to_s)
16     end
17
18 private
19
20     def create_token_recuerdo
21       self.token_recuerdo = Usuario.digest(Usuario.new_token_recuerdo)
22     end
23 end
24
25
```

Lo anterior hace una referencia a un método llamado `create_token_recuerdo`, y lo ejecuta antes de salvar a un usuario. Este código es usado sólo de manera interna por el modelo *Usuario*, así que no debe haber necesidad de exponerlo a usuarios externos, para lograr esto usamos la palabra reservada `private`.

El método `create_token_recuerdo`, necesita asignar a uno de los atributos de usuario y bajo este contexto, es necesario usar la palabra reservada `self` como precedente de *token_recuerdo* (que vendría a ser un equivalente en Java a `this`). Lo hacemos así ya que sin el `self`, crearíamos una variable local y nada más, por el contrario, al usar `self`, aseguramos que se asigne e inicie el *token_recuerdo* del usuario y como resultado será escrito en la base de datos junto con los otros atributos cuando el usuario es salvado (por eso en el capítulo 7, se usa `self.email` en lugar de `email`).

Vemos que la función *hash* que aplicamos es con el algoritmo *SHA1*, este algoritmo es más rápido que *Bcrypt* para encriptar contraseñas, esto es importante porque usuarios que estén con una sesión iniciada se estará ejecutando el algoritmo por cada página. *SHA1* es más menos seguro que *Bcrypt*, pero lo compensa el ser más rápido y esto es más que suficiente ya que el *token* a ser hasheado es una cadena de 16 dígitos al azar; la función *hexdigest* de *SHA1* para dicha cadena es prácticamente irrompible.

Por último, los métodos `digest` y `new_token_recuerdo` pertenecen a la clase `Usuario` ya que no necesitan una instancia de usuario para trabajar, y resultan ser métodos públicos (se encuentran por arriba de los métodos privados denotados por la

palabra reservada `private`) porque más adelante los tendremos fuera del modelo *Usuario*.

9.2.1. Método de inicio de sesión exitoso

Ya contamos con lo necesario para escribir el primer elemento para iniciar sesión, la función `inicia_sesion`. Nuestro método de autenticación consiste en colocar un *token* de recuerdo como una *cookie* en el navegador del usuario, entonces usar dicho *token* para encontrar al usuario en la base de datos cada vez que él visita una página. Esto se resume con el siguiente código y que hace uso del método `usuario_actual` que implementamos en la siguiente sección.

app/helpers/sesiones_helper.rb:

sesiones_helper.rb

```
1  module SesionesHelper
2
3    def inicia_sesion(usuario)
4      token_recuerdo = Usuario.new_token_recuerdo
5      cookies.permanent[:token_recuerdo] = token_recuerdo
6      usuario.update_attribute(:token_recuerdo, Usuario.digest(token_recuerdo))
7      self.usuario_actual = usuario
8    end
9  end
```

Código 9.2. **sesiones_helper.rb**

,

Creamos el *token*, luego, guardamos el *token* en las *cookie* del navegador; salvamos el *token* hasheado en la base de datos; por último, asignamos al usuario actual a el usuario que se da (cuando se manda llamar al método `inicia_sesion`). Esta última parte de asignar al usuario actual a `usuario` no es tan necesaria porque se la acción `create` hace un redireccionamiento inmediato, pero resulta ser una buena idea en caso de que quisiéramos usar el método `inicia_sesion` sin redireccionar.

En el código anterior hacemos uso de `update_attribute` para guardar el *token* en la base, lo hacemos así porque este método nos permite actualizar un sólo atributos sin tomar en cuenta las validaciones que tengamos (en este caso las validaciones de contraseña y confirmación de la misma). También hacemos uso de la utilidad `cookies` que proporciona *Rails*, permite manipular las *cookies* del navegador como si fueran *hashes*. Cada elemento de la *cookie* corresponde a un *hash* de dos elementos (un valor y opcionalmente una fecha de caducidad)[65]. Por ejemplo, si queremos guardar el *token* de recuerdo y que caduque en 20 años a partir de la fecha actual, tendríamos:

```
cookies[:token_recuerdo] = { valor: token_recuerdo, expires:
10.years.from_now.utc  }
```

[65] Para más información consultar: http://api.rubyonrails.org/classes/ActionDispatch/Cookies.html

La idea de que las cookies expiren en 20 años es tan común y usada que *Rails* añadió el método `permanent` para implementarla, es por eso que sólo escribimos:

```
cookies.permanent[:token_recuerdo] = token_recuerdo
```

Una vez que la *cookie* está lista, en las páginas que consultemos después podemos retornar información del usuario de la siguiente manera:

```
Usuario.find_by{ token_recuerdo: token_recuerdo }
```

9.2.2. Usuario actual

Anteriormente guardamos el *token* de recuerdo en una *cookie*, ahora vamos a ver la manera de retornar información del usuario en páginas subsecuentes. Como se dijo en la sección anterior, podemos retornar información a través de la línea `self.usuario_actual = usuario`. El propósito de tal línea (que es accesible tanto en vistas como en controladores) es construir cosas de la forma:

```
<%= usuario_actual.nombre %>
```

```
redirect_to usuario_actual
```

Notemos que la línea:

```
self.usuario_actual = usuario
```

es una asignación que debemos definir. *Ruby* nos brinda una sintaxis especial para definir dicha función de asignación:

sesiones_helper.rb

```
1  module SesionesHelper
2
3    def inicia_sesion(usuario)
4      token_recuerdo = Usuario.new_token_recuerdo
5      cookies.permanent[:token_recuerdo] = token_recuerdo
6      usuario.update_attribute(:token_recuerdo, Usuario.digest(token_recuerdo))
7      self.usuario_actual = usuario
8    end
9
10   def usuario_actual=(usuario)
11     @usuario_actual = usuario
12   end
13 end
```

En este caso vemos que podemos definir un método con el símbolo "=", cosa que en otros lenguajes no lo permitirían. Lo que hace es definir un método `usuario_actual=` expresamente diseñado para manejar las asignaciones a `usuario_actual`, es decir que el código:

```
self.usuario_actual =
```

es convertido a:

```
usuario_actual = (...)
```

En *Ruby* simple, podemos definir un segundo método, `usuario_actual`, pensado para retornar el valor de la variable `@usuario_actual`, como muestra el siguiente código:

sesiones_helper.rb

```
1  module SesionesHelper
2
3    def inicia_sesion(usuario)
4      token_recuerdo = Usuario.new_token_recuerdo
5      cookies.permanent[:token_recuerdo] = token_recuerdo
6      usuario.update_attribute(:token_recuerdo, Usuario.digest(token_recuerdo))
7      self.usuario_actual = usuario
8    end
9
10   def usuario_actual=(usuario)
11     @usuario_actual = usuario
12   end
13
14   def usuario_actual
15     @usuario_actual #no tiene efecto alguno por el momento
16   end
17 end
```

El problema del código anterior es que la sesión se olvida y no se guarda: tan pronto como el usuario cambie la página, la sesión terminará y al usuario se le cierra su sesión. Esto es por la naturaleza del protocolo *HTTP*, cuando un usuario hace una segunda petición, todas las variables involucradas toman sus valores por defecto y esto aplica también para las variables de instancia como `@usuario_actual` que tendría el valor nulo, o sea, `nil`. Para solucionar el problema, podemos buscar al usuario que tenga el *token* de recuerdo visto en la definición del método `inicia_sesion` y que se muestra en el código siguiente. Sólo recordar que como el *token* se encuentra en la base hasheado, primero debemos de aplicar el *hash* al *token* que proviene de la *cookie* antes de buscarlo en la base de datos.

sesiones_helper.rb

```ruby
module SesionesHelper

  def inicia_sesion(usuario)
    token_recuerdo = Usuario.new_token_recuerdo
    cookies.permanent[:token_recuerdo] = token_recuerdo
    usuario.update_attribute(:token_recuerdo, Usuario.digest(token_recuerdo))
    self.usuario_actual = usuario
  end

  def usuario_actual=(usuario)
    @usuario_actual = usuario
  end

  def usuario_actual
    token_recuerdo = Usuario.digest(cookies[:token_recuerdo])
    @usuario_actual ||= Usuario.find_by(token_recuerdo: token_recuerdo)
  end
end
```

Se usó `||=` que regresa `@usuario_actual` sin necesidad de tocar la base. Sólo nos sirve esa sintaxis si `usuario_actual` es usado más de una vez para una petición de usuario.

9.2.3. Añadiendo enlaces al layout

Es hora de añadir la funcionalidad a nuestra aplicación añadiendo los enlaces correspondientes al inicio y cierre de sesión. Toca hacer los arreglos necesarios para que los enlaces cambien cuando los usuarios inicien o cierren sesión, también vamos a añadir los enlaces para listar a todos los usuarios y para la página de configuración de cuenta (que completaremos en el capítulo 10) y uno para la página de perfil.

La manera en que vamos a cambiar los enlaces en nuestro sitio, será a través de condicionales *if-else* dentro de código embedido, algo como:

```
<% if inicia_sesion?  %>
      #enlaces para usuarios logueados
<% else %>
      #enlaces para usuarios no logueados
<% end %>
```

Decimos que un usuario ha iniciado sesión si existe un usuario en la sesión actual, es decir, si `usuario_actual` no es nulo. Haremos uso del operador "*not*", escrito con un signo de exclamación que cierra "!".

application_controller.rb

```
class ApplicationController < ActionController::Base
  # Prevent CSRF attacks by raising an exception.
  # For APIs, you may want to use :null_session instead.
  protect_from_forgery with: :exception
  include SesionesHelper
end
```

Definimos el método `inicio_sesion?` que retorna un valor booleano sobre si un usuario ha iniciado sesión o no. Vamos a añadir algunos enlaces, algunos no tendrán funcionalidad hasta el capítulo 10.

```
<%= link_to “Usuarios , ‘#’ %>

<%= link_to “Configuración”, ‘#’ %>

<%= link_to “Perfil”, ‘usuario_actual’ %>
```

Haciendo uso de las ventajas de *Bootstrap* y de la manera en que crea menús desplegables, nuestro nuevo *layout* nos queda como sigue:

_etiquetaHeader.html.erb

```
<header class="navbar navbar-fixed-top navbar-inverse">
  <div class="navbar-inner">
    <div class="container">
      <%= link_to "Mi Aplicación", '#', id: "encabezado" %>
      <nav>
        <ul class="nav pull-right">
          <li><%= link_to "Inicio",    root_path %></li>
          <li><%= link_to "Ayuda",    ayuda_path %></li>
          <% if inicio_sesion? %>
            <li><%= link_to "Usuarios", '#' %></li>
            <li id="fat-menu" class="dropdown">
              <a href="#" class="dropdown-toggle" data-toggle="dropdown">
                Cuenta<b class="caret"></b>
              </a>
              <ul class="dropdown-menu">
                <li><%= link_to "Perfil", usuario_actual %></li>
                <li><%= link_to "Configuración", '#' %></li>
                <li class="divider"></li>
                <li>
                  <%= link_to "Cerrar sesión", cierrases_path, method: "delete" %>
                </li>
              </ul>
              </li>
            <% else%>
            <li><%= link_to "Entrar", '#' %></li>
            <% end %>
          </ul>
        </nav>
      </div>
    </div>
</header>
```

El menú desplegable necesita de la biblioteca *Javascript* para *Bootstrap*, para incluirla lo hacemos desde el *asset pipeline* de *Rails* editando el archivo *JavaScript* de la aplicación como sigue:

```
13  //= require jquery
14  //= require jquery_ujs
15  //= require bootstrap
16  //= require turbolinks
17  //= require_tree .
18
```

Por último, enlazamos la ruta *iniciases_path* con el enlace *Entrar* de nuestra vista. El menú quedaría como en la Figura 9.8:

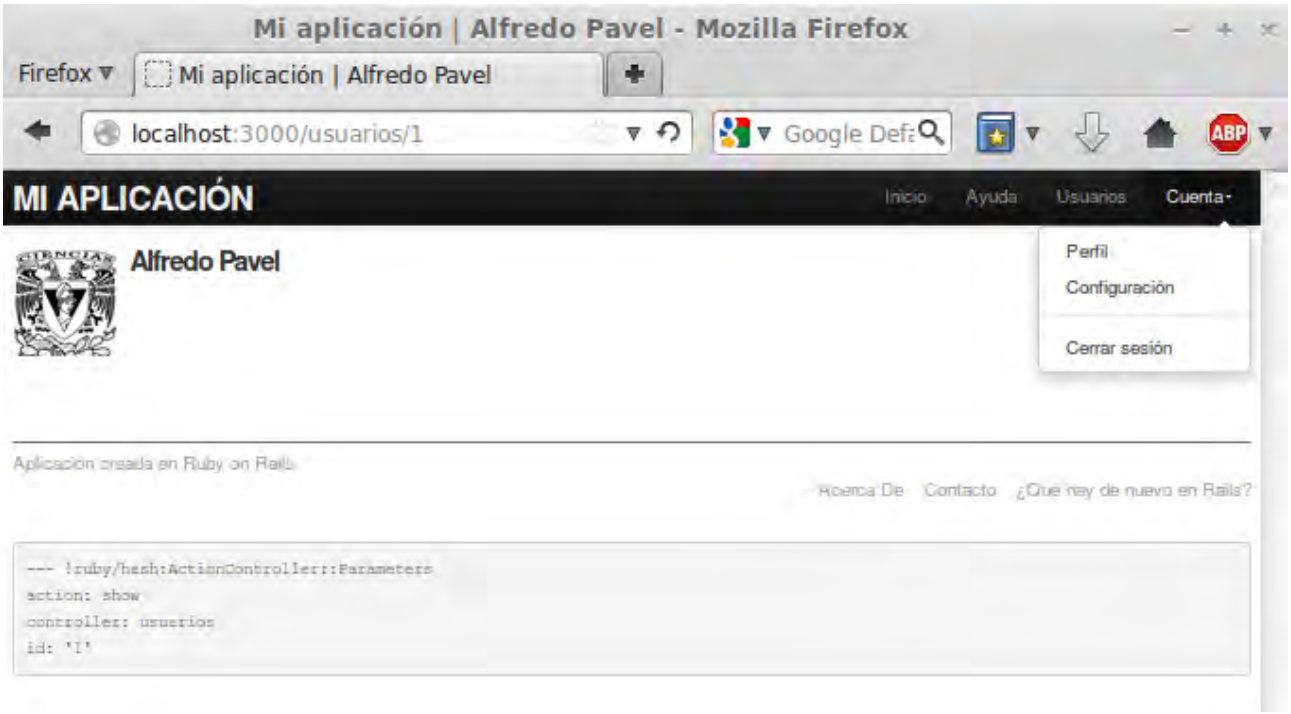


Figura 9.8. Menú desplegable.

Podemos verificar que iniciemos sesión, salir de la página o cerrar el navegador, y ver que seguimos logueados al visitar la página principal de nueva cuenta. Para más detalle, verificamos las *cookies* del navegador:

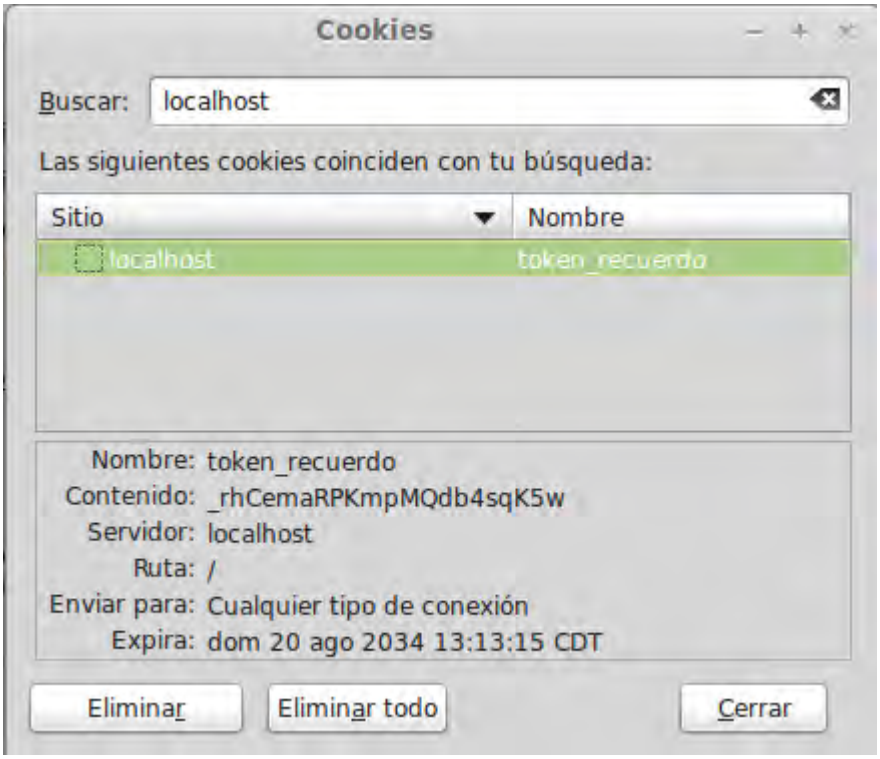


Figura 9.9. Cookie detallada.

9.2.4. Registrar e iniciar sesión

En esta pequeña sección vamos a hacer que un usuario nuevo, al registrarse, inicie sesión inmediatamente ya que si sólo se registra a como tenemos nuestra aplicación, no va a iniciar sesión y puede que esto lo confunda. Este es el último paso para empezar a desarrollar el cierre de sesión.

Sólo debemos de añadir: `inicia_sesion @usuario`, justo después de salvar al usuario en la base de datos (archivo **usuarios_controller.rb**):

```
11
12    def create
13      @usuario = Usuario.new(usuario_params)
14      if @usuario.save
15        inicia_sesion @usuario
16        flash[:success] = "Te has registrado con éxito: #{@usuario.nombre}!"
17        redirect_to @usuario
18      else
19        render 'new'
20      end
21    end
22
23    private
24
25    def usuario_params
```

9.2.5. Cerrar sesión

Para este momento las acciones del controlador *Sesiones* siguen la convención de *REST*, usamos `new` para una página de inicio de sesión y `create` para completar el inicio de sesión. Toca turno a la acción `destroy` para destruir sesiones, o lo que es lo mismo, cerrar sesión.

Tendremos una acción `destroy` que hará uso de una función `cerrar_sesion` como se muestra (**sesiones_controller.rb**):

```
16      end
17
18      def destroy
19        cerrar_sesion
20        redirect_to root_url
21      end
22    end
```

El método `cerrar_sesion` lo vamos a colocar en el *helper Sesiones*. Primero cambiamos el *token* de recuerdo en la base de datos (por si la *cookie* ha sido robada, puede ser usada todavía para autorizar a un usuario), después usamos el método `delete` sobre las *cookies* para quitar el *token* de recuerdo de la sesión; como paso final, al usuario actual le asignamos el valor `nil`.

```
21    end
22
23    def cerrar_sesion
24      usuario_actual.update_attribute(:token_recuerdo,
25                                      Usuario.digest(Usuario.new_token_recuerdo))
26      cookies.delete(:token_recuerdo)
27      self.usuario_actual = nil
28    end
29  end
```

9.3. Guardando cambios

Agregamos los archivos nuevos, hacemos commit, y agregamos los archivos de esta rama a la rama principal como hacemos siempre al final de cada capítulo. Para el siguiente capítulo vamos a restringir el acceso a algunas páginas, basándonos en el estado de sesión y en la identidad del usuario. Vamos a otorgar la habilidad a los usuarios de modificar su información y daremos a los administradores la habilidad de remover usuarios del sistema.

```
$ git add .

$ git commit -m 'Inicio y cierre de sesión finalizado'

$ git checkout master

$ git merge iniciar_cerrar_sesion

$ git push origin master
```

Capítulo 10: Modificación, Muestra y Eliminación de Usuarios

10. Desarrollo

Como se dijo antes, vamos a empezar con el desarrollo de editar la información de los usuarios para luego ir añadiendo más funciones como eliminar usuarios. Creamos la rama apropiada para este capítulo:

```
$ git checkout master

$ git checkout -b edicion_usuarios
```

10.1. Editar usuarios, primer paso

La forma en qué se va a editar la información de los usuarios es parecida a cuando creamos nuevos usuarios. En lugar de tener una acción `new` que genera una vista para los nuevos usuarios, tendremos una acción `edit` que genera una vista para la edición de los mismos; en lugar de que `create` responda a una petición *POST*, tendremos una acción `update` que responde a una petición *PATCH*. La principal diferencia radica en que cualquiera puede registrarse pero sólo los usuarios que se hayan logueado podrán editar sus datos.

Empezamos con la creación de la maqueta que nos ayudará a modelar nuestra vista para la edición de usuarios que la vemos en la Figura 10.1. Notemos que el enlace que tenemos para cambiar la imagen de *Gravatar*, apunta a la url http://gravatar.com/emails que es la que se encarga de editar o añadir las imágenes en el sitio.

Figura 10.1. Maqueta de edición de perfil.

Vamos a añadir el código a la acción `edit` del controlador `Usuarios`. Recordemos que el *id* de los usuarios están disponible en la variable `params[:id]`, con el que podemos encontrar a un usuario con el código siguiente:

```
23    def edit
24      @usuario = Usuario.find(params[:id])
25    end
26
27    private
28
29    def usuario params
```

La vista para editar la información de los usuarios es muy parecida a la vista de nuevos usuarios. Hacemos uso también del parcial *mensajes_error* como se ve en el Código 10.1 (visto en el capítulo 8):

app/views/usuarios/edit.html.erb

edit.html.erb

```
<% provide(:titulo, "Editar usuario") %>
<h1>Actualiza tu perfil</h1>

<div class="row">
  <div class="span6 offset3">
    <%= form_for(@usuario) do |atributo| %>
      <%= render 'shared/mensajes_error' %>

      <%= atributo.label :nombre %>
      <%= atributo.text_field :nombre %>

      <%= atributo.label :email %>
      <%= atributo.text_field :email %>

      <%= atributo.label :password %>
      <%= atributo.password_field :password %>

      <%= atributo.label :password_confirmation, "Confirmación" %>
      <%= atributo.password_field :password_confirmation %>

      <%= atributo.submit "Guardar cambios", class: "btn btn-large btn-primary" %>
    <% end %>

    <%= gravatar_de @usuario %>
    <a href="http://gravatar.com/emails">Cambiar imagen</a>
  </div>
</div>
```

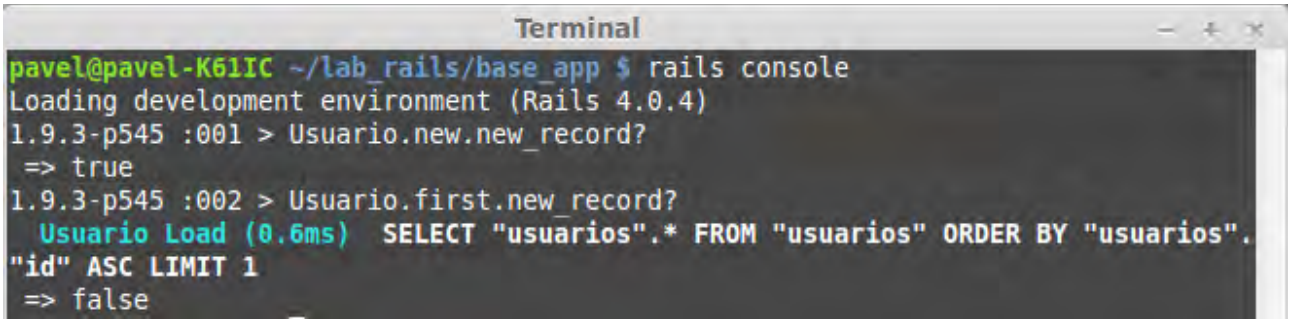
Código 10.1. **edit.html.erb**

El resultado de lo anterior se ve en la Figura 10.2, observamos como *Rails* automáticamente rellena los campos *nombre* y *email*, usando los atributos que otorga la variable `@usuario`:

Figura 10.2. Formulario de actualización de perfil.

Si comparamos el código `form_for(@usuario)` para un nuevo usuario con el código para la edición de la información, vemos que es exactamente el mismo, lo que nos lleva a plantearnos ¿cómo es que *Rails* puede identificar y usar una petición *POST* para nuevos usuarios y una petición *PATCH* para la edición de los mismos? La respuesta está en el método `new_record?` de *Active Record* que nos retorna un valor booleano:

```
$ rails console

>> Usuario.new.new_record?

>> Usuario.first.new_record?
```



Cuando construimos un formulario con `form_for(@usuario)`, *Rails* utiliza la petición *POST* si `@usuario.new_record?` es verdadero y *PATCH* si es falso.

Ahora, vamos a añadir el enlace a la página de configuración de usuario, que nos resulta muy sencillo; sólo debemos de usar la ruta personalizada `edit_usuario_path` (visto en la Tabla 8.1), junto con el *helper usuario_actual*:

_etiquetaHeader.html.erb

```
<header class="navbar navbar-fixed-top navbar-inverse">
  <div class="navbar-inner">
    <div class="container">
      <%= link_to "Mi Aplicación", root_path, id: "encabezado" %>
      <nav>
        <ul class="nav pull-right">
          <li><%= link_to "Inicio",    root_path %></li>
          <li><%= link_to "Ayuda",    ayuda_path %></li>
          <% if inicio_sesion? %>
            <li><%= link_to "Usuarios", '#' %></li>
            <li id="fat-menu" class="dropdown">
              <a href="#" class="dropdown-toggle" data-toggle="dropdown">
                Cuenta<b class="caret"></b>
              </a>
              <ul class="dropdown-menu">
                <li><%= link_to "Perfil", usuario_actual %></li>
                <li><%= link_to "Configuración", edit_usuario_path(usuario_actual) %></li>
                <li class="divider"></li>
                <li>
                  <%= link_to "Cerrar sesión", cierrases_path, method: "delete" %>
                </li>
              </ul>
              </li>
          <% else %>
          <li><%= link_to "Entrar", iniciases_path %></li>
          <% end %>
        </ul>
      </nav>
    </div>
  </div>
</header>
```

10.1.2. Problemas al editar usuarios

En esta sección trataremos los cambios de información del perfil no exitosos. Crearemos una acción `update` que utilizará el método `update_attributes` para actualizar a los usuarios, basándose en el contenido de la variable `params`. Si la actualización falla por información errónea en el formulario, el intento de actualizar retorna un falso, así lo que haremos será recargar la página de edición.

```
  def edit
    @usuario = Usuario.find(params[:id])
  end

  def update
    @usuario = Usuario.find(params[:id])
    if @usuario.update_attributes(usuario_params)
      #ejecutar actualizacion satisfactoria
    else
      render 'edit'
    end
  end
```

Como resultado de tener errores en el formulario, la página correspondiente se muestra en la Figura 10.3:

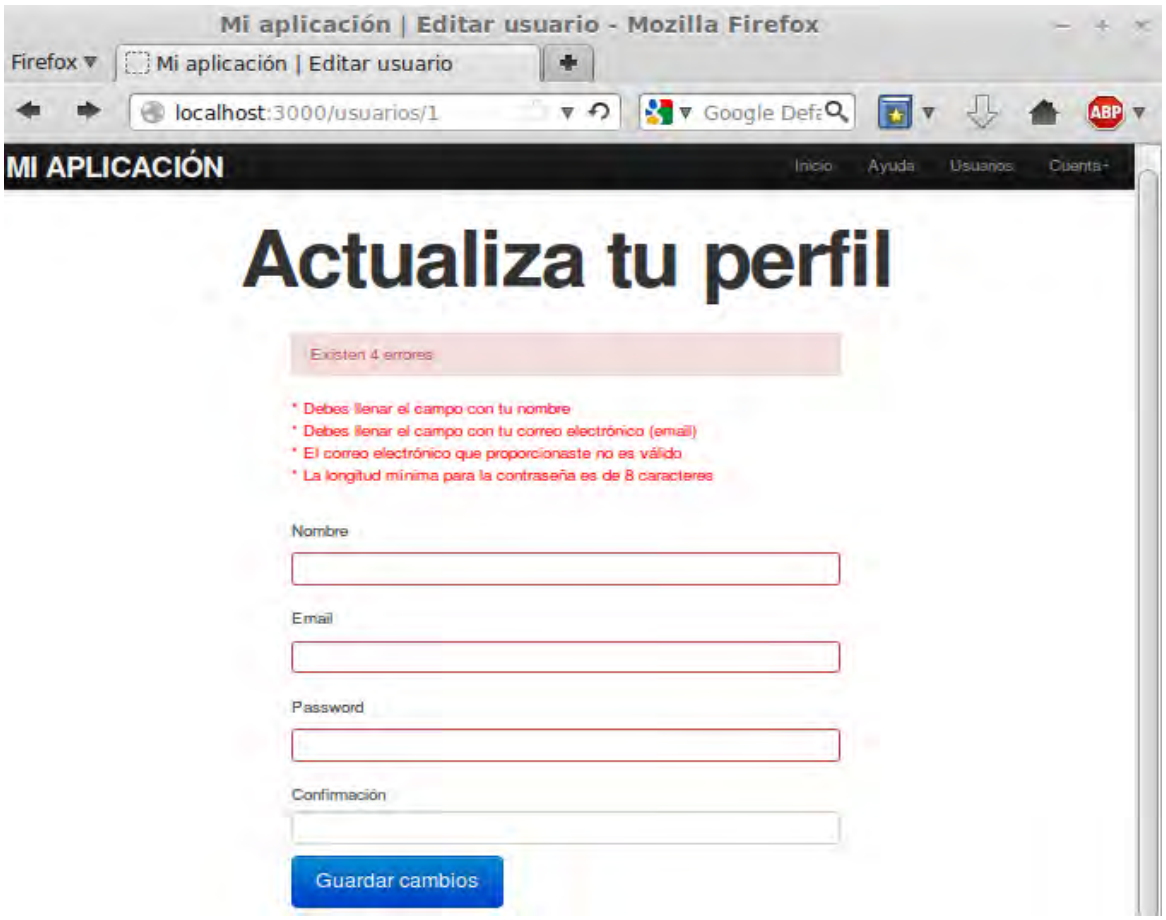


Figura 10.3. Errores en edición.

10.1.3. Edición exitosa

Toca turno a que nuestro formulario funcione (previa introducción de información válida). Por el momento la parte que funciona es la edición de la imagen de perfil, usando el enlace a *Gravatar*; sólo basta con dar click en el botón "editar" y se nos redirige a la página apropiada, en donde podremos subir una imagen y modificar su tamaño (Figura 10.4).

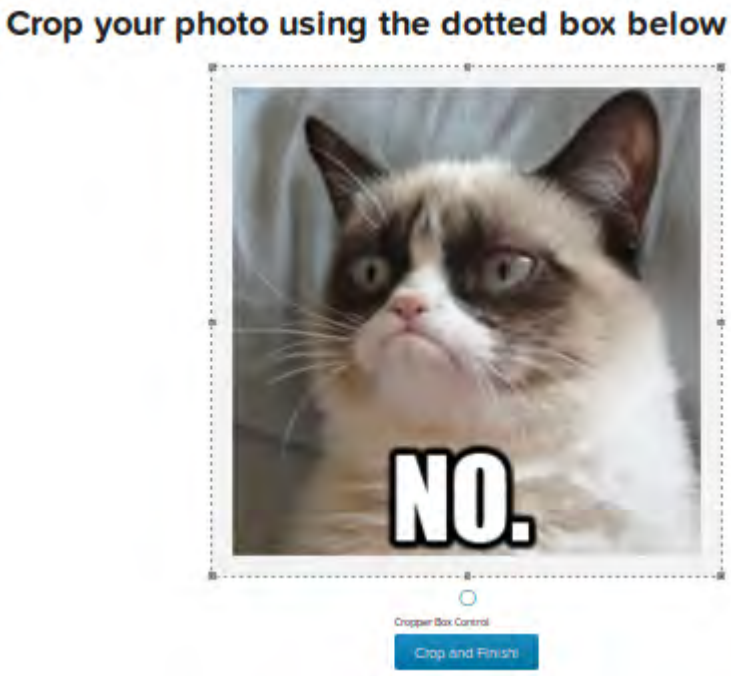


Figura 10.4. Cambio imagen con *Gravatar*.

Toca crear la acción `update` para la confirmación de cambios, que es muy similar a la acción `create` que notifica que un usuario se creó exitosamente. El código nos queda de la siguiente forma:

```
26
27    def update
28      @usuario = Usuario.find(params[:id])
29      if @usuario.update_attributes(usuario_params)
30        flash[:success] = "Perfil actualizado"
31        redirect_to @usuario
32      else
33        render 'edit'
34      end
35    end
```

10.2. Autorizaciones

En este momento nos toca implementar la autorización: autenticar que nos permite identificar usuarios de nuestro sitio, la autorización nos va a permitir tener control sobre lo que pueden hacer en el sitio.

Tanto la acción de edición, como la de actualizar son funcionales y completas pero dejan que cualquiera (incluso usuarios no logueados) pueda acceder a las acciones, y cualquier usuario logueado puede actualizar la información de cualquier otro usuario. En esta sección, vamos a implementar un pequeño modelo de seguridad que requiere la autenticación de usuarios y los previene que puedan actualizar información de otros

usuarios. Los usuarios que no inicien sesión, serán redirigidos a la página de registro con un mensaje de ayuda.

10.2.1. Usuarios con sesión

Tanto la seguridad de restricción en la acción `edit` y `update`, es exactamente la misma. Para esto vamos a usar el método `before_action` para tratar con algún método en particular, antes de ejecutar una acción. Para pedir que los usuarios estén logueados, definimos el método `usuario_logueado` e invocarlo usando: `before_action :usuario_logueado`:

usuarios_controller.rb

```
1  #encoding: UTF-8
2  class UsuariosController < ApplicationController
3    before_action :usuario_logueado, only: [:edit, :update]
4
5    def show
6      @usuario = Usuario.find(params[:id])
7    end
8
```

```
37   private
38
39   def usuario_params
40     params.require(:usuario).permit(:nombre, :email, :password, :password_confirmation)
41   end
42
43   #filtros
44
45   def usuario_logueado
46     redirect_to iniciases_url, notice: "Por favor, inicia sesión" unless inicio_sesion?
47   end
48 end
```

Por defecto, los filtros se aplican a todas las acciones de un controlador, pero en este caso y hemos restringido el filtro a sólo la acción `edit` y `update` a través de `:edit`, `:update` como parámetros al *hash* `:only`. También usamos un atajo para el mensaje `flash[:notice]` pasando un *hash* de opciones a la función `redirect_to` (esto no funciona con las claves :`error` o :`success`).

Probamos que todo funcione y muestre el mensaje (Figura 10.5):

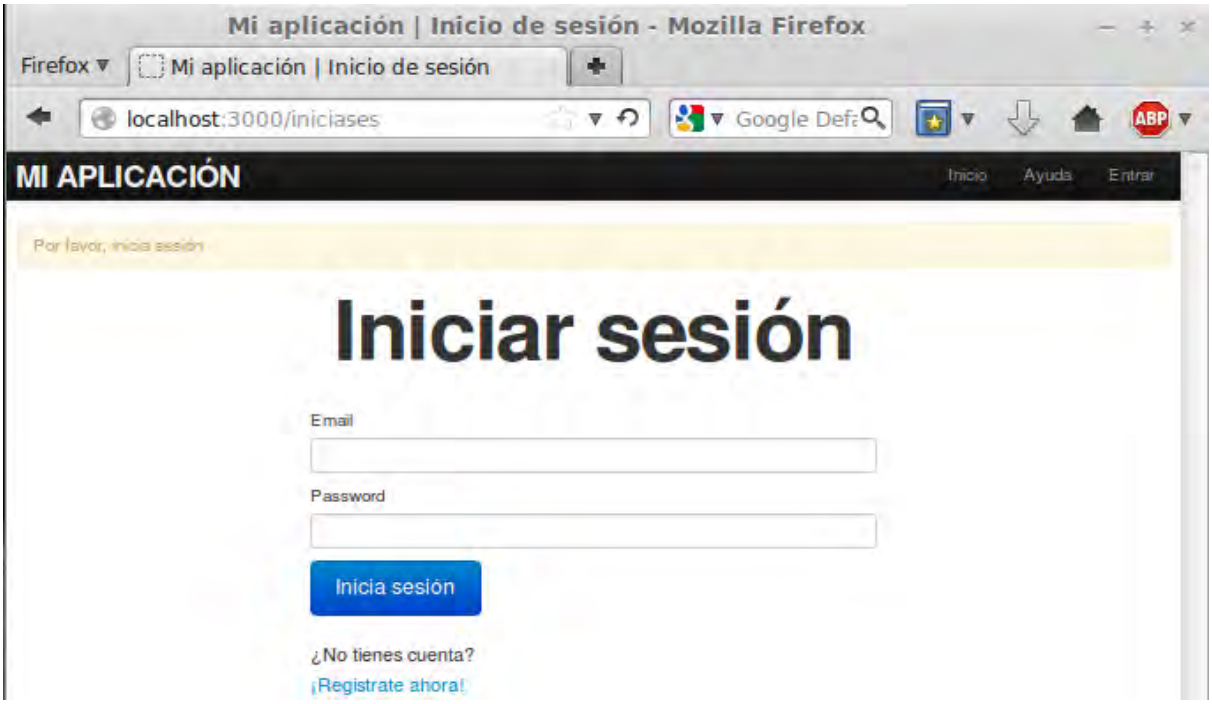


Figura 10.5. Prueba de *flash*.

10.2.2. Usuarios válidos

No basta con pedir a los usuarios que inicien sesión, los usuarios sólo deben modificar su propia información. Vamos a añadir un segundo filtro de tipo *before* para llamar al método `usuario_valido` como sigue:

usuarios_controller.rb

```
1  #encoding: UTF-8
2  class UsuariosController < ApplicationController
3    before_action :usuario_logueado, only: [:edit, :update]
4    before_action :usuario_valido,   only: [:edit, :update]
5
6    def show
```

```
50
51    def usuario_valido
52      @usuario = Usuario.find(params[:id])
53      redirect_to(root_url) unless usuario_actual?(@usuario)
54    end
55  end
56
```

El método `usuario_valido` va a usar al método booleano `usuario_actual?` que vamos a definir en el *helper Sesiones:*

```
18    def usuario_actual
19      token_recuerdo = Usuario.digest(cookies[:token_recuerdo])
20      @usuario_actual ||= Usuario.find_by(token_recuerdo: token_recuerdo)
21    end
22
23    def usuario_actual?(usuario)
24      usuario == usuario_actual
25    end
```

En el código de **usuarios_controller.rb**, muestra las acciones correspondientes a `edit` y `update`. Ahora usuario_valido define a `@usuario` y se puede omitir de ambas acciones.

10.2.3. Seguimiento de páginas

El sitio de autorización está completo pero existe un pequeño problema: cuando los usuarios intenten acceder a una de las páginas protegidas, van a ser redirigidos a su página de perfil sin importar el enlace que querían consultar. Más sencillo,si un usuario que no ha iniciado sesión trata de consultar la página de edición,después de iniciar sesión será redirigido a su perfil y no a la página de edición. Sería mejor que se le envié directamente a su página de edición.

Para dirigir a los usuarios a la página que desean, necesitamos guardar la ubicación de la página que se pide en algún lugar y luego redirigirlos a esa ubicación. Lo haremos con los métodos, `guardar_ubicacion` y `redirige_antes` dentro del *helper Sesiones*:

```
28      usuario_actual.update_attribute(:token_recuerdo,
29                                      Usuario.digest(Usuario.ne
30      cookies.delete(:token_recuerdo)
31      self.usuario_actual = nil
32    end
33
34    def redirige_antes(default)
35      redirect_to(session[:anterior_a] || default)
36      session.delete(:anterior_a)
37    end
38
39    def guardar_ubicacion
40      session[:anterior_a] = request.url if request.get?
41    end
42  end
```

Para almacenar la ubicación,haremos uso de `session` que nos proporciona *Rails*, también vamos a utilizar al objeto `request` para obtener la *URL* de la página solicitada. El método `guardar_ubicacion` agrega la *URL* solicitada en `session` con la clave `anterior_a`, pero sólo para una petición de tipo *`GET`*. Lo anterior previene que guardemos la *URL* solicitada, si un usuario, por ejemplo, envía un formulario cuando no está logueado; en ese caso, el redireccionamiento dará lugar a un error en el que una *URL* que espera una petición *POST*, *PATCH* o *DELETE*, recibe una petición *GET*.

Para hacer uso del método `guardar_ubicacion`, debemos añadirlo al filtro `usuario_logueado`:

```
45    #filtros
46
47    def usuario_logueado
48     unless inicio_sesion?
49       guardar_ubicacion
50       redirect_to iniciases_url, notice: "Por favor, inicia sesión"
51     end
52    end
```

Para el seguimiento de una página a sí misma, usamos el método `dirige_antes` para enviar al usuario ala *URL* solicitada si esta existe, o en otro caso, a una *URL* por defecto, la cual añadimos a la acción `create` del controlador `Sesiones`, a la que redirigir cuando un inicio de sesión es exitoso.

sesiones_controller.rb

```
1  #encoding: utf-8
2  class SesionesController < ApplicationController
3
4      def new
5      end
6
7      def create
8          usuario = Usuario.find_by(email: params[:sesion][:email].downcase)
9          if usuario && usuario.authenticate(params[:sesion][:password])
10             inicia_sesion usuario
11             redirige_antes usuario
12         else
13             flash.now[:error] = "Revisa que tu email y contraseña sean correctos"
14             render 'new'
15         end
16     end
17
18     def destroy
```

Ahora nuestra básica implementación de autenticación de usuarios y protección de página está completa.

10.3. Mostrando a los usuarios del sitio

En este apartado añadimos la acción `index` para los usuarios, la cual se encargará de mostrar a todos los usuarios del sitio en lugar de sólo uno. Veremos cómo llenar la base de datos con usuarios falsos mostrando un listado con “número de páginas”, así, al tener muchos usuarios podemos escalarlos para mostrarlos en pantalla. La maqueta queda como en la Figura 10.6:

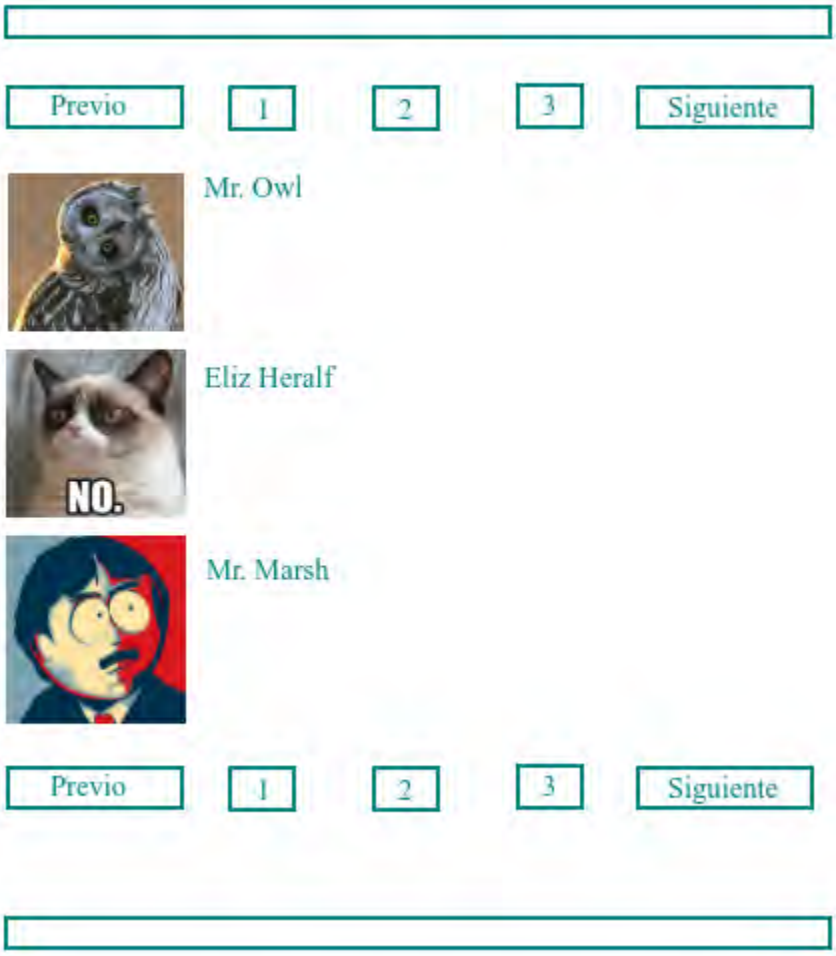


Figura 10.6. Maqueta de usuarios.

10.3.1. Índice de usuarios

Aunque mantenemos las páginas de tipo `show` de los usuarios (de manera individual) visibles a todos los visitantes de nuestra página, el índice de usuario debe ser restringido a usuarios que hayan iniciado sesión, así que habrá un límite de cuántos usuarios no registrados pueden ver por default.

Añadimos el índice `index` a la lista de acciones protegidas bajo el filtro `usuario_logueado`, luego para mostrar a todos los usuarios tal cual, utilizamos `Usuario.all`, asignándolos a la variable de instancia `@usuarios` para mostrarlos en la vista.

usuarios_controller.rb

```ruby
#encoding: UTF-8
class UsuariosController < ApplicationController
  before_action :usuario_logueado, only: [:index, :edit, :update]
  before_action :usuario_valido,   only: [:edit, :update]

  def index
    @usuario = Usuario.all
  end

  def show
    @usuario = Usuario.find(params[:id])
  end
```

Para realizar el índice de las páginas de usuarios, necesitamos hacer una vista (**index.html.erb**, Código 10.1) que itere a través de los usuarios y envuelva a cada uno en una etiqueta de lista `li`. Para realizar lo anterior usamos el método `each`, mostrando tanto la imagen de perfil y el nombre, mientras envolvemos todo en una lista no ordenada *HTML* (etiqueta `ul`).

app/views/usuarios/index.html.erb

index.html.erb

```erb
<% provide(:titulo, 'Todos los usuarios') %>
<h1>Listado de usuarios</h1>

<ul class="listaUsuarios">
  <% @usuarios.each do |usuario| %>
    <li>
      <%= gravatar_de usuario %>
      <%= link_to usuario.nombre, usuario %>
    </li>
  <% end %>
</ul>
```

Código 10.2. **index.html.erb**

Añadimos un poco de estilo en el archivo **estilo.css.scss**:

```scss
/*Estilo para listado de usuarios*/

.listaUsuarios {
  list-style: none;
  margin: 0;
  li {
    overflow: auto;
    padding: 5px 0;
    border-top: 1px solid $color_gris;
    &:last-child {
      border-bottom: 1px solid $color_gris;
    }
  }
}
```

Por último, añadimos la *URL* a los enlaces de usuario en el encabezado de navegación del sitio con la ruta usuarios_path, así todas las rutas personalizadas estarán usándose.

_etiquetaHeader.html.erb

```
<header class="navbar navbar-fixed-top navbar-inverse">
  <div class="navbar-inner">
    <div class="container">
      <%= link_to "Mi Aplicación", root_path, id: "encabezado" %>
      <nav>
        <ul class="nav pull-right">
          <li><%= link_to "Inicio",    root_path %></li>
          <li><%= link_to "Ayuda",    ayuda_path %></li>
          <% if inicio_sesion? %>
            <li><%= link_to "Usuarios", usuarios_path %></li>
            <li id="fat-menu" class="dropdown">
              <a href="#" class="dropdown-toggle" data-toggle="dropdown">
                Cuenta<b class="caret"></b>
              </a>
               <ul class="dropdown-menu">
                <li><%= link_to "Perfil", usuario_actual %></li>
                <li><%= link_to "Configuración", edit_usuario_path(usuario_actual) %></li>
                <li class="divider"></li>
                <li>
                  <%= link_to "Cerrar sesión", cierrases_path, method: "delete" %>
                </li>
              </ul>
              </li>
          <% else %>
          <li><%= link_to "Entrar", iniciases_path %></li>
          <% end %>
        </ul>
      </nav>
    </div>
  </div>
</header>
```

El resultado se muestra en la Figura 10.7:

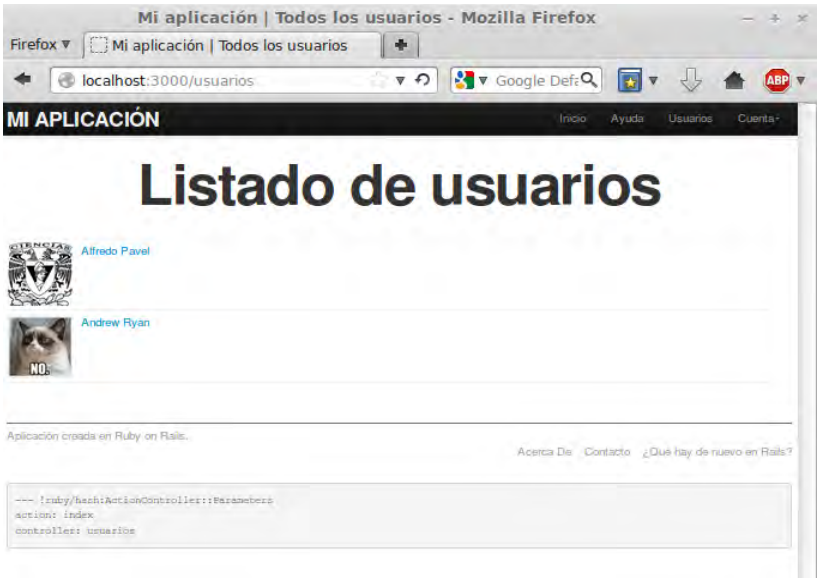


Figura 10.7. Listado de usuarios.

10.3.2. Usuarios de ejemplo

Como se vio en la Figura 10.7, nuestro usuario está muy solo, vamos a darle un poco de compañía. Podríamos añadir los usuarios manualmente a través de la página pero resultaría muy pesado al ser muchos usuarios, nos conviene en este caso utilizar *Ruby* para que haga los usuarios por nosotros.

Añadimos la gema *Faker* a nuestro archivo **Gemfile**, que nos permite hacer usuarios de ejemplo con nombres semi realistas y con direcciones de correo electrónico. Después lo instalamos con `bundle install` como siempre:

Gemfile

```
1  source 'https://rubygems.org'
2
3  # Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
4  gem 'rails', '4.0.4'
5
6  # Gema para bcrypt
7  gem 'bcrypt-ruby', '3.1.2'
8
9  # Gema para Faker
10 gem 'faker', '1.1.2'
11
```

Luego creamos una tarea a *Rake* para crear a los usuarios de ejemplo. Las tareas de *Rake* se encuentran en el directorio lib/tasks, definimos la tarea en el archivo **datos_ejemplo**, Código 10.3:

lib/tasks/datos_ejemplo.rake

datos_ejemplo.rake

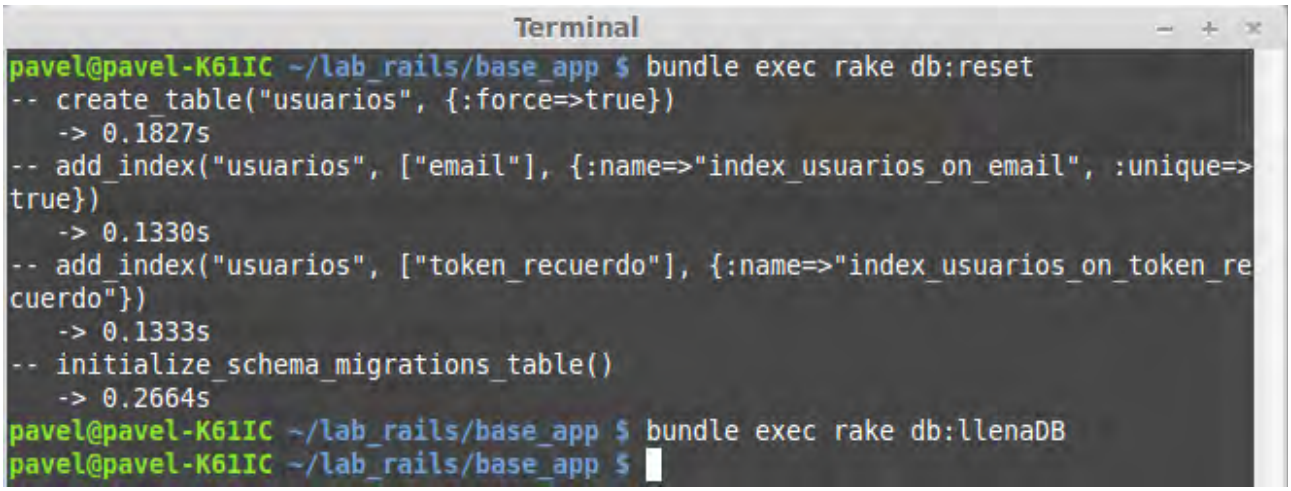
```
1  namespace :db do
2    desc "Llenamos la base"
3    task llenaDB: :environment do
4      Usuario.create!(nombre: "Alfredo Pavel",
5                      email: "laboratorio.rubyrails@gmail.com",
6                      password: "testing00",
7                      password_confirmation: "testing00")
8      99.times do |n|
9        nombre  = Faker::Name.name
10       email = "usuario_test#{n+1}@ejemplo.org"
11       password  = "testing#{n+1}"
12       Usuario.create!(nombre: nombre,
13                       email: email,
14                       password: password,
15                       password_confirmation: password)
16     end
17   end
18 end
```

Código 10.3. **datos_ejemplo.rake**

Lo anterior define una tarea `db:llenaDB` que genera a los usuarios ejemplo con un nombre y un email, replicando al anterior 99 veces. La línea `task llenaDB: :environment do`, se asegura de que las tareas de *Rake* tengan acceso al ambiente local de *Rails*, esto incluye al modelo *Usuario*.

Invocamos la tarea *Rake* con los siguientes comandos:

```
$ bundle exec rake db:reset

$ bundle exec rake db:llenaDB
```



Después de ejecutar la tarea, la aplicación tendrá ahora 100 usuarios de ejemplo (Figura 10.8):

Figura 10.8. Lista de Usuarios.

10.3.3. Paginación

Se nos presenta otro problema, nuestro usuario ya no se encuentra tan solo pero ahora la vista muestra a todos los usuarios en la misma página. La solución a esto es paginar los usuarios, podríamos mostrar sólo a 30 usuarios por página a la vez (por poner un ejemplo).

Aunque existen diferente métodos de paginación en *Rails*, usaremos el más simple y a la vez el más robusto llamado, `will_paginate`. Como con *Faker*, debemos añadir la *gema will_paginate* y *bootstrap-will_paginate*, que configura el método `will_paginate` para usar los estilos de paginación de *Bootstrap*.

Gemfile

```
1  source 'https://rubygems.org'
2
3  # Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
4  gem 'rails', '4.0.4'
5
6  # Gema para bcrypt
7  gem 'bcrypt-ruby', '3.1.2'
8
9  # Gema para Faker
10 gem 'faker', '1.1.2'
11
12 # Gemas necesarias para paginar
13 gem 'will_paginate', '3.0.4'
14 gem 'bootstrap-will_paginate', '0.0.9'
15
```

Instalamos:

```
$ bundle install
```

Para que funciones la paginación, debemos añadir algo de código a la vista *índex* que especifique a *Rails* paginar a los usuarios, también necesitamos reemplazar `Usuarios.all` en la acción `index` con un objeto que reconozca la paginación. Empezamos añadiendo el método `will_paginate` en la vista.

index.html.erb

```
1  <% provide(:titulo, 'Todos los usuarios') %>
2  <h1>Listado de usuarios</h1>
3
4  <%= will_paginate %>
5
6  <ul class="listaUsuarios">
7    <% @usuarios.each do |usuario| %>
8      <li>
9        <%= gravatar_de usuario %>
10       <%= link_to usuario.nombre, usuario %>
11     </li>
12   <% end %>
13 </ul>
14
15 <%= will_paginate %>
```

Dentro de la vista usuarios, el método `will_paginate` busca automáticamente por un objeto `@usuarios` y luego muestra los enlaces de paginación para acceder a las otras páginas. La vista anterior todavía no funciona, esto es porque `@usuarios` contiene el resultado de `Usuarios.all`, en su lugar `will_paginates` requiere que nosotros paginemos los resultados explícitamente usando el método `paginate`:

```
$ rails console

>> Usuario.paginate(page: 1)
```

```
1.9.3-p545 :010 > Usuario.paginate(page: 1)
  Usuario Load (0.4ms)  SELECT "usuarios".* FROM "usuarios" LIMIT 30 OFFSET 0
   (0.2ms)  SELECT COUNT(*) FROM "usuarios"
 => #<ActiveRecord::Relation [#<Usuario id: 1, nombre: "Alfredo Pavel", email: "
laboratorio.rubyrails@gmail.com", created_at: "2014-09-28 23:41:32", updated_at:
 "2014-09-28 23:49:53", password_digest: "$2a$10$TlTEhE7.tolQe73HudDmq.OJOhKjff4
YeFKYbPYmbn07...", token_recuerdo: "eca1663c7b52355e82606a971954124b9d98efa5">,
#<Usuario id: 2, nombre: "Susana Skiles II", email: "usuario_test1@ejemplo.org",
 created_at: "2014-09-28 23:41:32", updated_at: "2014-09-28 23:41:32", password_
digest: "$2a$10$EZkzhIx2VoLiRrYvuyUNPuqodGbzwD6LYGZtth9wv.v....", token_recuerdo
: nil>, #<Usuario id: 3, nombre: "Jayme Upton V", email: "usuario_test2@ejemplo
```

Vemos que `paginate` toma un argumento de un *hash* con llave `:page` y el valor igual a la página que solicitamos (por defecto retorna 30 usuarios por página, si es `nil`, regresa la primer página). Usando el método `paginate`, podemos paginar a los usuarios de nuestra aplicación usando `paginate` en lugar de `all` en la acción `index`. El parámetro `:pagina` es obtenido de `params[:page]`, que es generado automáticamente por `will_paginate`.

usuarios_controller.rb

```
1  #encoding: UTF-8
2  class UsuariosController < ApplicationController
3    before_action :usuario_logueado, only: [:index, :edit, :update]
4    before_action :usuario_valido,   only: [:edit, :update]
5
6    def index
7      @usuarios = Usuario.paginate(page: params[:page])
8    end
9
10   def show
```

Ahora sí, la página *index* ya debe de estar trabajando (Figura 10.9). Dado que agregamos a `will_paginate` arriba y debajo de la lista de usuarios, los enlaces de paginación aparecen en ambos lugares.

Figura 10.9. Paginación de Usuarios.

Probemos dando click al enlace 2 o *next* para visualizar los resultados siguientes como se muestra en la Figura 10.10:

Figura 10.10. Segunda página de usuarios.

10.3.4. Simplificando páginas

Rails cuenta con una herramienta para hacer a las vistas más simples o compactas, que nos sirve de mucho para la página *index*. Podemos hacer la simplificación sin miedo a perder la funcionalidad del sitio ya que hemos estado verificando el estado del mismo.

Primero reemplacemos la etiqueta `li` y su contenido con un llamada a `render` en la vista *index*.

index.html.erb

```
<% provide(:titulo, 'Todos los usuarios') %>
<h1>Listado de usuarios</h1>

<%= will_paginate %>

<ul class="listaUsuarios">
  <% @usuarios.each do |usuario| %>
    <%= render usuario %>
  <% end %>
</ul>

<%= will_paginate %>
```

Hacemos un render pero no sobre una cadena con el nombre del parcial, si no a la variable usuario (definida en el ciclo `each`) de la clase Usuario; en este contexto, *Rails* se encarga de buscar por un parcial llamado *_usuario.html.erb*, el cual creamos a continuación:

app/views/usuarios/_usuario.html.erb

_usuario.html.erb

```
<li>
      <%= gravatar_de usuario %>
      <%= link_to usuario.nombre, usuario %>
</li>
```

Código 10.4. **_usuario.html.erb**

Podemos hacerlo aún más sencillo haciendo un render directamente sobre la variable `@usuarios`.

index.html.erb

```
1
2  <% provide(:titulo, 'Todos los usuarios') %>
3  <h1>Listado de usuarios</h1>
4
5  <%= will_paginate %>
6
7  <ul class="listaUsuarios">
8      <%= render @usuarios %>
9  </ul>
10
11 <%= will_paginate %>
```

Rails infiere que `@usuarios` es una lista de objetos de tipo `Usuario`; más aún, cuando hacemos la llamada con una colección de usuarios, *Rails* itera a través de ellos y hace el render de cada uno con el parcial **_usuario.html.erb.**

10.4. Eliminado usuarios

En este punto nuestro índice de usuarios está casi completo pero nos falta completar una acción de *REST*, la acción `destroy`. En este apartado vamos a añadir los enlaces para eliminar usuarios, como se muestra en la Figura 10.11, definiendo la acción destroy para completar el borrado. Antes, debemos crear la clase de usuarios administradores que estarán autorizados para eliminar.

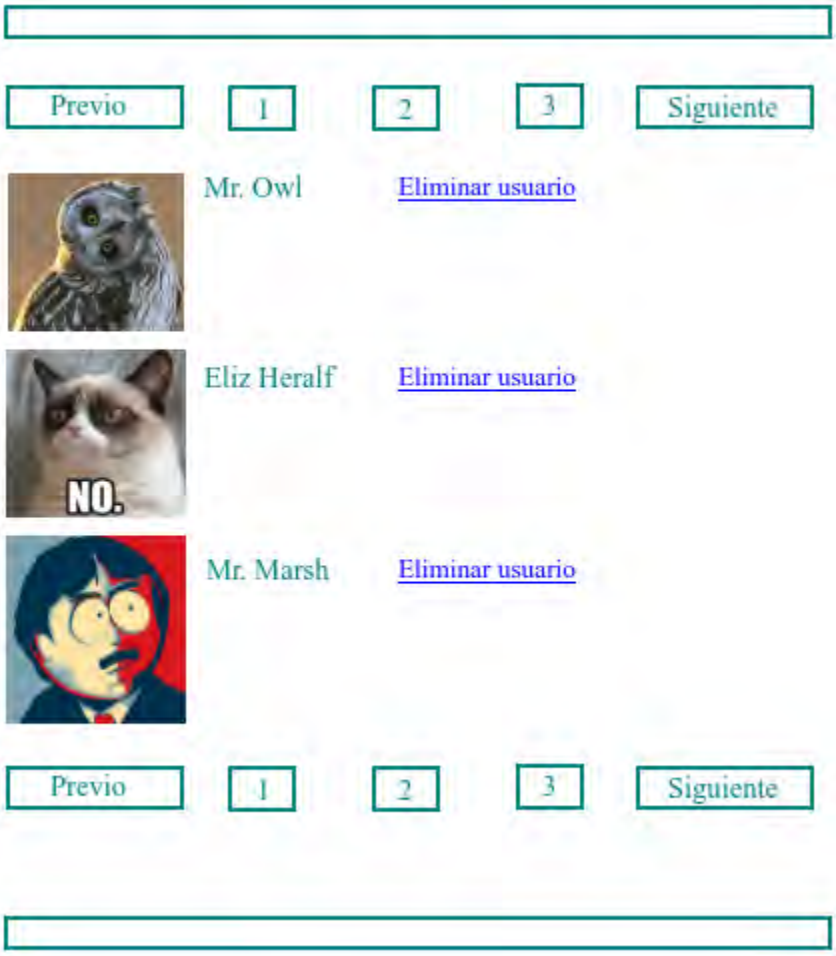


Figura 10.11. Maqueta para eliminar usuarios.

10.4.1. Administradores

Distinguimos a los usuarios que son administradores con el atributo booleano `admin` en el modelo `Usuario`, el cual veremos nos llevará hacia un método booleano `admin?` para probar el estado de administrador.

Añadimos el atributo *admin* a través de una migración, indicando el tipo booleano en la consola de comandos:

```
$ rails generate migration add_admin_to_usuarios
admin:boolean
```

20140929011744_add_admin_to.rb

```
1  class AddAdminTo < ActiveRecord::Migration
2    def change
3      add_column :usuarios, :admin, :boolean, default: false
4    end
5  end
6
```

Veamos que añadimos el argumento `default: false` a `add_column` en la migración de arriba, lo que significa que los usuarios no serán administrativos por default. Ahora migramos la base:

```
$ bundle exec rake db:migrate
```

Como habíamos dicho, *Rails* infiere la naturaleza booleana del atributo *admin* y automáticamente añade el método `admin?`

```
$ rails console --sandbox

>> usuario = Usuario.first

>> usuario.admin?

>> usuario.toggle!(:admin)

>> usuario.admin?
```

Terminal

```
pavel@pavel-K61IC ~/lab_rails/base_app $ rails console --sandbox
Loading development environment in sandbox (Rails 4.0.4)
Any modifications you make will be rolled back on exit
1.9.3-p545 :001 > usuario = Usuario.first
  Usuario Load (0.2ms)  SELECT "usuarios".* FROM "usuarios" ORDER BY "usuarios".
"id" ASC LIMIT 1
 => #<Usuario id: 1, nombre: "Alfredo Pavel", email: "laboratorio.rubyrails@gmai
l.com", created_at: "2014-09-28 23:41:32", updated_at: "2014-09-28 23:49:53", pa
ssword_digest: "$2a$10$TlTEhE7.tolQe73HudDmq.OJOhKjff4YeFKYbPYmbn07...", token_r
ecuerdo: "eca1663c7b52355e82606a971954124b9d98efa5", admin: false>
1.9.3-p545 :002 > usuario.admin?
 => false
1.9.3-p545 :003 > usuario.toggle!(:admin)
   (0.2ms)  SAVEPOINT active_record_1
  SQL (38.5ms)  UPDATE "usuarios" SET "admin" = ?, "updated_at" = ? WHERE "usuar
ios"."id" = 1  [["admin", true], ["updated_at", Mon, 29 Sep 2014 01:20:53 UTC +0
0:00]]
   (0.1ms)  RELEASE SAVEPOINT active_record_1
 => true
1.9.3-p545 :004 > usuario.admin?
 => true
```

Actualizamos nuestros datos de ejemplo que llenan la base para que el primero sea administrador por default:

datos_ejemplo.rake

```
namespace :db do
  desc "Llenamos la base"
  task llenaDB: :environment do
    Usuario.create!(nombre: "Alfredo Pavel",
                    email: "laboratorio.rubyrails@gmail.com",
                    password: "testing00",
                    password_confirmation: "testing00",
                    admin: true)
    99.times do |n|
      nombre  = Faker::Name.name
```

Reiniciamos la base y rellenamos de nuevo.

```
$ bundle exec rake db:reset

$ bundle exec rake db:llenaDB
```

datos_ejemplo.rake

```
namespace :db do
  desc "Llenamos la base"
  task llenaDB: :environment do
    Usuario.create!(nombre: "Alfredo Pavel",
                    email: "laboratorio.rubyrails@gmail.com",
                    password: "testing00",
                    password_confirmation: "testing00",
                    admin: true)
    99.times do |n|
      nombre  = Faker::Name.name
```

10.4.2. Completando la acción destroy

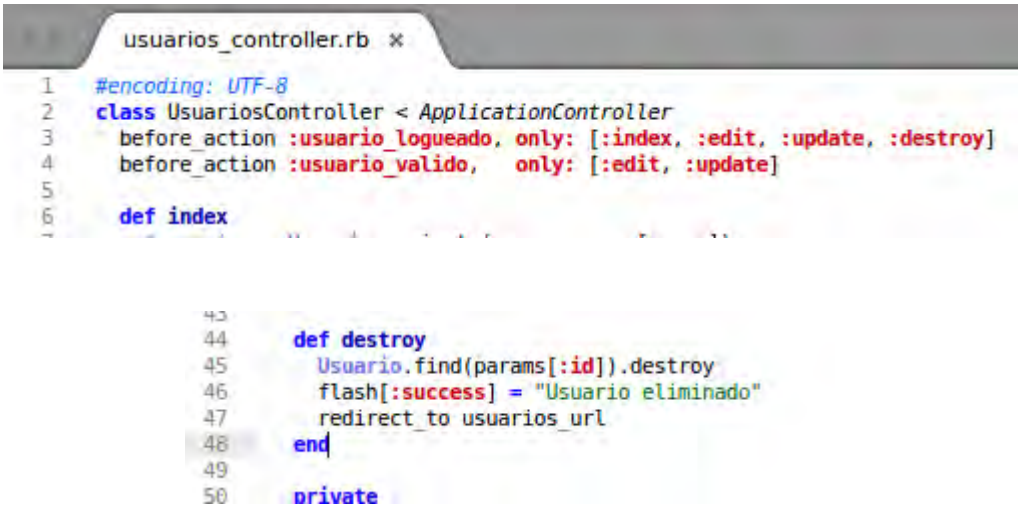
Para completar el recurso *Usuario* debemos añadir los enlaces de tipo *delete* y añadir la acción `destroy`. Primero vamos a añadir los enlaces a cada usuario dentro del listado generado por la página *index*, restringiendo el acceso solo a usuarios que sea administradores.

_usuario.html.erb

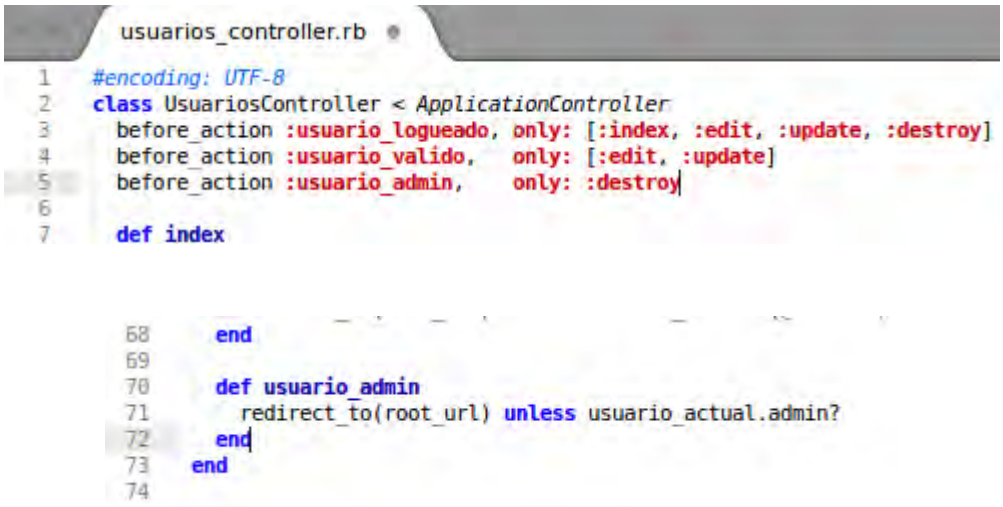
```
<li>
    <%= gravatar_de usuario %>
    <%= link_to usuario.nombre, usuario %>
    <% if usuario_actual.admin? && !usuario_actual?(usuario) %>
      |
    <%= link_to "eliminar", usuario, method: :delete, data: { confirm: "¿Seguro que
        deseas eliminar al usuario?" } %>
    <% end %>
</li>
```

Observemos que el argumento `method: :delete`, se encarga de moldear los enlaces para emitir la petición *DELETE* necesaria. Los navegadores web de forma nativa no pueden enviar petición de tipo *DELETE*, por eso *Rails* los engaña a través de *Javascript*.

Para que los enlaces funcionen, debemos de añadir la acción `destroy`. Ésta acción encuentra al usuario correspondiente y lo destruye a través del método `destroy` de *Active Record*, retornando al usuario al listado de usuarios en *index*. La acción la añadimos en el filtro `usuario_loguado`.



Todo bien hasta ahora pero tenemos dos agujeros de seguridad, uno es que si un usuario envía peticiones *DELETE* directamente de la linea de comandos, puede eliminar a cualquiera del sitio. Necesitamos tener control sobre la acción `destroy`. Para completar tal fin, creamos el filtro `usuario_admin` para restringir la acción sólo a usuarios que sean administradores:



10.5. Guardando cambios

Realizamos los pasos ya conocidos al finalizar cada capítulo. Para el siguiente trataremos con los mensajes (micropost) de cada usuario, así como su edición, restricciones y seguimiento de usuarios.

```
$ git add .

$ git commit -m ‘Edición de usuarios finalizado’

$ git checkout master

$ git merge edicion_usuarios

$ git push origin master
```

Capítulo 11: Mensajes de Usuario

11. Desarrollo

Con este capítulo completamos las acciones *REST* para los usuarios, a su vez empezamos nuestro segundo recurso: mensajes de usuario. Los mensajes de usuario serán cortos y están ligados a cada usuario. Construimos primeramente el modelo, asociándolo con el modelo *Usuario* gracias al uso de los métodos: `has_many` y `belong_to`.

Creamos la rama con la que trabajaremos.

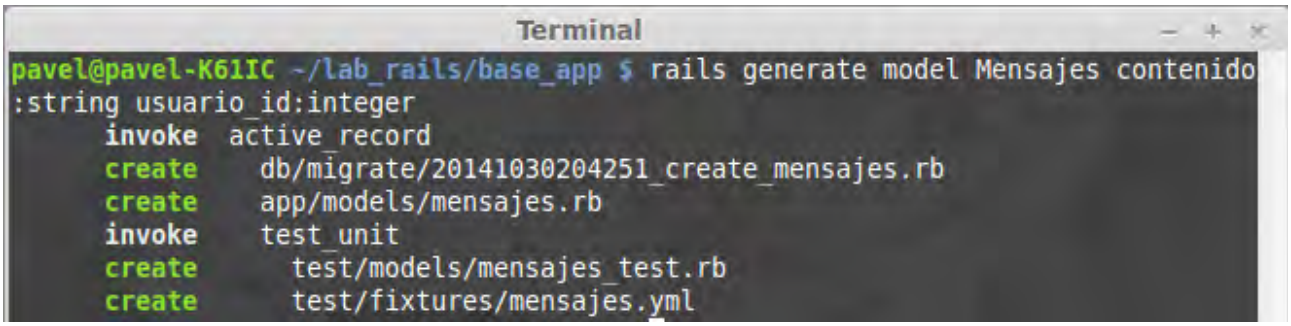
```
$ git checkout master

$ git checkout -b mensajes_usuario
```

11.1. Modelo Mensajes

El modelo de mensajes incluirá, validaciones y la asociación de éste con el modelo *Usuario*. El modelo *Mensajes* necesita dos atributos: *contenido* (que contiene el contenido del mensaje) y *usuario_id* (que asocia el mensaje con un usuario en particular).

Generamos el modelo:

```
$ rails generate model Mensajes contenido:string
usuario_id:integer
```



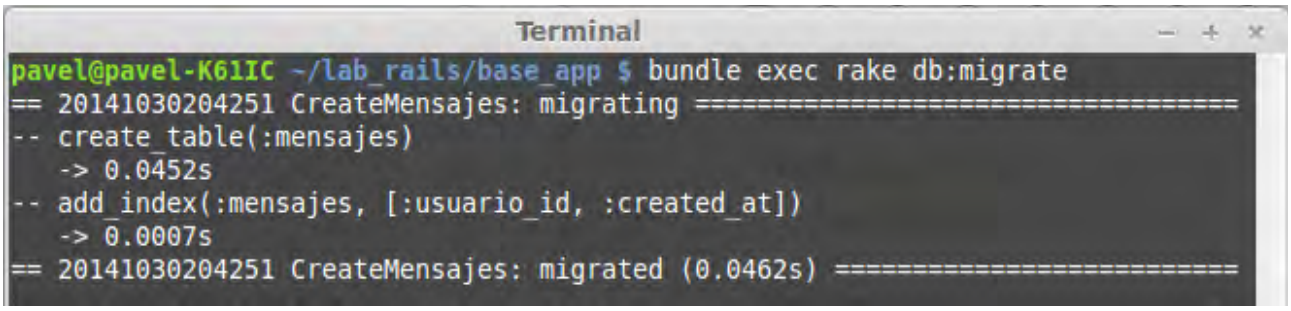
Notemos que el índice *index* se agrega a dos atributos dentro de la migración generada por el comando anterior: *usuario_id* y *created_at*. Esto es porque queremos regresar a los usuarios asociados dado un id en un orden de creación, también *Rails* crea una clave múltiple (*ActiveRecord* usara ambas claves al mismo tiempo).

20141030204251_create_mensajes.rb

```
1  class CreateMensajes < ActiveRecord::Migration
2    def change
3      create_table :mensajes do |t|
4        t.string :contenido
5        t.integer :usuario_id
6
7        t.timestamps
8      end
9      add_index :mensajes, [:usuario_id, :created_at]
10   end
11 end
12
```

Ahora migramos la base de datos:

```
$ bundle exec rake db:migrate
```



Nuestra base contará con la siguiente estructura (Tabla 11.1):

Tabla 11.1. Estructura del modelo *Mensajes*.

Mensajes	
id	integer
contenido	string
usuario_id	integer
created_at	datetime
updated_at	datetime

Para que podamos saber que usuario hizo un mensaje, necesitamos el *id* del usuario. Haremos uso de *ActiveRecord* para asociar este *id*, validando como hemos hecho en capítulos anteriores. Modificamos el archivo **mensajes.rb**.

app/models/mensajes.rb

```
mensajes.rb
1  class Mensajes < ActiveRecord::Base
2      validates :usuario_id, presence: true
3  end
4
```

11.1.2 Asociando modelos.

Al requerir saber qué mensaje está asociado a cuál alumno, debemos de asociar ambos modelos. Bajo esta asociación también podemos saber cuántos mensajes tiene un alumno especifico, cómo vemos en la Figura 11.1 y el caso contrario con la Figura 11.2.

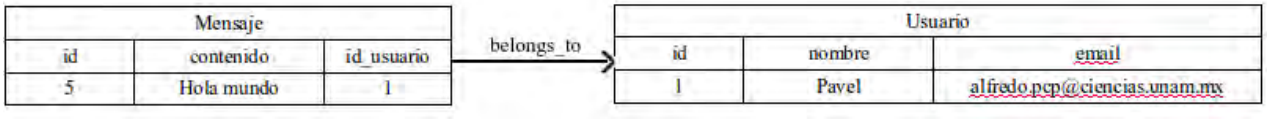


Figura 11.1. Relación con belongs_to.

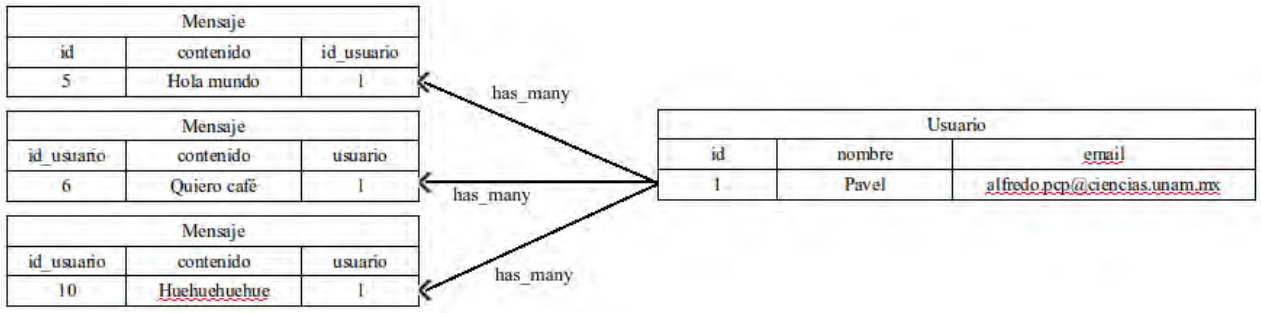


Figura 11.2. Relación con has_many.

Vemos qué haremos uso de los métodos `belongs_to` y `has_many`, *Rails* construye otras funciones a partir de ellos y que se muestran en la Tabla. 11.2.

Método	Propósito
mensaje.usuario	Nos regresa un objeto *usuario* asociado a un mensaje
usuario.mensajes	Nos regresa un arreglo de mensajes de un usuario
usuario.mensajes.create(argumento)	Crea un mensaje con user_id = user.id
usuario.mensajes.create! (argumento)	Crea un mensaje (surge una excepción si falla la creación del mismo)
usuario.mensajes.build(argumento)	Regresa un nuevo objeto *mensaje*

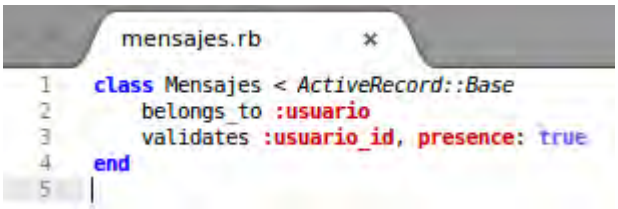
Tabla 11.2. Métodos que asocian usuario/mensaje.

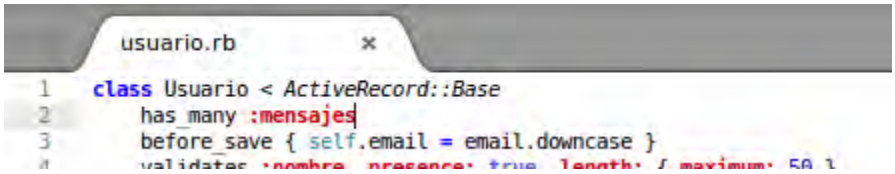
Vemos que tenemos lo siguiente:

```
usuario.mensajes.create
usuario.mensajes.create!
usuario.mensajes.build
```

Este patrón es la forma estándar de hacer un mensaje, a través de la asociación con un usuario. Cuando creamos un nuevo mensaje se hace de esta forma, el *id* del usuario es automáticamente asociado al valor correcto. Definiendo la asociación, la variable `@mensajes` automáticamente tiene un id de usuario igual al usuario asociado. Para retorna un usuario de cierto mensaje tenemos el método `mensajes.usuario`.

Vamos a agregar los métodos `belongs_to` y `has_many` a nuestro modelo **mensajes.rb** y a **usuario.rb**:





11.1.3. Orden y dependencia de mensajes de usuario

No sólo nos basta con verificar la existencia del atributo *mensajes*. En este capítulo vamos a añadir el orden y la dependencia de los mensajes. Primero, cuando recuperemos los mensajes de un usuario de la base de datos, tenemos que tener un orden descendente para mostrar los últimos mensajes primero. Para cumplir tal fin, usaremos la herramienta `default_scope` que recibe un argumento para el orden:

mensajes.rb

```
1  class Mensajes < ActiveRecord::Base
2      belongs_to :usuario
3      default_scope -> { order('created_at DESC') }
4      validates :usuario_id, presence: true
5  end
6
```

El orden viene dado por el argumento `created_at DESC`, que como debemos saber es un orden descendente en *SQL*. La sintaxis para el objeto es un *procedure*, toma el bloque y lo evalúa hasta que sea llamado por el método `call`. Por ejemplo:

```
>> -> { puts “foo” }

>> -> { puts “foo” }.call
```

```
pavel@pavel-K61IC ~/lab_rails/base_app $ rails c
Loading development environment (Rails 4.0.4)
1.9.3-p545 :001 > -> { puts "foo" }
 => #<Proc:0x000000039e6008@(irb):1 (lambda)>
1.9.3-p545 :002 > -> { puts "foo" }.call
foo
 => nil
```

Aparte del orden, necesitamos algo más para añadir mensajes. Recordemos que los administradores tienen el poder de eliminar usuarios (destruirlos). Por la misma razón, si un usuario es destruido, lo deben hacer también los mensajes que haya creado. Lo anterior lo logramos añadiendo una opción al método `has_many` como sigue:

usuario.rb

```
1  class Usuario < ActiveRecord::Base
2      has_many :mensajes, dependent: :destroy
3      before_save { self.email = email.downcase }
```

La opción que pasamos al método `has_many`, se las arregla para que los mensajes dependientes (los que pertenecen a un usuario en concreto) sean destruidos cuando el

usuario sea destruido. Esto previene que los mensajes del usuario destruido se queden en la base de datos cuando algún administrador desee remover usuarios del sistema.

11.1.4. Validando el contenido.

Para poder pasar ya del modelo, nos hacen falta las validaciones del contenido. Debemos de verificar la existencia del atributo y no debe de rebasar los 250 caracteres. Lo logramos agregando la línea

```
validates :contenido, :presence: true, length: {  maximum:
250 }
```

```
1  class Mensaje < ActiveRecord::Base
2      belongs_to :usuario
3      default_scope -> { order('created_at DESC') }
4      validates :contenidq, presence: true, length: { maximum:250 }
5      validates :usuario_id, presence: true
6  end
```

11.2. Mostrar mensajes

En esta sección vamos a mostrar los mensajes que cree un usuario, lo haremos en la página *show* del usuario y no en una página tipo *index* para tener una mejor interacción y un enfoque de una pequeña red social. La maqueta de dicha interfaz quedaría como:

Figura 11.3. Maqueta de mensajes de usuario.

La idea es crear una serie de mensajes asociados con un usuario, luego verificamos que la página dedicada a los mensajes tenga el contenido de cada uno, asimismo debe mostrar el número de mensajes que se muestran (para esto usaremos el método `count`).

show.html.erb

```
1  <% provide(:titulo, @usuario.nombre) %>
2
3  <div class = "row">
4      <aside class = "span4">
5          <section>
6              <h1>
7                <%= gravatar_de @usuario %>
8                <%= @usuario.nombre %>
9              </h1>
10         </section>
11     </aside>
12     <div class="span8">
13         <% if @usuario.mensajes.any? %>
14             <h3>Mensajes (<%= @usuario.mensajes.count %>)</h3>
15             <ol class="mensajes">
16                 <%= render @mensajes %>
17             </ol>
18             <%= will_paginate @mensajes %>
19         <% end %>
20     </div>
21 </div>
```

Notemos que añadimos la paginación a los mensajes con

```
<%= will_paginate @mensajes %>
```

Esto funciona porque `will_paginate` asume la existencia de la variable de instancia `@usuarios`. Seguimos en el controlador `Usuarios` pero como queremos paginar los mensajes, pasamos la variable `@mensajes` al método de paginación. Obviamente, debemos definir dicha variable en la acción `show` de usuario.

```
<ol class=”mensajes”>
  <%= render @mensajes%>
</ol>
```

Con las líneas anteriores mostramos los mensajes en sí y los mostramos de manera ordenada con la etiqueta `<ol>`. Para que el *render* funcione correctamente debemos crear el parcial correspondiente (como lo hicimos con los usuarios en el capítulo anterior). Definimos el parcial **_mensaje.html.erb** (Código 11.1) como sigue:

app/views/mensajes/_mensaje.html.erb

```
_mensaje.html.erb
<li>
    <span class="contenido"><%= mensaje.contenido %></span>
    <span class="timestamp">
        Creado hace<%= time_ago_in_words(mensaje.created_at) %>.
    </span>
</li>
```

Código 11.1. **_mensaje.html.erb**.

Usamos el método `time_ago_in_words` que nos mostrará la fecha en la que se creo el post usando palabras. Se verá más adelante su funcionamiento.

Definimos la variable `@mensajes` en el controlador de usuarios para que todo lo anterior funcione:

```
usuarios_controller.rb
#encoding: UTF-8
class UsuariosController < ApplicationController
  before_action :usuario_logueado, only: [:index, :edit, :update, :destroy]
  before_action :usuario_valido,   only: [:edit, :update]
  before_action :usuario_admin,    only: :destroy

  def index
    @usuarios = Usuario.paginate(page: params[:page])
  end

  def show
    @usuario = Usuario.find(params[:id])
    @mensajes = @usuario.mensajes.paginate(page: params(:page))
  end

  def new
```

La página resultante es la siguiente:

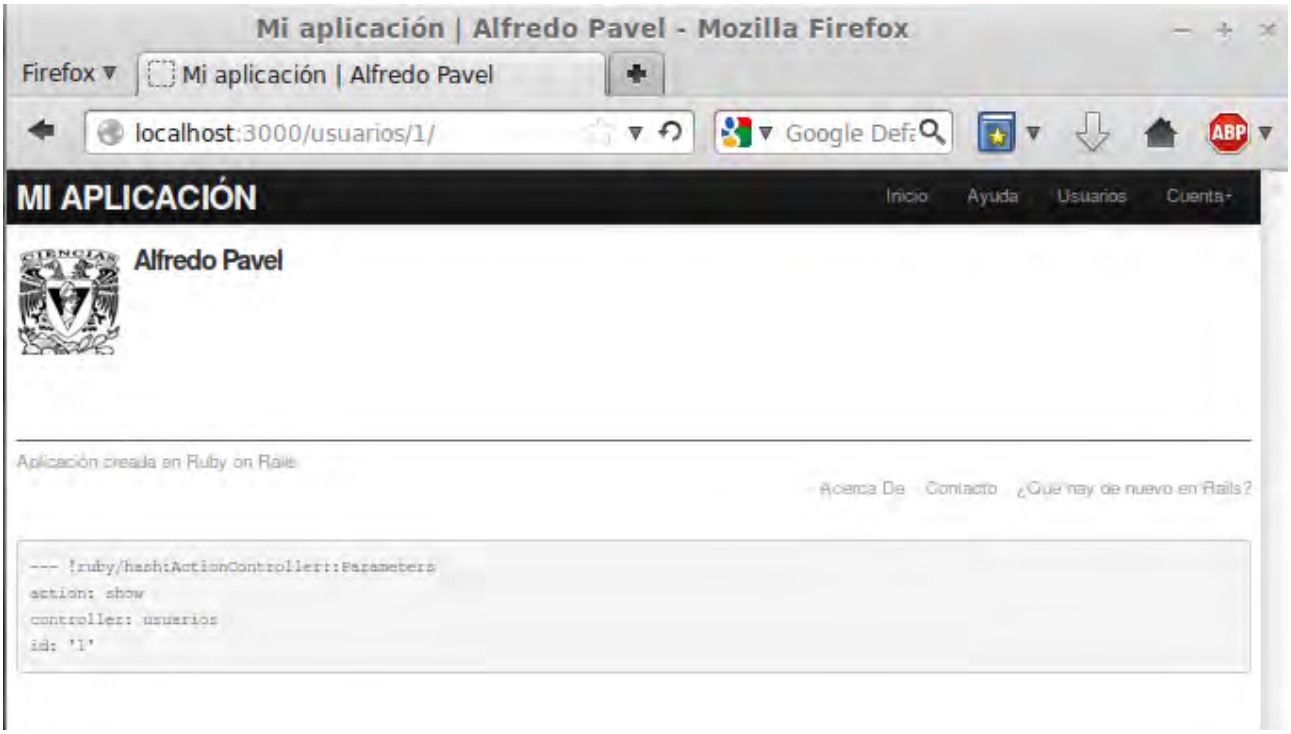


Figura 11.4. Página de perfil.

11.2.1. Primeros mensajes

Ahora que tenemos nuestra vista para mostrar mensajes de usuario, debemos ahora de crear dichos mensajes. De la lista que generamos en el capítulo anterior nos quitaría bastante tiempo en generar mensajes para todos, por lo mismo limitemos los usuarios a solo cinco la opción `:limit`.

```
Usuarios = Usuario.all(limit:  5)
```

Crearemos 50 mensajes por cada uno de los usuarios utilizando la *gema Faker* y el método `Lorem.sentence`, para esto hacemos una nueva tarea para la base con los siguientes datos:

```
17      end
18      usuarios = Usuario.all(limit: 5)
19      50.times do
20        contenido = Faker::Lorem.sentence(5)
21        usuarios.each { |usuario| usuario.mensajes.create!(contenido: contenido) }
22      end
23    end
24  end
```

Para generar los mensajes de los usuarios, tenemos que correr la tarea `db:llenaDB`

```
$ bundle exec rake db:reset

$ bundle exec rake db:llenaDB
```

En este punto ya podemos ver el resultado de todo lo que hemos hecho anteriormente, podemos ir al perfil del primer usuario y obtendremos sus mensajes, justo como en la Figura 11.5:

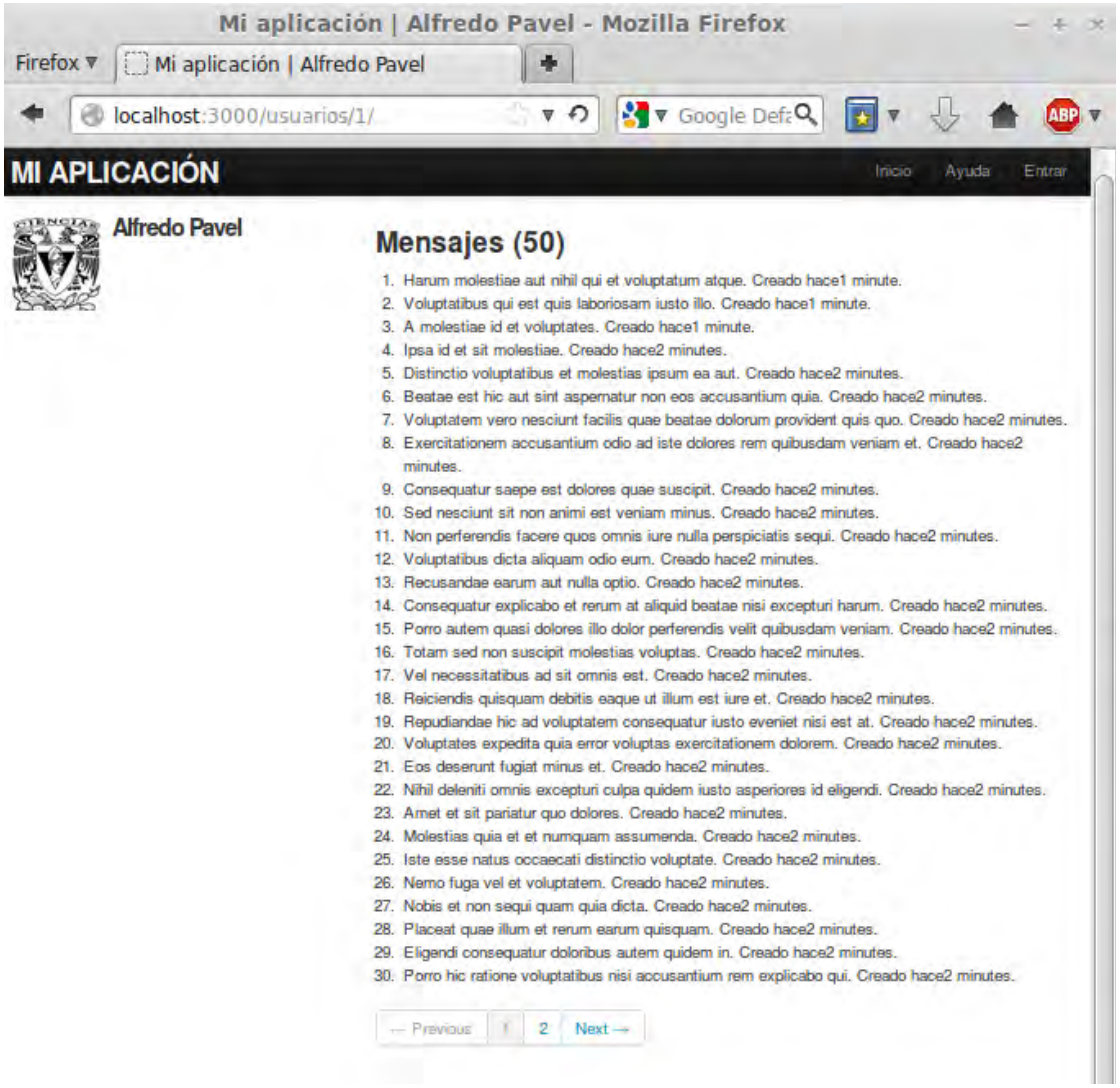


Figura 11.5. Mensajes de usuarios.

Vemos arriba que los mensajes todavía no cuenta con un formato apropiado, usaremos nuestra hoja de estilo *CSS* y añadimos el estilo correspondiente:

```
195
196  /*Estilo para mensajes*/
197
198  .mensajes {
199    list-style: square;
200    margin: 20px 0 0 0;
201    li {
202    padding: 10px 0;
203    border-top: 2.5px solid #dcdcdc;
204    }
205  }
206
207  .timestamp {
208    color: #bcbcbc;
209  }
210
211  .contenido {
212  display: block;
213  margin-bottom: 20px;
214  }
215
216  .gravatar {
217    float: left;
218    margin-right: 10px;
219  }
220
221  aside {
222    textarea {
223      height: 100px;
224      margin-bottom: 5px;
225    }
226  }
227
228
```

La página principal de los mensajes queda como sigue:

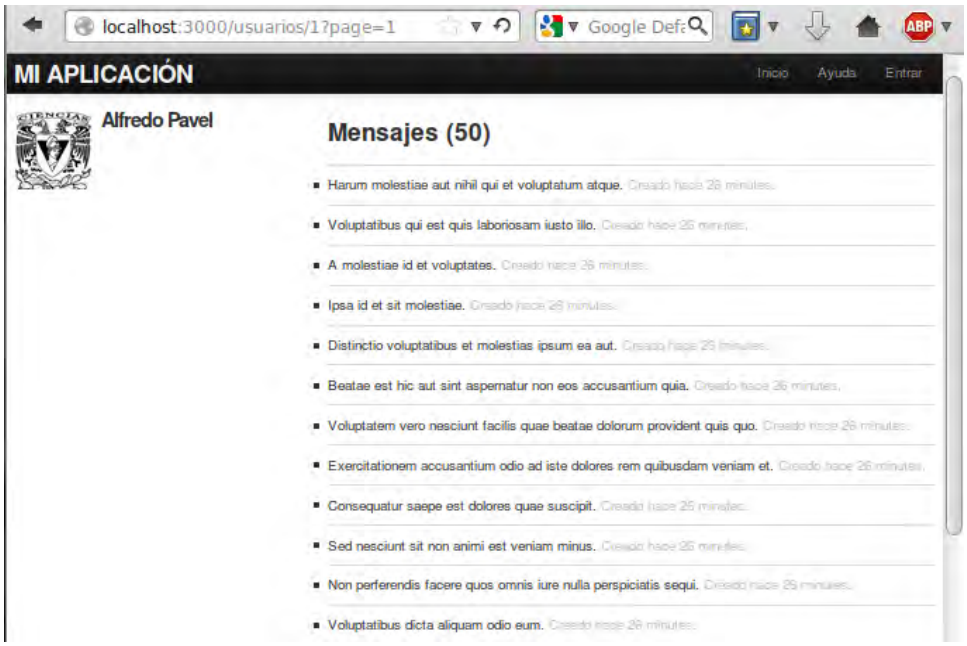


Figura 11.6. Mensajes con estilo.

Por último vemos que la paginación funciona y numera las páginas con los mensajes del usuario como se ve en la Figura 11.7.

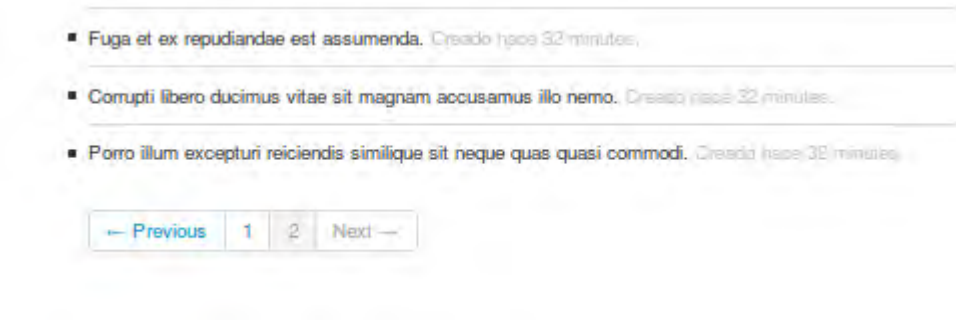


Figura 11.7. Paginación de mensajes.

Notemos que los tiempos de creación de los mensajes se generan con palabras ("*1 minute ago*" por ejemplo), esto es gracias al método `time_ago_in_words` usado previamente.

11.3. Modificando mensajes

En este apartado vamos a generar el formulario web para la creación de los mensajes de usuario. También tendremos una pequeña idea sobre la *lista de mensajes* que tiene como función llevar un registro de los mensajes de usuario que nos interesan. Por último añadimos la opción de eliminar mensajes de usuario.

Como la interfaz de los mensajes se basa en los controladores `Usuario` y `PaginasEstaticas`, no necesitamos crear acciones `new` o `edit` en el controlador `Mensajes`; las únicas acciones necesarias son `create` y `destroy`.

Añadimos las rutas correspondientes a nuestro recurso *Mensajes*:

routes.rb

```
1  BaseApp::Application.routes.draw do
2    resources :usuarios
3    resources :sesiones, only: [:new, :create, :destroy]
4    resources :mensajes, only: [:create, :destroy]
5    root 'paginas_estaticas#inicio'
```

El recurso anterior contiene las rutas *REST* siguientes:

Tabla 11.3. Rutas *REST* para mensajes.

URL	Petición HTTP	Acción Correspondiente	Función
/mensajes	POST	create	Crea un nuevo mensaje
/mensajes/x	DELETE	destroy	Elimina el mensaje con id x

11.3.1. Acceso

Empezamos con nuestro recurso *Mensajes* con algunos parámetros de control en el correspondiente controlador. Como hemos estado manejando, las acciones `create` y `destroy` deben tener como requisito haber iniciado sesión. Para esto tenemos que usan un filtro que llame al método `usuario_logueado`, sólo necesitábamos éste método en el controlador `Usuario` pero ahora lo necesitamos en el controlador `Mensajes` así que lo movemos al *helper Sesiones* (es decir, *cortamos* el código en el controlador `Usuario` y lo pegamos en el controlador `Mensajes`):

```
23    def usuario_actual?(usuario)
24      usuario == usuario_actual
25    end
26
27    def usuario_logueado
28      unless inicio_sesion?
29        guardar_ubicacion
30        redirect_to iniciases_url, notice: "Por favor, inicia sesión"
31      end
32    end
```

Con lo anterior ya tenemos acceso al método `usuario_logueado` en el controlador **mensajes_controller.rb** (Código 11.2), ya contamos con un acceso restringido a las acciones `create` y `edit` con el filtro siguiente:

app/controllers/mensajes_controller.rb

```
mensajes_controller.rb ×
1  class MensajesController < ApplicationController
2      before_action :usuario_logueado
3
4      def create
5      end
6
7      def destroy
8      end
9  end
```

Código 11.2. **mensajes_controller.rb**.

11.3.2. Creación de mensajes

La implementación al crear mensajes es similar a la creación de usuarios, la diferencia es que el formulario de creación de mensajes, se situará dentro de la página de perfil, es decir, bajo la ruta raíz "/" como en la Figura 11.8.

Figura 11.8. Maqueta para crear un mensaje.

Como la creación de mensajes sólo existe para los usuarios registrados y que estén logueados, un objetivo primordial es tener versiones diferentes de la página principal dependiendo si el usuario es un visitante o un usuario que ha iniciado sesión.

Primero implementamos nuestra acción `create`, similar a como en los usuarios; la diferencia está en usar la asociación usuario-mensaje para construir (`build`) el mensaje nuevo como se muestra a continuación:

mensajes_controller.rb

```ruby
class MensajesController < ApplicationController
    before_action :usuario_logueado

    def create
        @mensaje = usuario_actual.mensajes.build(mensaje_params)
        if @mensaje.save
            flash[:success] = "¡Mensaje creado!"
            redirect_to root_url
        else
            render 'paginas_estaticas/inicio'
        end
    end

    def destroy
    end


    private

    def mensaje_params
        params.require(:mensaje).permit(:contenido)
    end
end
```

Ahora creamos nuestro código *HTML* para el formulario de creación de mensajes, que dependiendo de si el usuario ha iniciado sesión o no, muestra la página principal correspondiente:

inicio.html.erb

```
<% if inicio_sesion? %>
    <div class="row">
        <aside class="span4">
            <section>
                <%= render 'shared/datos_usuario' %>
            </section>
            <section>
                <%= render 'shared/form_mensaje' %>
            </section>
        </aside>
    </div>
<% else %>

    <div class="center hero-unit">

        <h1>Mi Aplicación</h1>
        <h2>Esta página es el comienzo de una aplicación sencilla pero que en un futuro crecerá</h2>
        <p>
            Esta es la página de prueba para
            las vistas de mi aplicación
        </p>
<%= link_to "¡Registrate, es rápido y fácil!", registro_path, class: "btn btn-large btn-primary" %>
    </div>

<%= link_to image_tag("rails_icono.png", alt: "Rails Icono"), 'http://rubyonrails.org/' %>
<% end %>
```

Con el código anterior, debemos crear los parciales *datos_usuario* y *form_mensaje* respectivamente, como se muestra en el Código 11.3 y Código 11.4 respectivamente.

app/views/shared/_datos_usuario.html.erb

_datos_usuario.html.erb

```
<%= link_to gravatar_de(usuario_actual), usuario_actual %>
<h1>
    <%= usuario_actual.nombre %>
</h1>
<span>
    <%= link_to "Mi perfil", usuario_actual %>
</span>
<span>
    <%= pluralize(usuario_actual.mensajes.count, "mensajes") %>
</span>
```

Código 11.3. **mensajes_controller.rb**.

Usamos el método `pluralize` para referirnos a la cantidad de mensajes en plural.

App/views/shared/_form_mensaje.html.erb

_form_mensaje.html.erb

```
<%= form_for(@mensaje) do |atributo| %>
    <%= render 'shared/mensajes_error', object: atributo.object %>
        <div class="field">
            <%= atributo.text_area :contenido, placeholder: "Escribe lo que piensas..." %>
        </div>
    <%= atributo.submit "Enviar", class: "btn btn-large btn-primary" %>
<% end %>
```

Código 11.4. **_form_mensaje.html.erb**.

Para que el código del formulario funcione, debemos definir la variable `@mensaje` como sigue:

paginas_estaticas_controller.rb

```
class PaginasEstaticasController < ApplicationController
  def inicio
    @mensaje = usuario_actual.mensajes.build if inicio_sesion?
  end

  def ayuda
  end

  def acercade
  end
```

También debemos redefinir el parcial *mensajes_error*, recordemos que este parcial hace referencia a la variable `@usuario` y en nuestro caso usamos la variable `@mensaje`. Debemos definir mensajes de error que puedan trabajar sin importar el tipo de objeto que le pasemos. En nuestro parcial **mensajes_error.html.erb** la variable antes

mencionada puede acceder al objeto asociado a través de `atributo.object`. Para pasar el objeto al parcial, usamos un *hash* con un valor igual al objeto y una clave igual al nombre designado a la variable en el parcial:

```
<%= render 'shared/mensajes_error', object: atributo.object
%>
```

Con este objeto podemos construir un mensaje de error personalizado como sigue:

_mensajes_error.html.erb

```
<% if object.errors.any? %>
  <div id="errores_estilo">
    <div class="alert alert-error">
      <% if object.errors.count == 1 %>
        Existe 1 error
      <% else %>
        Existen <%= object.errors.count %> errores
      <% end %>
    </div>
    <ul>
    <% object.errors.full_messages.each do |msg| %>
        <% if msg == "Nombre can't be blank" %>
            <li>* <%= "Debes llenar el campo con tu nombre" %></li>
        <% elsif msg == "Email can't be blank" %>
            <li>* <%= "Debes llenar el campo con tu correo electrónico (email)" %></li>
        <% elsif msg == "Email is invalid" %>
            <li>* <%= "El correo electrónico que proporcionaste no es válido" %></li>
        <% elsif msg == "Password can't be blank" %>
            <li>* <%= "La contraseña no puede ser vacía" %></li>
        <% elsif msg == "Password is too short (minimum is 8 characters)" %>
            <li>* <%= "La longitud mínima para la contraseña es de 8 caracteres" %></li>
        <% elsif msg == "Password confirmation doesn't match Password" %>
            <li>* <%= "La contraseña no coincide" %></li>
        <% elsif msg == "Email has already been taken" %>
            <li>* <%= "El correo ya está registrado" %></li>
        <% end %>
    <% end %>
    </ul>
  </div>
<% end %>
```

Ahora actualicemos los mensajes de error correspondientes a los usuarios, tanto para su registro como para editar información:

new.html.erb

```
<% provide(:titulo, 'Registro') %>
<h1>Registro</h1>

<div class = "row">
  <div class = "span6 offset3">
    <%= form_for(@usuario) do |atributo| %>
      <%= render 'shared/mensajes_error', object: atributo.object %>

      <%= atributo.label :nombre %>
```

edit.html.erb

```
1
2  <% provide(:titulo, "Editar usuario") %>
3  <h1>Actualiza tu perfil</h1>
4
5  <div class="row">
6    <div class="span6 offset3">
7      <%= form_for(@usuario) do |atributo| %>
8        <%= render 'shared/mensajes_error', object: atributo.object %>
9
10       <%= atributo.label :nombre %>
```

Hasta este punto podemos verificar todo el trabajo anterior como se muestra en la Figura 11.9 y en la Figura 11.10.

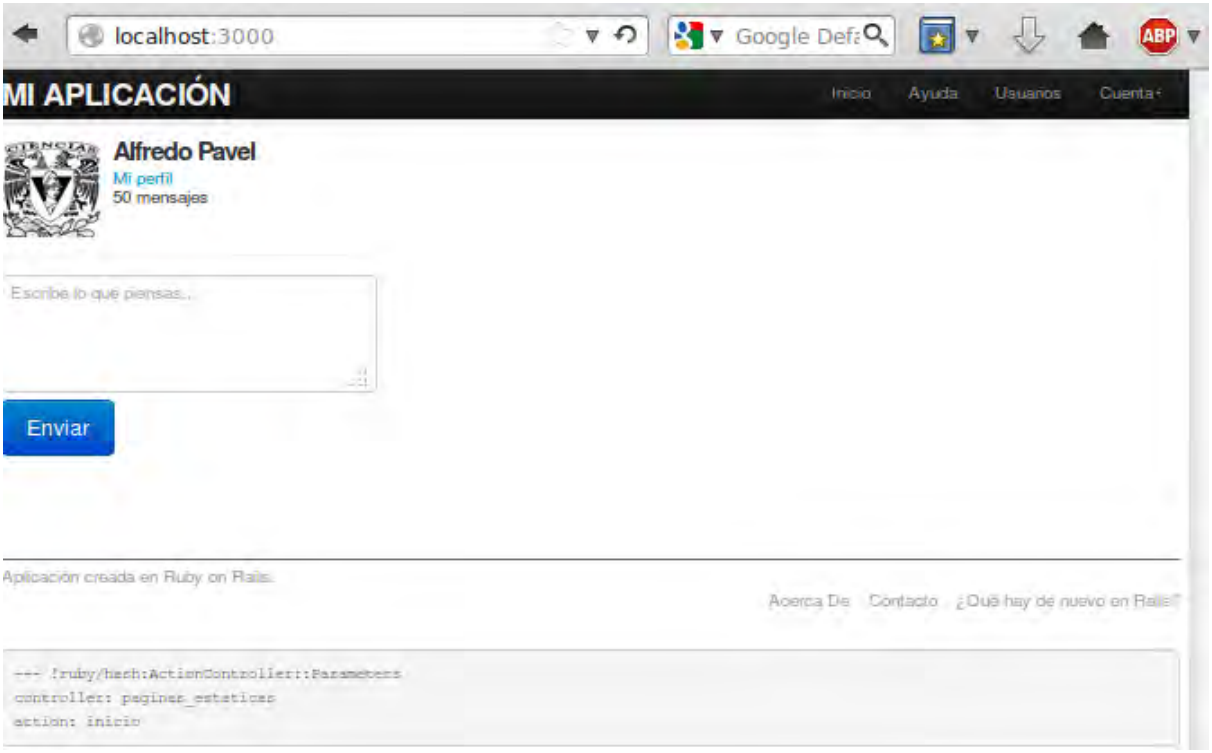


Figura 11.9. Página de inicio.

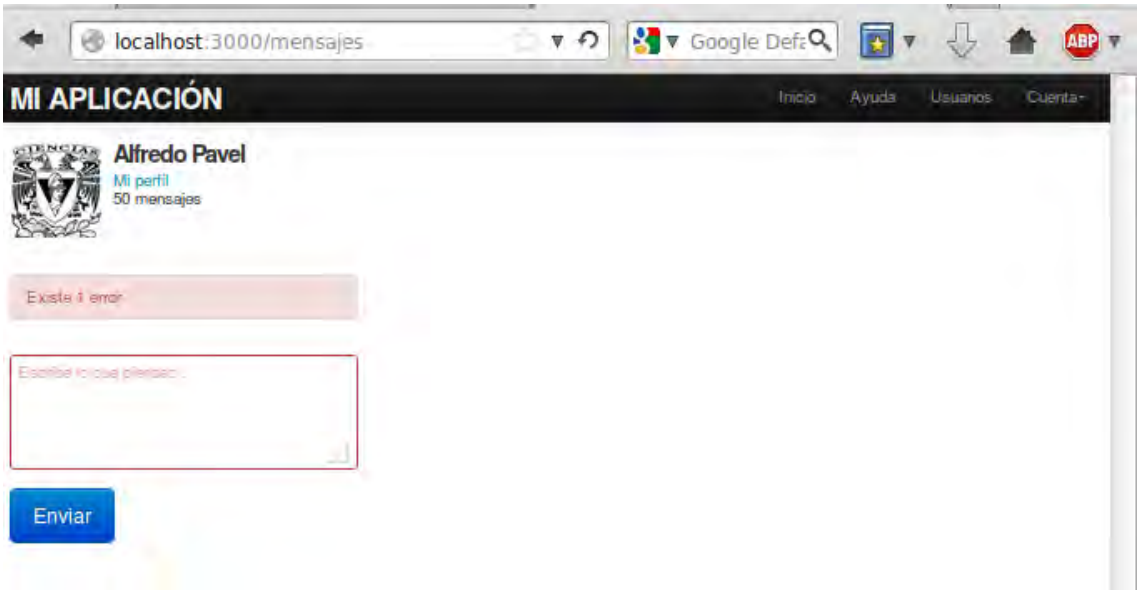


Figura 11.10. Mensaje de error.

11.3.3. Muro de mensajes

Esta sección tendrá como fin, el hacer una lista en nuestra página de perfil con todos los mensajes que hayamos creado, una especie de muro de mensajes si le ponemos nombre. Debemos de tener un método que sólo incluya los mensajes del usuario actual y que excluya los mensajes de otros usuarios. Vamos a añadir dicho el método `muro` a nuestro modelo seleccionando todos los mensajes con el atributo *usuario_id* igual al *id* del usuario actual, lo cual logramos con el método `where` del modelo *Mensajes*.

usuario.rb

```
1  class Usuario < ActiveRecord::Base
2    has_many :mensajes, dependent: :destroy
3    before_save { self.email = email.downcase }
4    validates :nombre, presence: true, length: { maximum: 50 }
5    VALIDA_EMAIL = /\A[\w+\-.]+@[a-z\d\-]+(?:\.[a-z\d\-]+)*\.[a-z]+\z/
6    validates :email, presence: true, format: { with: VALIDA_EMAIL },
7              uniqueness: { case_sensitive: false }
8    has_secure_password
9    validates :password, length: { minimum: 8 }
10
11   def Usuario.new_token_recuerdo
12       SecureRandom.urlsafe_base64
13     end
14
15     def Usuario.digest(token)
16       Digest::SHA1.hexdigest(token.to_s)
17     end
18
19     def muro
20       Mensaje.where("usuario_id = ?", id)
21     end
22
23 private
```

Para usar el muro en nuestra aplicación, primeramente añadimos la variable de instancia `@items_muro` para el usuario actual, luego añadimos el parcial **muro.html.erb** a la página de inicio (Código 11.5).

paginas_estaticas_controller.rb

```
class PaginasEstaticasController < ApplicationController

  def inicio
    if inicio_sesion?
      @mensaje = usuario_actual.mensajes.build
      @items_muro = usuario_actual.muro.paginate(page: params[:page])
    end
  end

  def ayuda
```

app/views/shared/_muro.html.erb

_muro.html.erb

```
<% if @items_muro.any? %>
  <ol class="mensajes">
    <%= render partial: 'shared/item_muro', collection: @items_muro %>
  </ol>
<%= will_paginate @items_muro %>
<% end %>
```

Código 11.5. **_muro.html.erb**.

En el código anterior pasamos el parámetro `:collection` con los items del muro, lo que hace que el render use el parcial que le damos (*item_muro*) para mostrar cada item en la colección. Para que esto funcione, implementamos el parcial **item_muro.html.erb** (Código 11.6)**.**

app/views/shared/_item_muro.html.erb

_item_muro.html.erb

```
<li id="<%= item_muro.id %>">
    <%= link_to gravatar_de(item_muro.usuario), item_muro.usuario %>
    <span class="usuario">
        <%= link_to item_muro.usuario.nombre, item_muro.usuario %>
    </span>
    <span class="contenido"><%= item_muro.contenido %></span>
    <span class="timestamp">
        Creado hace <%= time_ago_in_words(item_muro.created_at) %>.
    </span>
</li>
```

Código 11.6. **_item_muro.html.erb**.

Ahora podemos añadir el muro a la página de inicio haciendo un render al parcial correspondiente. El resultado lo vemos en la Figura 11.11.

```erb
<% if inicio_sesion? %>
    <div class="row">
        <aside class="span4">
            <section>
                <%= render 'shared/datos_usuario' %>
            </section>
            <section>
                <%= render 'shared/form_mensaje' %>
            </section>
        </aside>
        <div class="span8">
            <h3>Muro de mensajes</h3>
            <%= render 'shared/muro' %>
        </div>
    </div>
<% else %>
```

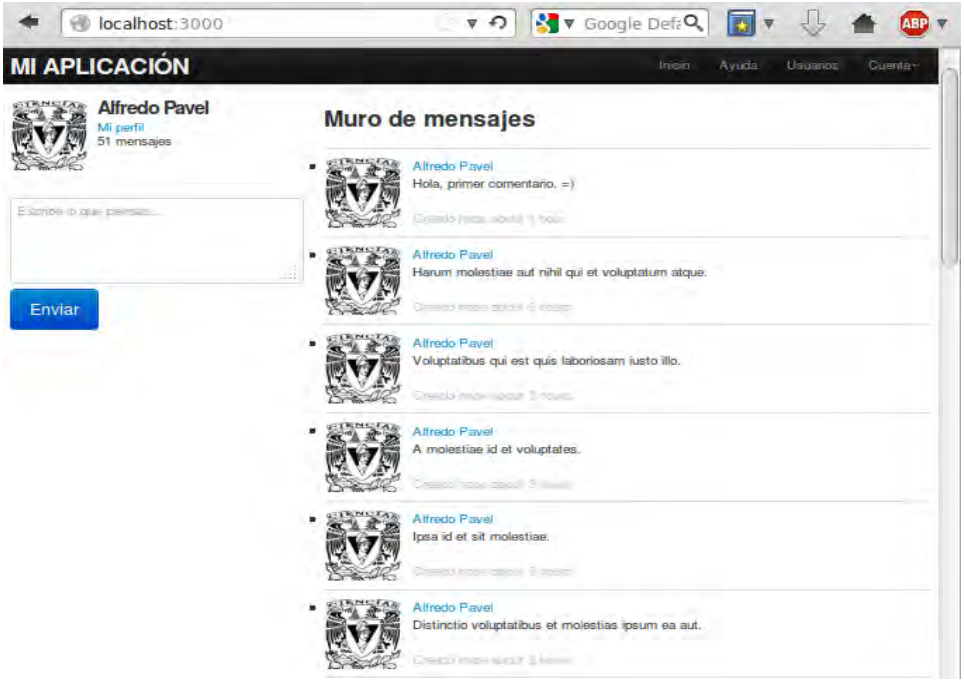


Figura 11.11. Muro de mensajes.

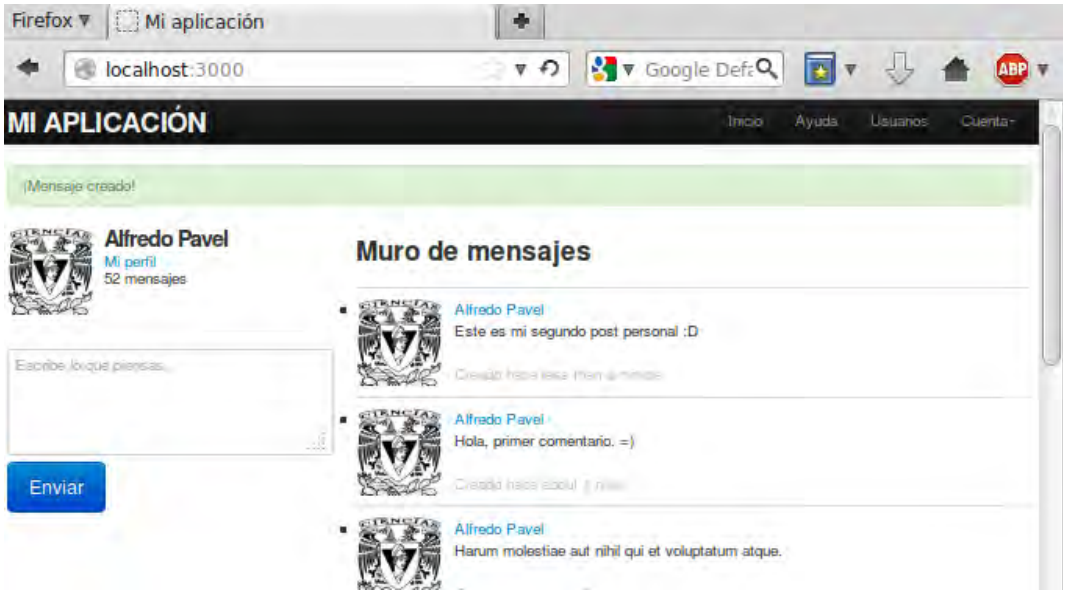


Figura 11.12. Creación de un nuevo mensaje.

Al crear un nuevo mensaje todo funciona según lo esperado. Tenemos un pequeño problema y es cuando falla el envío de un mensaje, la página de inicio esperaría a la variable de instancia `@items_muro` y al no encontrarlo se "rompe". La solución sencilla es suprimir todo el muro haciendo una asignación a un arreglo vacío:

mensajes_controller.rb

```
1  class MensajesController < ApplicationController
2      before_action :usuario_logueado
3
4      def create
5          @mensaje = usuario_actual.mensajes.build(mensaje_params
6          if @mensaje.save
7              flash[:success] = "¡Mensaje creado!"
8              redirect_to root_url
9          else
10             @items_muro = []
11             render 'paginas_estaticas/inicio'
12         end
13     end
```

11.3.4. Eliminando mensajes

Sólo nos falta para completar este capítulo, que podamos eliminar los mensajes. Tenemos una maqueta como la que sigue para eliminar mensajes los cuales a diferencia de la eliminación de usuarios que se hace a través de un administrador, lo puede hacer el usuario que haya iniciado sesión.

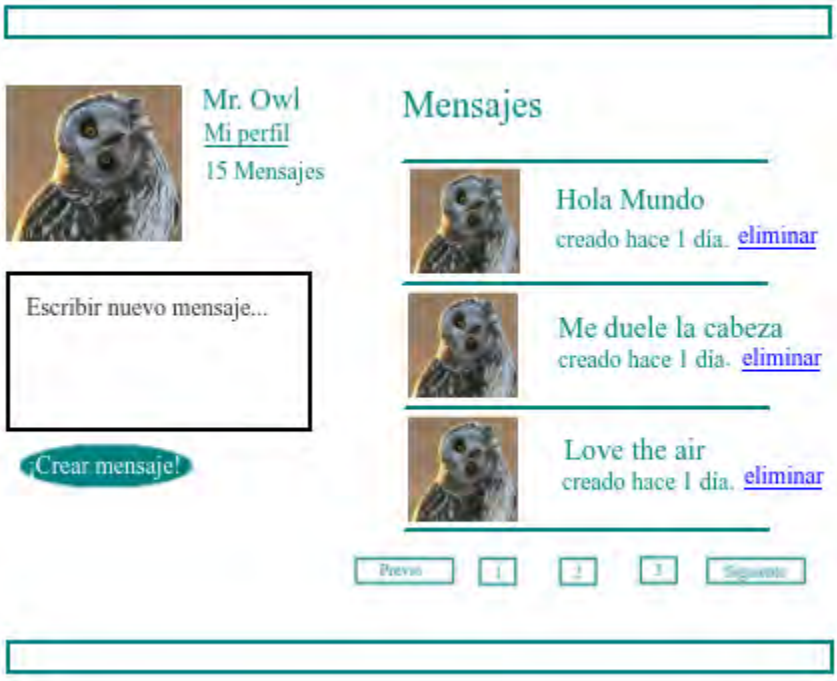


Figura 11.13. Maqueta para eliminar mensajes.

Añadimos un enlace para eliminar un mensaje en el parcial **mensaje.html.erb** y también lo añadimos al parcial **item_muro.html.erb** como se muestra a continuación:

_mensaje.html.erb

```
<li>
    <span class="contenido"><%= mensaje.contenido %></span>
    <span class="timestamp">
        Creado hace <%= time_ago_in_words(mensaje.created_at) %>.
    </span>
    <% if usuario_actual?(mensaje.usuario) %>
        <%= link_to "eliminar", mensaje, method: :delete,
        data: { confirm: "¿Estás seguro?" },
        title: mensaje.contenido %>
    <% end %>
</li>
```

_item_muro.html.erb

```
<li id="<%= item_muro.id %>">
    <%= link_to gravatar_de(item_muro.usuario), item_muro.usuario %>
    <span class="usuario">
        <%= link_to item_muro.usuario.nombre, item_muro.usuario %>
    </span>
    <span class="contenido"><%= item_muro.contenido %></span>
    <span class="timestamp">
        Creado hace <%= time_ago_in_words(item_muro.created_at) %>.
    </span>
    <% if usuario_actual?(item_muro.usuario) %>
        <%= link_to "eliminar", item_muro, method: :delete,
        data: { confirm: "¿Estás seguro?" },
        title: item_muro.contenido %>
<% end %>
</li>
```

Usaremos el filtro `usuario_mensajes` para verificar que el usuario actual tenga mensajes con su *id*.

mensajes_controller.rb

```
class MensajesController < ApplicationController
    before_action :usuario_logueado, only: [ :create, :destroy]
    before_action :usuario_mensajes,   only: :destroy

    def create
        @mensaje = usuario_actual.mensajes.build(mensaje_params)
        if @mensaje.save
            flash[:success] = "¡Mensaje creado!"
            redirect_to root_url
        else
            @items_muro = []
            render 'paginas_estaticas/inicio'
        end
    end

    def destroy
        @mensaje.destroy
        redirect_to root_url
    end


    private

    def mensaje_params
        params.require(:mensaje).permit(:contenido)
    end

    def usuario_mensajes
        @mensaje = usuario_actual.mensajes.find_by(id: params[:id])
        redirect_to root_url if @mensaje.nil?
    end
end
```

Notemos que encontramos los mensajes usando el filtro a través de la asociación:

```
usuario_actual.mensajes.find_by(id: params[:id])
```

Lo anterior nos asegura obtener sólo los mensajes pertenecientes al usuario actual. Los resultados de implementar el código anterior lo vemos en la página de inicio (Figura 11.14) y si eliminamos un mensaje nos mostrará la alerta correspondiente y el mensaje ya no se mostrará (Figura 11.15).

Figura 11.14. Opción eliminar mensaje.

Figura 11.15. Mensaje eliminado.

11.4. Guardando cambios

Guardamos los cambios y los subimos al repositorio. En el siguiente capítulo daremos fin a nuestra aplicación añadiendo la función de "seguimiento" a los usuarios, los cuales podrán "seguir" a otros usuarios y poder ver sus mensajes en el perfil personal.

```
$ git add .

$ git commit -m 'Mensajes finalizado'

$ git checkout master

$ git merge  mensajes_usuario

$ git push origin master
```

Capítulo 12: Conclusiones

El objetivo de este trabajo es apoyar a los alumnos de Ingeniería de Software que quieran hacer la aplicación del curso con *Ruby on Rails*.

El apoyo es a través de enseñar el desarrollo de una aplicación web de *microblogging* que permita la interacción entre usuarios a través de mensajes (*post*), usando el marco de trabajo *Ruby on Rails*, lo cual significa adquirir y/o expandir los conocimientos y habilidades necesarias para el desarrollo de software para la plataforma web. Apoyando a los estudiantes además del manejo de *Ruby on Rails,* el conjunto de habilidades que incluyen: control de versiones, *HTML/CSS*, bases de datos, creación y modificación de usuarios y mensajes; seguridad básica, justo como destacamos en la introducción.

A lo largo de este trabajo hemos visto los pasos necesarios para el desarrollo de la aplicación web mencionada anteriormente. Aprendimos como instalar y configurar el ambiente de trabajo en un entorno *Linux*, con todas las herramientas necesarias. Empezamos a conocer el lenguaje *Ruby* y la sintaxis que se usa para desarrollar aplicaciones, familiarizando al alumno con el lenguaje. Con los conocimientos sobre *Ruby*, creamos una aplicación demo con *Rails* sencilla para dar paso al uso de páginas estáticas y dinámicas. Iniciamos el desarrollo de la aplicación principal con la adición de los usuarios al sistema, es decir, la creación, actualización, búsqueda y eliminación de los mismos, así como la creación de contraseñas seguras para la autenticación al sistema. Añadimos pequeñas funciones a nuestros usuarios como restricciones para el registro de usuarios, la capacidad de contar con una página y una imagen de perfil (*Gravatar*), así como un layout creado con *Bootstrap* que nos facilita la tarea del diseño de la página, simple pero bonita y funcional. Aprendimos la importancia y uso de sesiones en nuestra aplicación para tener mayor seguridad. Para una parte avanzada del trabajo, los usuarios ya son capaces de actualizar sus perfiles, pueden ver qué otros usuarios se encuentran en el sitio y añadimos también a los administradores, que serán los encargados de eliminar usuarios. Por último, añadimos el modelo de mensajes entre usuarios haciendo uso del modelo de usuarios para poder asociarlos a sus autores. Listamos los mensajes de los usuarios en su página de perfil, pudiendo editarlos o eliminarlos. Los usuarios también pueden ver incluso, mensajes de otros usuarios.

Todo lo anterior nos brinda las bases para empezar el desarrollo de aplicaciones web con el *framework Ruby on Rails*.

Se pueden crear nuevos proyectos con distintos enfoques de ambientes de desarrollo que nos brinden la oportunidad de expandir nuestros conocimientos para el desarrollo de funciones web e incluso interactuar con más tecnologías y lenguajes para crear sistemas más complejos pero a la vez, más funcionales, seguros y con un acabado profesional.

Bibliografía

- Flores, R., & Darrow, K. (2014). Getting Started with Bootstrap 3. Ryan Flores.
- Henderson, M. (2011). Ruby (Large print ed.). Thorpe.
- Pressman, R. S. (2010). Ingeniería de Sofware. McGraw-Hill.
- Sommerville, I. (2012). Ingeniería de Software. Addison-Wesley (Pearson).
- Pytel, C., & Saleh, T. (2011). Rails AntiPatterns: best practice Ruby on rails refactoring; [fully up-tp-date for Rails 3]. Upper Saddle River, NJ.: Addison-Wesley.
- Ruby, S., Thomas, D., & Heinemeier, D. (2013). Agile web development with Rails 4. Facets of Ruby.
- Marshall, K., Pytel, C., & Yurek, J. (2007). Pro Active record databases with Ruby and Rails. Berkeley, CA: Apress.
- Weizenbaum, N., Eppstein, C., Mathis, B., & Netherland, N. (2013). Sass and Compass in action. Shelter Island, NY: Manning Publications.
- Hartl M. (2012). Ruby on Rails 3 Tutorial: Learn Rails by Example. Addison-Wesley Professional
- Olsen R. (2011). Eloquent Ruby. Addison-Wesley Professional Ruby Series.
- Ruby S., & Thomas D., & Heinemeier H. D. (2013). Agile Web Development with Rails 4. Facets of Ruby.