

# Aprendiendo Ruby on Rails 5.0

Mario Alberto Chávez Cárdenas © 2016  
<http://railsenespanol.co>

# Bienvenido

Bienvenido a ésta aventura de conocer Ruby on Rails. Estas a punto adentrarte en el mundo de desarrollo Web a través de Ruby on Rails, él cual es conocido como un framework moderno que permite a los desarrolladores construir aplicaciones de manera muy rápida.

Ruby on Rails desde su llegada cambio la forma en como construimos aplicaciones Web, haciendo que nuestro trabajo como desarrolladores fuera más simple e inclusive divertido, características que hacen que éste framework sea tan atractivo para muchos desarrolladores.

Durante el curso de éste libro vamos a conocer que tiene Ruby on Rails que lo hace tan especial, pero antes de embarcarnos en ésta tarea, vamos a desviarnos un poco para iniciar por el principio, y el principio es el lenguaje de programación Ruby.

# Capítulo 1 - Ruby, el lenguaje de los desarrolladores felices

## Historia

Ruby es un lenguaje de programación de uso general, es decir no es lenguaje diseñado exclusivamente para el desarrollo Web, pero no podemos negar de que ha encontrado un nicho importante en ésta área.

La historia de Ruby se remonta hasta 1993, cuando fue creado por el japonés Yukihiro Matsumoto, mejor conocido como [Matz](#). Ruby fue liberado al público en 1995.

Desde entonces Ruby ha atraído a un gran número de desarrolladores debido a su diseño simple y sintaxis elegante.

Ruby es el resultado de la visión de [Matz](#) por crear un lenguaje de programación que mezcla ciertas características de sus lenguajes favoritos, según [Matz](#), él tomó en cuenta:

- Tomó la simplicidad de [Lisp](#), removiendo macros y expresiones [S](#)
- Agregó un sistema de objetos simple
- Agregó bloques, inspirado por funciones de orden superior
- Agregó métodos tal como existen en [Smalltalk](#)
- Agregó funcionalidad encontrada en [Perl](#), pero en una forma OO

Como podemos ver [Matz](#) tomó [Lisp](#), [Smalltalk](#) y [Perl](#) para crear Ruby, inclusive [Matz](#) alguna vez comentó que Ruby realmente era [Lisp](#) en un inicio <sup>1</sup>.

Ruby cuenta con una sintaxis diferente a muchos lenguajes de programación, algunos desarrolladores se refieren a una sintaxis simple, pero [Matz](#) la define como:

"He intentado hacer Ruby natural, no simple"

---

<sup>1</sup> <http://blade.nagaokaut.ac.jp/cgi-bin/vframe.rb?key=179515&cginame=namazu.rb&submit=Search&dbname=ruby-talk&max=50& whence=0>

Pero también ha comentado que:

"Ruby es sencillo en apariencia, pero es muy complejo internamente, tal como es el cuerpo humano"<sup>2</sup>.

El esfuerzo de [Matz](#) para lograr esto en Ruby ha sido exitoso, su sintaxis natural es simple de aprender, y es increíble como Ruby nos ayuda a ser productivos con muy pocas líneas de código. Éste nivel de productividad es lo que hace de los **Rubystas** ser **desarrolladores felices**.

## ¿Qué nos ofrece Ruby como lenguaje de programación?

Técnicamente Ruby es un lenguaje Orientado a Objetos, de hecho todo en Ruby es un objeto, bueno casi todo.

Con respecto a sus características de Orientación a Objetos, Ruby cumple con la definición de [Alan Kay](#)<sup>3</sup>, el creador del término Orientado a Objetos, es decir:

- Permite el envío de mensajes entre los objetos
- Protección y retención de datos (Encapsulación)
- Enlace dinámico de tipos y resolución de métodos (Late-binding)

Además de lo anterior, Ruby, soporta la definición más común de programación orientado a objetos, donde podemos definir clases que a su vez pueden ser instanciadas a objetos en memoria.

La herencia de clase también es soportada, pero en el caso de Ruby, una clase solo puede heredar de una super clase. Si bien esto puede sonar como una limitante contra otros lenguajes de programación, en Ruby es posible extender una clase mediante el uso de Módulos, es decir, una clase puede combinar más de un Modulo a un tiempo.

Otra característica que Ruby comparte con algunos de los lenguajes modernos de programación es el **Garbage Collector** o simplemente **GC**. El GC nos permite como desarrolladores el pedir un espacio de memoria para almacenar un dato, pero a diferencia de por ejemplo C, no tenemos que preocuparnos por liberar ese espacio de memoria explícitamente, en éste caso el GC se en-

---

<sup>2</sup> <http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/2773>

<sup>3</sup> <http://userpage.fu-berlin.de/>

carga de identificar los espacios que ya no son requeridos en cierto punto de ejecución de nuestro programa y marcar ese espacio para que pueda ser reutilizado para guardar nuevos datos.

El sistema de tipos de Ruby es dinámico, es decir no requiere de que explícitamente indiquemos el tipo de dato de una variable, Ruby al ejecutar nuestro código infiere el tipo de dato a partir del valor asignado. Esta cualidad de Ruby permite que nos enfoquemos en lo que un objeto puede hacer - hay que recordar que todo en Ruby es un objeto - y no en el tipo de dato, esto se conoce como **Duck Typing**.

Ruby pertenece a la categoría de los lenguajes dinámicos, es decir, Ruby no se compila como sucedería con un lenguaje por ejemplo de la familia de C. Nuestro código en Ruby es interpretado al momento de ejecutarse, esto sucede dentro de la maquina virtual de Ruby, **YARV - Yet Another Virtual Machine** -, conforme nuestro código va siendo cargado, YARV convierte el código a una representación intermedia donde le aplica micro-optimizaciones para después ser ejecutado.

La forma descrita de como Ruby se ejecuta a través de YARV es cierta cuando nos referimos al intérprete **MRI**, pero como veremos un poco mas adelante MRI no es el único interprete para Ruby.

El dinamismo de Ruby le otorga una de las herramientas más importantes del lenguaje: Metaprogramación. En términos simples Metaprogramación nos permite escribir programas que generan programas. Metaprogramación es una de las herramientas más importantes con las que cuenta un desarrollador de Ruby.

Las características anteriores son importantes para definir a Ruby como lenguaje, pero quizás la característica que es más visible y que hace de entrada que Ruby sea un lenguaje diferente es su sintaxis. Algunos definen que Ruby tiene algo llamado **Syntactic Sugar**, es un termino utilizado para definir que la sintaxis de Ruby no se interpone entre la idea y lo que se espera como resultado, es decir la sintaxis de Ruby nos ayuda a escribir código más conciso, expresivo y con menos líneas de código podemos escribir cosas complejas que en otros lenguajes se traduciría a algunas decenas de lineas de código.

Un efecto de la sintaxis de Ruby es la facilidad para escribir **DSL** o **Lenguajes de dominio específico**, es decir se puede escribir una nueva sintaxis encima de la sintaxis de Ruby que generalmente ahorra tiempo en ciertas tareas con Ruby, un ejemplo de este caso es **Ruby On Rails**, pero ya tocaremos ese tema más adelante en el libro.

## Intérpretes de Ruby

En la sección anterior se hizo referencia a que Ruby puede ser ejecutado a través de más de un intérprete.

### *MRI*

El intérprete oficial de Ruby ha sido **MRI** o **Matz Ruby Interpreter**, éste es el intérprete original que ha acompañado a Ruby desde su lanzamiento.

MRI está escrito en **C** y debido a esto también se le conoce como **CRuby**.

A través del tiempo MRI ha ido cambiando con el objetivo de mejorar el rendimiento de los programas escritos en Ruby, técnicamente el lenguaje como tal ha sufrido una serie de cambios sintácticos, pero los cambios más grande que ha sufrido MRI han sido en términos de su funcionamiento interno, por ejemplo, a partir de la versión 1.9 incluir la máquina virtual **YARV** y la mejora del algoritmo del **Garbage Collector**.

MRI es un intérprete portable, es decir, puede ejecutarse en distintas plataformas y sistemas operativos. Podemos ejecutar programas de Ruby con MRI en:

- Linux
- Unix
- Windows
- OSX
- Entre otra docena de sistemas operativos menos populares.

La versión más reciente de **MRI** es la versión 2.3.1, la cual apenas fue liberada el 26 de abril del 2016.

### *Rubinius*

Otro de los intérpretes que nos permiten ejecutar Ruby es Rubinius<sup>4</sup>. Rubinius fue creado en el 2006 por *Evan Phoenix*.

---

<sup>4</sup> <http://rubini.us/>

Rubinius tiene como objetivo el proveer una implementación alterna para ejecutar Ruby en ambientes concurrentes y de la forma más rápida posible.

El alto grado de compatibilidad que Rubinius mantiene con **MRI** es el reflejo de los desarrolladores de Rubinius de simplificar la migración del **MRI** a Rubinius.

Una de las características de Rubinius que puede ser atractiva para los desarrolladores de Ruby, es que mantiene la tradición de lenguajes como **LISP** y **SmallTalk** de implementar, hasta donde sea posible, Ruby en código de Ruby.

Como parte del proyecto de Rubinius, en el 2006 también se creo RubySpecs<sup>5</sup>, es decir la especificación ejecutable del lenguaje de Ruby. Ésta especificación es la que ayuda a medir y calificar el nivel en que un intérprete implementa la sintaxis de Ruby y las clases de la librería estándar de Ruby.

La versión más reciente disponible de Rubinius es la versión 2.0, versión que es compatible con la versión 2.0 de **MRI**.

## JRuby

JRuby<sup>6</sup> es el punto donde Java y Ruby se interceptan. JRuby nos permite ejecutar programas de Ruby encima de la máquina virtual de Java, con todos los beneficios es ésta ultima nos provee.

La creación de *Charles Nutter* y *Thomas Enebo*, busca ofrecer una opción más para un Ruby más rápido y mejor soporte a hilos.

Adicional a esto, JRuby permite que desde nuestros programas de Ruby podamos tener acceso a las librerías disponibles en el mundo Java, además de que es posible empotrar el intérprete de JRuby en programas de Java, proporcionándoles la capacidad de ejecutar **scripts** dinámicos.

En su versión más reciente, JRuby 1.7.5, ofrece soporte experimental compatible con la versión 2.0 de **MRI**.

---

<sup>5</sup> <http://rubyspec.org/>

<sup>6</sup> <http://jruby.org/>

## Iniciando con Ruby

Ya que tenemos un poco más de contexto sobre Ruby, vamos a aprender de su sintaxis y de la comunidad al rededor del lenguaje.

### *Instalando Ruby*

Los tres entornos más comunes para ejecutar Ruby son: Linux, OSX y Windows.

Dependiendo de la plataforma donde se desea ejecutar Ruby son las opciones que tenemos para instalarlo.

### Windows

En Windows la instalación es más sencilla gracias a RailsInstaller<sup>7</sup>, sólo es necesario descargar el instalador, seguir el asistente de instalación y en cuestión de minutos Ruby va a estar listo para utilizarse.

Para los ejemplos y ejercicios de éste libro necesitamos por lo mínimo Ruby 1.9.3; en RailsInstaller existe un instalador para Ruby 2.2.4 pero aún hay algunos detalles con ciertas librerías en Windows para ésta versión de Ruby.

### OSX

Para OSX es posible instalar Ruby con un instalador de RailsInstaller, aunque quizás la opción más práctica es instalarlo desde código fuente.

Para poder instalar Ruby 2.3.1 desde código fuente es necesario realizar las siguientes actividades - el paso 4 va a estar definido a detalle un poco más abajo :-

1. Instalar la última versión de *XCode*<sup>8</sup> disponible en el sitio de Apple.
2. Una vez instalado *XCode*, hay que abrirlo e ir a *Preferencias* y después a *Descargas*, ahí hay que instalar *Command Line Tools*.

---

<sup>7</sup> <http://railsinstaller.org/en>

<sup>8</sup> <https://developer.apple.com/xcode/downloads/>

3. Instalar *Homebrew*<sup>9</sup>, el cual es un manejador de paquetes para OSX.
4. Instalar Ruby.

En la terminal de OSX instalamos *rbenv*<sup>10</sup> y *ruby-build*<sup>11</sup> con la ayuda de *Homebrew*, ambos paquetes nos ayudan a instalar y manejar nuestra instalación de Ruby en OSX de una manera simple.

```
$ brew update
```

```
$ brew install rbenv
```

```
$ brew install ruby-build
```

```
$ rbenv install 2.3.1
```

```
$ rbenv rehash
```

```
$ rbenv global 2.3.1
```

El comando *rbenv install 2.3.1* se encarga de descargar, compilar e instalar Ruby, por lo que puede tardar varios minutos en ejecutarse. El último comando configura el ambiente para usar Ruby 2.3.1 por omisión.

Para mayor información en como funcionan *rbenv* y *ruby-build* hay que visitar sus repositorios en Github.

---

<sup>9</sup> <http://brew.sh/indexes.html>

<sup>10</sup> <https://github.com/sstephenson/rbenv>

<sup>11</sup> <https://github.com/sstephenson/ruby-build>

## *Linux*

Para los usuarios de Linux las instrucciones varían un poco entre las diferentes distribuciones y los sistemas de paquetes de cada una; obviamente va a ser muy difícil cubrirlas todas, por lo que las instrucciones siguientes va a estar basadas en la distribución Ubuntu Precise 12.4.

Los pasos para instalar Ruby de forma muy general se describen a continuación, el detalle de los mismos aparece un poco más abajo con los comandos a ejecutar en la terminal.

1. Actualizar nuestro sistema operativo
2. Instalar librerías requeridas para compilar Ruby
3. Instalar *rbenv* y *ruby-build*

```
$ sudo apt-get update

$ sudo apt-get install zlib1g-dev openssl libopenssl-ruby1.9.1 libssl-dev li-
bruby1.9.1 libreadline-dev git-core

$ cd ~

$ git clone git://github.com/sstephenson/rbenv.git .rbenv

$ echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.bashrc

$ echo 'eval "$(rbenv init -)"' >> ~/.bashrc

$ exec $SHELL

$ mkdir -p ~/.rbenv/plugins

$ cd ~/.rbenv/plugins
```

```
$ git clone git://github.com/sstephenson/ruby-build.git
```

```
$ cd ~
```

```
$ rbenv install 2.3.1
```

```
$ rbenv rehash
```

```
$ rbenv global 2.3.1
```

## Ruby instalado y listo para usarse

En este punto sin importar el sistema operativo en que estés trabajando, Ruby ya debe de estar listo para utilizarse y para comprobar que así es, en la terminal ejecutamos el siguiente comando, con el cual vamos a poder ejecutar la consola interactiva de Ruby:

```
$ irb
```

```
irb(main):001:0>
```

Si al ejecutarla se nos muestra un error, hay que revisar los pasos de instalación y volver a probar, si estamos en la consola, hay que teclear **exit** y presionamos la tecla de *enter* para salir.

## Estructura de Ruby

El lenguaje Ruby está estructurado en 3 grandes bloques, el **Core**<sup>12</sup>, las librerías estándar<sup>13</sup> y las gemas<sup>14</sup>.

---

<sup>12</sup> <http://www.ruby-doc.org/core-2.0.0/>

<sup>13</sup> <http://www.ruby-doc.org/stdlib-2.0.0/>

<sup>14</sup> <https://rubygems.org/>

## *Core*

El **Core** representa el fundamento del lenguaje en términos del interprete, máquina virtual, **garbage collector** y la sintaxis.

## *Librerías estándar*

El conjunto de librerías que extienden el funcionamiento de Ruby más allá de su sintaxis es definido por las librerías estándar.

## *Gemas*

Finalmente encontramos las Gemas o **Gems** que son librerías desarrolladas por la comunidad, *RubyGems* es el repositorio donde se publican las Gemas de Ruby. Actualmente *RubyGems* cuenta con alrededor de 64,500 gemas registradas.

# Conociendo la sintaxis de Ruby

En este punto ya estamos listos para poder empezar a conocer ciertos aspectos de la sintaxis de Ruby.

En ésta sección nos vamos a enfocar en aspectos que nos permitan iniciar de forma rápida con Ruby, pero implica que vamos a dejar otros aspectos fuera; revisar todo lo que podemos hacer con Ruby puede llevar todo un libro completo, dado que este libro no ésta enfocado en el lenguaje exclusivamente se justifica ésta decisión.

Para ejecutar los ejemplos de código utilizaremos *irb* que como ya vimos es la consola de modo interactivo de Ruby y que nos permite introducir código de Ruby y hacer que éste se evalúe al momento.

Iniciemos la consola interactiva desde la terminal de nuestro sistema operativo con:

```
$ irb
```

En cada evaluación el resultado de la misma es presentado en las siguientes líneas y aparece indicada con =>, símbolo que en Ruby se conoce como **Hash Rocket**.

```
irb(main):001:0> 1
```

```
=> 1
```

El **prompt** de la consola es `irb(main):001:0>`, cada vez que veamos una línea similar implica que la consola está esperando por nosotros para introducir alguna instrucción.

Si introducimos una sintaxis inválida o que lleve a una evaluación que produzca un error, obtendremos un mensaje como el mostrado a continuación, indicando el tipo de error, un mensaje y la línea de código que lo provocó.

```
irb(main):001:0> x
```

```
NameError: undefined local variable or method `x' for main:Object
```

```
from (irb):4
```

```
from /.rbenv/versions/2.1.2/bin/irb:12:in `<main>'
```

## Variables

Una variable en Ruby es representada por un identificador, el cual le da el nombre a la variable. El identificador debe de ser una nombre válido, por ejemplo no iniciar con un número o tener el nombre de una palabra reservada de Ruby.

Como convención en Ruby un nombre de variable siempre se escribe en minúsculas y si el nombre de la variable esta conformado por más de una palabra, cada palabra va separada por un guión bajo `_`, a esta forma de nombrar se le conoce como **snake case**.

De igual forma por convención todos los nombres de identificadores se escriben en inglés.

```
irb(main):001:0> name = 'Ruby'
```

```
=> "Ruby"
```

```
irb(main):002:0> quantity = 1
```

```
=> 1
```

Como vemos en el ejemplo, para asignar un valor a una variable simplemente usamos el signo de `=`, el cual precisamente implica asignación y se lee: Asigna el valor de la derecha a la variable de la izquierda.

Debido al sistema de tipos dinámicos de Ruby no es necesario especificar el tipo de dato de la variable, Ruby infiere el tipo de dato a partir del valor que se asigna a la variable.

Ésta característica de Ruby también permite redefinir una variable con un tipo de dato diferente.

```
irb(main):001:0> name = 'Ruby'
```

```
=> "Ruby"
```

```
irb(main):002:0> name = 1
```

```
=> 1
```

Es completamente válido en Ruby, aunque en términos prácticos esto puede llevar a problemas graves en un programa, por lo que los programadores de Ruby tratan de evitar redefinir variables a tipos de datos diferentes al original o a lo que el nombre de la variable implica.

Para imprimir el valor de una variable en la terminal simplemente escribirnos el nombre de la variable y presionamos enter.

```
irb(main):001:0> name = 'Ruby'
```

```
=> "Ruby"
```

```
irb(main):002:0> name
```

```
=> "Ruby"
```

## *Constantes*

Una constante, al igual que una variable, nos sirve para poder asignar un valor a un identificador que nos permita acceso rápido a éste valor. Pero a diferencia de una variable, una constante no cambia su valor durante la vida de un programa.

La convención de nombre para una constante es similar a la convención para una variable, con la única diferencia de el nombre de la constante se escribe en mayúsculas.

```
irb(main):001:0> LANGUAGE = 'Ruby'
```

```
=> "Ruby"
```

```
irb(main):002:0> LANGUAGE
```

```
=> "Ruby"
```

Mencionamos que el valor de una constante no cambia durante la vida de un programa, pero en realidad el lenguaje Ruby si permite modificar el valor de una constante, pero al modificar el valor **MRI** despliega una advertencia del cambio.

```
irb(main):001:0> LANGUAGE = 'Ruby'
```

```
=> "Ruby"
```

```
irb(main):002:0> LANGUAGE = 1
```

```
(irb):3: warning: already initialized constant LANGUAGE
```

```
(irb):1: warning: previous definition of LANGUAGE was here
```

```
=> 1
```

## Números

Ahora que ya sabemos como definir variables para guardar datos en Ruby, vamos conociendo que tipo de datos son los que podemos guardar.

En términos de valores numéricos en Ruby podemos trabajar con valores enteros y fraccionarios.

```
irb(main):001:0> 10
```

```
=> 10
```

```
irb(main):002:0> -30
```

```
=> -30
```

```
irb(main):003:0> 0
```

```
=> 0
```

```
irb(main):004:0> 774747474747474
```

```
=> 774747474747474
```

Los ejemplos anteriores pertenecen a la representación de números enteros positivos, negativos y el cero. En Ruby los valores enteros se representan con el tipo de datos **FixNum**.

Los números fraccionarios, que en Ruby se representan con el tipo de dato **float**.

```
irb(main):005:0> 3.1416
```

```
=> 3.1416
```

```
irb(main):006:0> -54.45
```

```
=> -54.45
```

```
irb(main):007:0> 0.0002
```

```
=> 0.0002
```

```
irb(main):008:0> 0.0
```

```
=> 0.0
```

Con los valores numéricos es posible realizar operaciones aritméticas, para tal efecto Ruby cuenta con una serie de operadores binarios:

```
irb(main):011:0> 2+5
```

```
=> 7
```

```
irb(main):012:0> 8-4
```

```
=> 4
```

```
irb(main):013:0> 3*6
```

```
=> 18
```

```
irb(main):014:0> 24/5
```

```
=> 4
```

```
irb(main):015:0> 24%5
```

```
=> 4
```

Para el caso de la división notamos que el consiente solo incluye la parte entera, y en la siguiente operación, módulo, vemos que el residuo de la misma division es diferente a cero, si nuestra intención en la division es mostrar el cociente con la parte decimal, entonces podemos replantear la division de la siguiente forma:

```
irb(main):016:0> 24/5.0
```

```
=> 4.8
```

Al forzar la division con uno de los valores como `float`, forzamos a que el resultado de la misma también sea `float`.

Como vimos en los ejemplos anteriores con Ruby es posible realizar operaciones aritméticas. Para realizar la evaluación de las operaciones, Ruby observa la procedencia de operadores.

```
irb(main):001:0> 1 + 2 * 3 / 4.0
```

```
=> 2.5
```

Si deseamos modificar el orden en como se evalúan las operaciones, entonces las agrupamos con paréntesis.

```
irb(main):001:0> (1 + 2) * (3 / 4.0)
```

```
=> 2.25
```

## Cadenas de texto

Ruby cuenta con un tipo de datos para almacenar cadenas de texto, éste tipo de dato es **String**. Una cadena de texto en Ruby puede contener caracteres invisibles, generalmente estos caracteres se conocen como caracteres de control y sirven, por ejemplo, para indicar saltos de línea.

Para expresar una cadena de texto podemos utilizar la comillas simple ' o la doble comillas ". Como veremos hay algunas diferencias entre utilizar una u otra.

```
irb(main):001:0> text = 'Hola Ruby'  
  
=> "Hola Ruby"
```

En el ejemplo anterior utilizamos la comillas simple para representar una cadena de texto. En el caso de éste tipo de comilla podemos **escapar** ciertos caracteres con la ayuda de \.

```
irb(main):001:0> text = 'Hola \\Ruby\''
```

```
=> "Hola 'Ruby'"
```

```
irb(main):002:0> text = 'Hola \\\Ruby\\\'
```

```
=> "Hola \\\Ruby\\\"
```

```
irb(main):003:0> puts text
```

```
Hola \\Ruby\
```

```
=> nil
```

En el último ejemplo hacemos uso de la instrucción **puts** para indicarle a Ruby de que deseamos imprimir el contenido de la variable **text**, con **puts** se interpreta correctamente el **escape** de \\.

Con el uso de las comillas dobles podemos **escapar** una mayor cantidad de caracteres, incluyendo los caracteres de control.

```
irb(main):001:0> text = "Hola Ruby"  
  
=> "Hola Ruby"  
  
irb(main):002:0> text = "Hola \"Ruby\""  
  
=> "Hola \"Ruby\""  
  
irb(main):003:0> puts text  
  
Hola "Ruby"  
  
=> nil  
  
irb(main):004:0> text = "Hola \\Ruby\\\""  
  
=> "Hola \\Ruby\\\""  
  
irb(main):005:0> puts text  
  
Hola \\Ruby\\\"  
  
=> nil  
  
irb(main):006:0> text = "Hola Ruby\n\n"
```

```
=> "Hola Ruby\n\n"
```

```
irb(main):007:0> puts text
```

```
Hola Ruby
```

```
=> nil
```

En el último ejemplo vemos como al usar la instrucción `puts` para desplegar el contenido de la variable el carácter de control `\n` es interpretado correctamente como **salto de línea**.

Otra diferencia importante entre utilizar la comillas simple y la comillas doble en cadenas de texto es la interpolación. Veamos el ejemplo donde tenemos una cadena de texto, pero queremos formarla a partir de un valor declarado en otra variable.

```
irb(main):001:0> days = 2
```

```
=> 2
```

```
irb(main):002:0> text = "Han pasado " + days.to_s + " días"
```

```
=> "Han pasado 2 días"
```

Hay que notar que para poder utilizar el valor de la variable `days` al concatenar las cadenas de texto, tuvimos que hacer una conversión del valor numérico a cadena de texto, esto lo logramos con la llamada al método `to_s`.

Gracias a la interpolación en Ruby, podemos hacer que éste tipo de tareas sea más sencilla.

```
irb(main):001:0> days = 2
```

```
=> 2
```

```
irb(main):002:0> text = "Han pasado #{days} días"
```

```
=> "Han pasado 2 días"
```

```
irb(main):003:0> text = "Han pasado #{days * 2} días"
```

```
=> "Han pasado 4 días"
```

Como vemos en los ejemplos, con el uso de `#{...}` podemos interpolar un valor proveniente de otra variable en una cadena de texto. En el último ejemplo inclusive podemos ver que el valor a interpolar puede ser código normal de Ruby, el cual será interpretado e interpolado en la cadena.

En Ruby es posible que nosotros indiquemos el carácter delimitador para una cadena de texto, para éste caso contamos con las literales `%q` y `%Q` que tienen la funcionalidad de la comilla simple y la comilla doble respectivamente.

```
irb(main):001:0> texto = %q{Hola 'Ruby'}
```

```
=> "Hola 'Ruby'"
```

```
irb(main):002:0> texto = %q!Hola 'Ruby'!
```

```
=> "Hola 'Ruby'"
```

```
irb(main):003:0> texto = %Q!Hola 'Ruby'!
```

```
=> "Hola 'Ruby'"
```

```
irb(main):004:0> texto = %Q!Hola #{nombre}!
=> "Hola Ruby"

irb(main):005:0> texto = %Q!Hola #{nombre * 3}!
=> "Hola RubyRubyRuby"
```

Otra forma de declarar cadenas de texto con Ruby es mediante el uso de **here document**, ésta modalidad nos permite declarar una cadena de texto de múltiples líneas sin la necesidad de usar caracteres de escape.

```
irb(main):001:0> text = <<_DOC_
irb(main):002:0" Hola Ruby

irb(main):003:0" Esta es una declaración de

irb(main):004:0" cadena de texto multilínea

irb(main):005:0" _DOC_
=> "Hola Ruby\nEsta es una declaración de\ncadena de texto multilínea\n"

irb(main):006:0> puts text

Hola Ruby

Esta es una declaración de
```

cadena de texto multilínea

=> nil

El delimitador **\_DOC\_** puede ser cualquier identificador, por ejemplo **\_MSG\_**.

### *En Ruby casi todo es un objeto*

Hasta éste momento hemos visto como trabajar con tipos de datos básicos como números y cadenas de texto, pero en Ruby tienen un trato diferente, ya que en Ruby se tiene la noción de que **casi** todo es un objeto.

Ésta aserción es relativamente fácil de comprobar.

```
irb(main):001:0> 4.class
```

=> Fixnum

```
irb(main):002:0> "Hola Ruby".class
```

=> String

Con el ejemplo anterior podemos ver que podemos enviar mensajes a éstos tipos de datos primitivos. La forma de enviar el mensaje al dato primitivo nos puede ser familiar, ya que la notación es la misma que en la mayoría de los lenguajes orientados a objetos, en donde usamos el **.** y acto seguido el nombre del mensaje.

En Ruby podemos consultar que mensajes podemos enviar a un objeto, veamos el caso de los mensajes disponibles para el valor **4**.

```
irb(main):001:0> 4.methods
```

```
=> [:to_s, :-, :+, :-, :*, :/, :div, :%,
     :modulo, :divmod, :fdiv, :**, :abs, :magnitude, :==, :==|,
     :<=>, :>, :>=, :<, :<=, :~, :&, :|, :^, :
```

```
[], :<<, :>>, :to_f, :size, :zero?, :odd?, :even?, :succ, :integer?, :upto, :down
to, :times, :next, :pred, :chr, :ord, :to_i, :to_int, :floor, :ceil, :truncate, :
round, :gcd, :lcm, :gcdlcm, :numerator, :denominator, :to_r, :rationalize, :singl
eton_method_added, :coerce, :i, :
+@, :eql?, :quo, :remainder, :real?, :nonzero?, :step, :to_c, :real, :imaginary,
:imag, :abs2, :arg, :angle, :phase, :rectangular, :rect, :polar, :conjugate, :con
j, :between?, :nil?, :=, :hash, :class, :singleton_class, :clone, :dup, :initiali
ze_dup, :initialize_clone, :taint, :tainted?, :untaint, :untrust, :untrusted?, :t
rust, :freeze, :frozen?, :inspect, :methods, :singleton_methods, :protected_metho
ds, :private_methods, :public_methods, :instance_variables, :instance_variable_ge
t, :instance_variable_set, :instance_variable_defined?, :instance_of?, :kind_of?,
:is_a?, :tap, :send, :public_send, :respond_to?, :respond_to_missing?, :extend, :
display, :method, :public_method, :define_singleton_method, :object_id, :to_enum,
:enum_for, :equal?, :!, :!=, :instance_eval, :instance_exec, :__send__, :__id__]
```

En el listado de los mensajes disponibles para el tipo de dato **FixNum** vemos que los mensajes aparecen con una notación especial que antepone el **:** al nombre del mensaje, ésta notación en Ruby se conoce como **Símbolo**, de momento es suficiente con saber como se llama ésta notación, más adelante veremos más detalle sobre los **Símbolos**.

Veamos algunos ejemplo de los mensajes que podemos enviar a un tipo de datos **FixNum**.

```
irb(main):001:0> 4.even?
```

```
=> true
```

```
irb(main):002:0> 4.next
```

```
=> 5
```

```
irb(main):003:0> 4.between? 3, 6
```

```
=> true
```

## Símbolos

Los símbolos en Ruby son una característica un poco extraña, principalmente si solo se tiene experiencia con lenguajes basados en C. Los símbolos se utilizan de manera intensiva dentro de los programas en Ruby.

Los siguientes ejemplos corresponden a símbolos en Ruby:

```
irb(main):001:0> :hola_ruby
```

```
=> :hola_ruby
```

```
irb(main):002:0> :hola_ruby.class
```

```
=> Symbol
```

```
irb(main):001:0> :"Hola Ruby"
```

```
=> :"Hola Ruby"
```

```
irb(main):002:0> :"Hola Ruby".class
```

```
=> Symbol
```

En ambos casos la clase es `Symbol` sin importar la diferencia de escribirlos con " o sin ella, aunque la primera notación en forma `snake_case` es la más común.

Un símbolo es realmente una cadena de texto, pero inmutable, es decir no se puede modificar el contenido como se podría hacer con una cadena de texto convencional.

```
irb(main):001:0> 'Hola Ruby' << ' !!!!'
```

```
=> "Hola Ruby !!!!"

irb(main):002:0> :"Hola Ruby" << ' !!!!'
NoMethodError: undefined method `<<' for :"Hola Ruby":Symbol

from (irb):2

from from /.rbenv/versions/2.1.2/bin/irb:12:in `<main>'
```

En términos de un programa en Ruby, un símbolo generalmente representa una idea abstracta en lugar de representar un valor. Debido a esto Ruby trata de forma diferente a un símbolo en comparación a una cadena de texto. Como una cadena de texto es mutable, Ruby necesita guardar en memoria una copia de su contenido cada vez que se utiliza dicha cadena de texto.

```
irb(main):001:0> "hola mundo".object_id
=> 70144093356720

irb(main):002:0> "hola mundo".object_id
=> 70144093330220
```

El `object_id` de un objeto en Ruby, podemos pensar como una forma de representar su ubicación en memoria, aunque esto no es del todo cierto ya que no indica su posición en la memoria física.

En el ejemplo anterior podemos ver como al utilizar la cadena cadena de texto, Ruby crea una copia por cada cadena de texto, esto lo podemos observar con solo ver el numero que retorna el envío del mensaje `object_id`.

Ahora veamos el mismo ejemplo, pero para símbolos.

```
irb(main):001:0> :hola_ruby.object_id
```

```
=> 457768
```

```
irb(main):002:0> :hola_ruby.object_id
```

```
=> 457768
```

Ahora el resultado del envío del mensaje `object_id` para ambos casos nos retorna el mismo valor, debido a esto podemos asumir que no importa cuantas ocasiones declaremos el mismo símbolo Ruby no crea copias del mismo, en su lugar los reutiliza.

Esto se debe a que los símbolos son almacenados en otra parte de memoria, donde el `garbage collector` no recicla el espacio de memoria de los símbolos; esto quiere decir que una vez que declaramos un símbolo, este permanece en memoria durante la vida del programa.

Podemos acceder a la estructura donde se almacenan los símbolos mediante el objeto `Symbol`.

```
irb(main):001:0> Symbol.all_symbols.size
```

```
=> 2934
```

```
irb(main):002:0> Symbol.all_symbols[1..30]
```

```
=>
```

```
[:<IFUNC>, :<CFUNC>, :respond_to?, :"core#set_method_alias", :"core#set_variable_alias", :"core#undef_method", :"core#define_method", :"core#define_singleton_method", :"core#set_postexe", :each, :length, :size, :lambda, :intern, :gets, :succ, :method_missing, :send, :__send__, :initialize, :_, :__autoload__, :__classpath__, :__tmp_classpath__, :__classid__, :__attached__, :BasicObject, :Object, :Module, :Class]
```

Es posible realizar conversiones entre un símbolo y una cadena de texto y viceversa, en el caso de los símbolos, estos puede recibir el mensaje `to_s` y de esa forma transformar a cadena de tex-

to, de igual forma una cadena de texto puede recibir el mensaje `to_sym` para obtener su representación en forma de símbolo.

```
irb(main):001:0> 'hola ruby'.to_sym
```

```
=> :"hola ruby"
```

```
irb(main):002:0> :hola_ruby.to_s
```

```
=> "hola_ruby"
```

## Arreglos

Ruby cuenta con estructuras de datos que nos permiten almacenar información que tiene cierta relación entre sí. Un ejemplo de éste tipo de estructuras es el Arreglo o `Array`.

Un `Array` puede almacenar una lista de elementos homogéneos, es decir que pueden ser de tipos base diferentes, los cuales pueden ser referenciados por un índice, el cual representa su posición dentro de la lista. El índice de elementos de un `Array` inicia en la posición 0 - cero -.

Para representar un `Array` en Ruby, utilizamos los delimitadores `[` y `]` y cada elemento contenido está separado por una `,`.

```
irb(main):001:0> values = [1, 2, 3, 4]
```

```
=> [1, 2, 3, 4]
```

```
irb(main):002:0> values.class
```

```
=> Array
```

Para acceder a unos de los elementos del `Array` solo pasamos su posición con el envío del mensaje `[]`.

```
irb(main):001:0> values = [1, 2, 3, 4]  
=> [1, 2, 3, 4]
```

```
irb(main):002:0> values[0]
```

```
=> 1
```

```
irb(main):003:0> values[2]
```

```
=> 3
```

En Ruby, a diferencia de otros lenguajes, si pasamos una posición de índice donde no existe elemento no obtenemos un error de que el índice está fuera de rango, en su lugar simplemente obtenemos un `nil`

```
irb(main):001:0> values = [1, 2, 3, 4]  
=> [1, 2, 3, 4]
```

```
irb(main):002:0> values[10]
```

```
=> nil
```

En el caso de que pasemos una posición de índice negativa, entonces Ruby va a leer los elementos de derecha a izquierda, siendo el elemento más a la derecha el de posición `-1`.

```
irb(main):001:0> values = [1, 2, 3, 4]  
=> [1, 2, 3, 4]
```

```
irb(main):002:0> values[-1]
```

```
=> 4
```

```
irb(main):002:0> values[-2]
```

```
=> 3
```

```
irb(main):002:0> values[-5]
```

```
=> nil
```

Si lo queremos es obtener de un **Array** un grupo de elementos contiguos, entonces podemos indicar un Rango o **Range** de elementos, en lugar de pasar solo el indice de una posición.

```
irb(main):001:0> values = [1, 2, 3, 4]
```

```
=> [1, 2, 3, 4]
```

```
irb(main):002:0> values[1..3]
```

```
=> [2, 3, 4]
```

Para agregar nuevos elementos al final un **Array** podemos utilizar el operador **<<** o el mensaje **push**.

```
irb(main):001:0> values = [1, 2, 3, 4]
```

```
=> [1, 2, 3, 4]
```

```
irb(main):002:0> values << 5
```

```
=> [1, 2, 3, 4, 5]
```

```
irb(main):003:0> values
```

```
=> [1, 2, 3, 4, 5]
```

```
irb(main):004:0> values.push 6
```

```
=> [1, 2, 3, 4, 5, 6]
```

```
irb(main):005:0> values
```

```
=> [1, 2, 3, 4, 5, 6]
```

Pero si lo que queremos es agregar un elemento al principio del **Array**, entonces podemos hacer uso del mensaje **unshift**.

```
irb(main):001:0> values = [1, 2, 3, 4]
```

```
=> [1, 2, 3, 4]
```

```
irb(main):002:0> values.unshift 0
```

```
=> [0, 1, 2, 3, 4]
```

```
irb(main):003:0> values
```

```
=> [0, 1, 2, 3, 4]
```

A un **Array** es posible realizarle las operaciones aritméticas de **+** y **\***.

```
irb(main):001:0> values = [1, 2, 3, 4]  
=> [1, 2, 3, 4]
```

```
irb(main):002:0> values + values  
=> [1, 2, 3, 4, 1, 2, 3, 4]
```

```
irb(main):003:0> values * 3  
=> [1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
```

Un **Array** en Ruby es un objeto, por lo tanto puede recibir una serie de mensajes que nos permiten realizar operaciones interesantes, podemos observar que mensajes soporta un **Array** con el envío del mensaje 'methods' que ya vimos anteriormente.

```
irb(main):001:0> values = [1, 2, 3, 4]  
=> [1, 2, 3, 4]
```

```
irb(main):002:0> values.rotate  
=> [2, 3, 4, 1]
```

```
irb(main):003:0> values.empty?  
=> false
```

```
irb(main):004:0> values.shuffle
```

```
=> [2, 3, 1, 4]
```

```
irb(main):005:0> values.join ', '
```

```
=> "1, 2, 3, 4"
```

```
irb(main):006:0> values.include? 3
```

```
=> true
```

Anteriormente en ésta sección de [Array](#) se mencionó el concepto de Rango para seleccionar un subconjunto de elementos de un [Array](#); volveremos a utilizar un Rango para crear un [Array](#) que contenga un conjunto de elementos contiguos.

```
irb(main):001:0> (1..10).to_a
```

```
=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
irb(main):002:0> ('c'..'m').to_a
```

```
=> ["c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m"]
```

## Diccionarios

Otra de las estructuras de las que disponemos en Ruby son los diccionarios o simplemente [Hash](#). Ésta estructura nos permite contener pares de elementos relacionados, donde una parte del par representa una llave y la segunda parte representa un valor.

En un [Hash](#) la llave nos sirve como elemento para acceder al valor con el que está relacionada.

```
irb(main):001:0> numbers = { 'uno' => 1, 'dos' => 2, 'tres' => 3 }
```

```
=> {"uno"=>1, "dos"=>2, "tres"=>3}
```

```
irb(main):002:0> numbers['dos']
```

```
=> 2
```

Como vemos en el ejemplo anterior los elementos de un **Hash** se delimitan por **{** y **}**, los elementos van separados por una **,** y el par se presenta con la llave a la izquierda y el valor a la derecha del **=>** – **Hashrocket** –.

En un **Hash** no pueden haber 2 elementos con misma llave, pero si pueden haber 2 elementos con el mismo valor.

```
irb(main):001:0> numbers = { 'uno' => 1, 'dos' => 2, 'tres' => 3 }
```

```
=> {"uno"=>1, "dos"=>2, "tres"=>3}
```

```
irb(main):002:0> numbers['tres'] = 2
```

```
=> 2
```

```
irb(main):003:0> numbers
```

```
=> {"uno"=>1, "dos"=>2, "tres"=>2}
```

En Ruby la forma más común de representar la llave de un elemento es mediante el uso de símbolos.

```
irb(main):001:0> numbers = { :uno => 1, :dos => 2, :tres => 3 }
```

```
=> {:uno=>1, :dos=>2, :tres=>3}
```

```
irb(main):002:0> numbers[:dos]
```

```
=> 2
```

A partir de la versión 1.9 de Ruby la sintaxis de un **Hash** se puede representar sin el uso del `=>`.

```
irb(main):001:0> numbers = { uno: 1, dos: 2, tres: 3 }
```

```
=> {:uno=>1, :dos=>2, :tres=>3}
```

```
irb(main):002:0> numbers[:dos]
```

```
=> 2
```

Un **Hash** puede contener como valor un tipo de dato primitivo, como una cadena de texto o un entero, o bien un **Array** u otro **Hash**.

Finalmente, y no creo que sea sorpresa, un **Hash** es un objeto que puede recibir una numero de mensajes para realizar algunas operaciones con los datos que contiene.

```
irb(main):001:0> numbers = { uno: 1, dos: 2, tres: 3 }
```

```
=> {:uno=>1, :dos=>2, :tres=>3}
```

```
irb(main):002:0> numbers.has_key? :cuatro
```

```
=> false
```

```
irb(main):003:0> numbers.has_value? 1
```

```
=> true
```

```
irb(main):004:0> numbers.keys
```

```
=> [:uno, :dos, :tres]
```

```
irb(main):005:0> numbers.values
```

```
=> [1, 2, 3]
```

## *Métodos en Ruby*

Hasta este momento se ha hecho referencia a que los objetos reciben mensajes, en el espíritu de la definición de [Alan Kay](#), pero estos mensajes también se les conoce como **métodos**, nombre con el cual nos referiremos de ahora en adelante en el libro.

En Ruby al iniciar una terminal interactiva (IRB), automáticamente se crea un objeto raíz **main**, el cual es la instancia receptora cuando se define algún objeto dentro de Ruby, es fácil comprobar la existencia de este objeto, simplemente tenemos que iniciar una sesión en la terminal y ejecutar:

```
irb(main):001:0> self
```

```
=> main
```

```
irb(main):002:0> self.class
```

```
=> Object
```

Debido a esto, podemos definir métodos al objeto **main** sin la necesidad definir una clase - El tema de clases lo visitaremos un poco mas adelante -.

```
irb(main):001:0> def hello
```

```
irb(main):002:1> puts 'Hola mundo'
```

```
irb(main):003:1> end
```

```
=> :hello
```

```
irb(main):004:0> hello
```

```
Hola mundo
```

```
=> nil
```

La palabra clave **def** seguida de un nombre valido define un método con ese nombre, los nombres de métodos al igual que las variables utilizan la convención de **snake\_case**.

El cuerpo del método contiene el código de Ruby a ejecutar, por convención el cuerpo se identa 2 espacios a la derecha, el final del cuerpo del método se marca con la palabra clave **end**.

Para ejecutar el método simplemente introducimos el nombre en la consola y obtenemos el resultado esperado.

A diferencia de otros lenguajes no hay que especificar el tipo de datos que un método va a retornar o declararlas como **void**.

En Ruby la última evaluación que se realiza en el cuerpo de un método es el resultado que se retorna, es por eso que en el ejemplo anterior la ejecución del método **hello** retorna un **nil**.

En el siguiente ejemplo se puede apreciar que la evaluación de la última línea del cuerpo del método es el valor que se retorna.

```
irb(main):001:0> def circunference
```

```
irb(main):002:1>   ratio = 2
```

```
irb(main):003:1>   pi = 3.1416
```

```
irb(main):004:1> pi * (ratio * ratio)
```

```
irb(main):005:1> end
```

```
=> :circunference
```

```
irb(main):006:0> circumference
```

```
=> 12.5664
```

En Ruby existe la palabra clave `return` pero como vimos en el ejemplo, no es necesario que la utilicemos de manera explícita, a menos de que quisiéramos terminar la ejecución de un método antes de que llegue al final del cuerpo.

```
irb(main):001:0> def circumference
```

```
irb(main):002:1> ratio = 2
```

```
irb(main):003:1> pi = 3.1416
```

```
irb(main):004:1> return pi * (ratio * ratio)
```

```
irb(main):005:1> end
```

```
=> :circunference
```

```
irb(main):006:0> circumference
```

```
=> 12.5664
```

Un método que acepta parámetros puede ser un poco más interesante.

```
irb(main):001:0> def triangle_area(base, height)
```

```
irb(main):002:1>   (base * height) / 2.0
```

```
irb(main):003:1> end
```

```
=> :triangle_area
```

```
irb(main):004:0> triangle_area 3.0, 5.0
```

```
=> 15.0
```

```
irb(main):005:0> triangle_area
```

```
ArgumentError: wrong number of arguments (0 for 2)
```

```
from (irb):1:in `triangle_area'
```

```
from (irb):5
```

```
from /.rbenv/versions/2.1.2/bin/irb:12:in `<main>'
```

Como vemos en la ejecución de nuestra función, no es necesario especificar los paréntesis para indicar cuales son nuestros parámetros, solo son requeridos cuando queremos eliminar ambigüedad.

Si llamamos nuestra función sin pasar alguno o todos los parámetros obtenemos el error de "**wrong number of arguments (X for Y)**". Si queremos que alguno de esos parámetros sean opcionales entonces en la declaración debemos de asignarle un valor en caso de omisión.

```

irb(main):001:0> def triangle_area(base = 1, altura = 1)

irb(main):002:1>   (base * altura) / 2.0

irb(main):003:1> end

=> :triangle_area

irb(main):004:0> triangle_area

=> 0.5

irb(main):005:0> triangle_area 2

=> 1.0

irb(main):006:0> triangle_area 2, 3

=> 3.0

```

Los parámetros se pasan en el orden en que están definidos en la declaración de la función, pero en Ruby y en algunas librerías vamos a notar que estas aceptan parámetros con nombre; esto se logra aceptando solamente un parámetro como **Hash** y extrayendo los valores a variables independientes:

```

irb(main):001:0> def triangle_area(params = {})

irb(main):002:1>   base = params[:base] || 1

irb(main):003:1>   altura = params[:altura] || 1

```

```

irb(main):004:1>

irb(main):005:1*   (base * altura) / 2.0

irb(main):006:1> end

=> :triangle_area

irb(main):007:0> triangle_area altura: 2, base: 4

=> 4.0

```

*Nota: El operador evalúa el resultado de la izquierda primero y si es nil entonces evalúa el resultado de la derecha.*

En este caso el **Hash** nos permite simular parámetros con nombre, por lo tanto podemos pasarlos en cualquier orden a la función, también es importante notar que anteriormente mencionamos que para declarar un **Hash** se tenía que realizar con las llaves **{ }**, pero en el caso de pasar un **Hash** como parámetro a una función las llaves no son necesarias.

En Ruby 2.0 los parámetros con nombre han sido incluidos como parte de la funcionalidad, por lo tanto podemos reescribir la misma función con sintaxis de Ruby 2.0 como sigue:

```

irb(main):001:0> def triangle_area(base: 1, altura: 1)

irb(main):002:1>   (base * altura) / 2.0

irb(main):003:1> end

=> :triangle_area

irb(main):004:0> triangle_area altura: 2, base: 5

```

```
=> 5.0
```

Ya vimos varias combinaciones para definir parámetros a funciones en Ruby, esta ultima variación nos va a permitir recibir un numero ilimitado de parámetros, para tal efecto utilizamos el operador **splat**; este operador tiene solo una restricción, debe de ser el ultimo parámetro en la definición de una función.

```
irb(main):001:0> def multi_params(*params)
irb(main):002:1>   puts "Parametro 1 #{params[0]}"
irb(main):003:1>   puts "Parametro 2 #{params[1]}"
irb(main):004:1>   puts "Parametro 3 #{params[2]}"
irb(main):005:1>   puts "Parametro 4 #{params[3]}"
irb(main):006:1>   puts "Parametro 5 #{params[4]}"
irb(main):007:1> end
=> :multi_params

irb(main):008:0> multi_params 'a', 'b', 'c', 1, 2
Parametro 1 a
Parametro 2 b
Parametro 3 c
```

Parametro 4 1

Parametro 5 2

=> nil

Ruby permite múltiple asignación, podemos aprovechar esa funcionalidad para acceder a los parámetros múltiples desde dentro de la función en variables independientes.

```
irb(main):001:0> def multi_params(*params)
irb(main):002:1>   a, b, c, d, e, f = params
irb(main):003:1>
irb(main):004:1*>   puts "Parametro 1 #{a}"
irb(main):005:1>   puts "Parametro 2 #{b}"
irb(main):006:1>   puts "Parametro 3 #{c}"
irb(main):007:1>   puts "Parametro 4 #{d}"
irb(main):008:1>   puts "Parametro 5 #{e}"
irb(main):009:1> end
=> :multi_params
```

```
irb(main):010:0> multi_params 'a', 'b', 'c', 1, 2
```

Parametro 1 a

Parametro 2 b

Parametro 3 c

Parametro 4 1

Parametro 5 2

=> nil

La múltiple asignación puede ser parcial, es decir no necesitamos proveer una variable para cada parámetro que reciba la función, para esto definimos una de las variables como splat.

```
irb(main):001:0> def multi_params(*params)
```

```
irb(main):002:1>   a, b, *c = params
```

```
irb(main):003:1>
```

```
irb(main):004:1*>   puts "Parametro 1 #{a}"
```

```
irb(main):005:1>   puts "Parametro 2 #{b}"
```

```
irb(main):006:1>   puts "Resto de parametros #{c}"
```

```
irb(main):007:1> end  
  
=> :multi_params  
  
irb(main):008:0> multi_params 'a', 'b', 'c', 1, 2  
  
Parametro 1 a  
  
Parametro 2 b  
  
Resto de parametros ["c", 1, 2]  
  
=> nil
```

Como mencionamos anteriormente el operador **splat** para los parámetros debe de ser el último que se defina, en caso de que deseemos recibamos mas parámetros en la función entonces declaramos la función como sigue.

```
irb(main):001:0> def operacion(operador, *params)  
  
irb(main):002:1>   params.inject(operador)  
  
irb(main):003:1> end  
  
=> :operacion  
  
irb(main):004:0> operacion :+, 1, 2, 3, 4, 5  
  
=> 15
```

```
irb(main):005:0> operacion :-, 1, 2, 3, 4, 5
```

```
=> -13
```

```
irb(main):006:0> operacion :*, 1, 2, 3, 4, 5
```

```
=> 120
```

*Nota: La función inject es una función disponible para arreglos.*

## Bloques, Procs y Lambdas

Hasta este momento hemos visto como definir funciones por nombre y como ejecutarlas, ahora vamos a ver una de las características que hacen a Ruby ser ... Ruby.

Iniciaremos con los bloques, ya que son los más comunes en Ruby, un bloque nos permite definir funciones anónimas las cuales mantienen referencias al entorno donde fueron definidas, es decir pueden acceder a variables que posiblemente ya no estén disponibles en el entorno de ejecución - **scope** -; a este tipo de funcionalidad se le conoce como **Closures**; esta es una característica básica de los lenguajes funcionales.

Vamos a ver en términos prácticos que es un bloque en Ruby.

```
irb(main):001:0> 10.times do |i|
```

```
irb(main):002:1*>   puts "El valor es #{i}"
```

```
irb(main):003:1> end
```

```
El valor es 0
```

```
El valor es 1
```

El valor es 2

El valor es 3

El valor es 4

El valor es 5

El valor es 6

El valor es 7

El valor es 8

El valor es 9

En este caso la función `times` un iterador el cual se describe como **iniciando desde 0 para cada ciclo guarda el valor en i e incrementa su valor en 1 hasta que i sea menor a 10**, pero esta función acepta un parámetro especial, un bloque, este bloque esta delimitado por la palabra clave `do`, y a continuación se enumeran los parámetros de este bloque con las barras `||`, y a continuación se escribe el cuerpo de la función y se indica la terminación de la función anónima con la palabra clave `end`.

El código que se encuentra entre `do` y `end` es una función que a diferencia de las funciones que vimos anteriormente no cuenta con un nombre por lo tanto no puede llamarse en otras partes de nuestro programa.

Los bloques en Ruby tienen una sintaxis alterna que no modifica la forma en como funcionan.

```
irb(main):001:0> 10.times { |i| puts "El valor es #{i}"}
```

El valor es 0

El valor es 1

El valor es 2

El valor es 3

El valor es 4

El valor es 5

El valor es 6

El valor es 7

El valor es 8

El valor es 9

Anteriormente en la definición de **Closure** se indicó que aun bloque puede mantener referencias a las variables definidas en su entorno y por lo tanto puede hacer uso de ellas, el siguiente ejemplo muestra este caso.

```
irb(main):001:0> factor = 2
```

```
=> 2
```

```
irb(main):002:0> 10.times do |i|
```

```
irb(main):003:1*>   puts "El valor es #{factor * i}"
```

```
irb(main):004:1> end
```

El valor es 0

El valor es 2

El valor es 4

El valor es 6

El valor es 8

El valor es 10

El valor es 12

El valor es 14

El valor es 16

El valor es 18

Aquí podemos ver como la variable **factor** fue declarada fuera del cuerpo del bloque y aún así es posible utilizarla dentro de la definición del mismo.

Las clases que implementan el modulo **Enumerable**<sup>15</sup>, son unas de las que más explotan esta funcionalidad de bloques, como ejemplo tenemos a la clase **Array**.

```
irb(main):001:0> (1..20).select{|i| i.odd?}
```

---

<sup>15</sup> <http://ruby-doc.org/core-2.1.0/Enumerable.html>

```
=> [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

```
irb(main):002:0> (1..20).select{|i| i.even?}
```

```
=> [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

```
irb(main):003:0> (1..20).select{|i| i%3 == 0}
```

```
=> [3, 6, 9, 12, 15, 18]
```

```
irb(main):004:0> (1..20).detect{|i| i == 6}
```

```
=> 6
```

```
irb(main):005:0> (1..20).detect{|i| i == 60}
```

```
=> nil
```

Ahora que conocemos que un bloque veamos como podemos implementar una función que reciba un bloque como parámetro.

```
irb(main):001:0> def time
```

```
irb(main):002:1>   start = Time.now
```

```
irb(main):003:1>   result = yield
```

```
irb(main):004:1>   puts "Completado en #{Time.now - start}"
```

```
irb(main):005:1> end  
  
=> :time  
  
irb(main):006:0>  
  
irb(main):007:0*> time do  
  
irb(main):008:1*>   sleep 5  
  
irb(main):009:1> end  
  
=> Completado en 5.00021
```

Vemos que no tenemos que realizar una declaración especial para que una función acepte un bloque como parámetro, pero dentro de la función para poder ejecutar el código del bloque necesitamos de realizar una llamada a la función `yield`.

Ahora si ejecutamos la misma función sin pasar un bloque vamos a obtener un error.

```
irb(main):001:0> time  
  
LocalJumpError: no block given (yield)  
  
from (irb):9:in `time'  
  
from (irb):16  
  
from /.rbenv/versions/2.1.1/bin/irb:01:in `<main>'
```

El error indica que la función `time` intentó ejecutar la llamada a la función `yield` pero no pasamos un bloque para ser ejecutado. Esto es sencillo de solucionar.

```

irb(main):001:0> def time

irb(main):002:1>   start = Time.now

irb(main):003:1>   result = yield if block_given?

irb(main):004:1>   puts "Completado en #{Time.now - start}"

irb(main):005:1> end

=> :time

irb(main):006:0> time

=> Completado en 5.0e-06

```

Con la verificación de `block_given?` podemos verificar si la función recibió un bloque como parámetro o no y entonces realizar la llamada a `yield` si es pertinente.

Otra forma de indicar en la firma de nuestra función de forma más explícita que podemos aceptar un bloque es definiendo un parámetro para este propósito. La única restricción es que este parámetro debe ser el último definido en la firma de la función, anteriormente cuando hablamos del parámetro de tipo `splat` habíamos dicho que debe ser el último parámetro lo cual es cierto siempre y cuando no decidimos aceptar un bloque.

```

irb(main):001:0> def calculate_numbers(*numbers, &block)

irb(main):002:1>   new_numbers = []

irb(main):003:1>

```

```

irb(main):004:1*   numbers.each do |number|


irb(main):005:2*     new_numbers << block.call(number)

irb(main):006:2>   end

irb(main):007:1>

irb(main):008:1*   new_numbers

irb(main):009:1> end

=> :calculate_numbers

irb(main):010:0>

irb(main):011:0* calculate_numbers 1, 2, 3, 4 do |num|


irb(main):012:1*   num * 2

irb(main):013:1> end

=> [2, 4, 6, 8]

```

En el ejemplo anterior el parámetro `&block` es quien se encarga de recibir nuestro bloque, y vemos que para poder ejecutarlo es necesario llamar al método `call`. También podemos observar que en la llamada `call` podemos pasar parámetros que nuestra función anónima espera recibir.

En este punto muy probablemente ya nos dimos cuenta que la gran diferencia entre pasar un bloque a una función que no declara específicamente un parámetro para este propósito y una fun-

ción que si lo hace, es que en el caso del último ejemplo `block` es un objeto. Si modificamos un poco nuestro código podemos confirmar que `block` es un objeto de tipo `Proc`.

```
irb(main):001:0> def calculate_numbers(*numbers, &block)
irb(main):002:1>   new_numbers = []
irb(main):003:1>
irb(main):004:1*>   puts block.class
irb(main):005:1>   numbers.each do |number|
irb(main):006:2*>     new_numbers << block.call(number)
irb(main):007:2>   end
irb(main):008:1>
irb(main):009:1*>   new_numbers
irb(main):010:1> end
=> :calculate_numbers
irb(main):011:0>
irb(main):012:0*> calculate_numbers 1, 2, 3, 4 do |num|
```

```
irb(main):013:1* num * 2
```

```
irb(main):014:1> end
```

Proc

```
=> [2, 4, 6, 8]
```

Esta es la segunda variante de **Closures** que tenemos en Ruby, la gran ventaja de **Proc** es que podemos mantener una referencia y pasarla como parámetro a otros métodos.

```
irb(main):001:0> def receive_proc(&block)
```

```
irb(main):002:1> puts "receive_proc"
```

```
irb(main):003:1> block.call
```

```
irb(main):004:1> end
```

```
=> :receive_proc
```

```
irb(main):005:0>
```

```
irb(main):006:0*> def test_proc(&block)
```

```
irb(main):007:1> puts "test_proc"
```

```
irb(main):008:1> receive_proc(&block)
```

```
irb(main):009:1> end

=> :test_proc

irb(main):010:0>

irb(main):011:0* test_proc do

irb(main):012:1*   puts "Being called from different method"

irb(main):013:1> end

test_proc

receive_proc

Being called from different method

=> nil
```

Mejor aún, podemos declarar una función anónima y guardarla como un **Proc** para poder ser utilizada más adelante en nuestro programa, esta es una característica que permite a Ruby transformar código a datos.

```
irb(main):001:0> multiply_by_two = Proc.new {|num| num * 2 }

=> #<Proc:0x007f88ab5c24f8@(irb):1>

irb(main):002:0>
```

```

irb(main):003:0* def calculate_numbers(*numbers, &block)

irb(main):004:1>   new_numbers = []

irb(main):005:1>

irb(main):006:1*   numbers.each do |number|


irb(main):007:2*     new_numbers << block.call(number)

irb(main):008:2>   end

irb(main):009:1>

irb(main):010:1*   new_numbers

irb(main):011:1> end

=> :calculate_numbers

irb(main):012:0>

irb(main):013:0* calculate_numbers 1, 2, 3, 4, &multiply_by_two

=> [2, 4, 6, 8]

```

Es posible declarar un **Proc** con la sintaxis alterna disponible a partir de Ruby 2.0

```
irb(main):001:0> multiply_by_two = ->(num) { num * 2 }
```

```
=> #<Proc:0x007fdb50a68500@(irb):1 (lambda)>
```

Si ponemos un poco de atención al mensaje que recibimos después de declarar nuestra función anónima, vemos que tiene una anotación que dice `lambda`. Esto nos lleva al tercer concepto a cubrir en esta sección.

Una función anónima de tipo `lambda` es ejecutada con la llamada al método `call` tal y como lo hacemos con un `Proc`.

```
irb(main):001:0> multiply_by_two.call 3
```

```
=> 6
```

Otra forma declarar un lambda es a través de la palabra clave `lambda`.

```
irb(main):001:0> multiply_by_two = lambda{|num| num * 2}
```

```
=> #<Proc:0x007fb55b0d4a68@(irb):1 (lambda)>
```

En realidad `Proc` y `lambda` son extremadamente parecidas, utilizamos ambas para declarar funciones anónimas dentro del Ruby, que como ya vimos nos da la ventaja de poder pasar código como si fuera datos en la forma de argumentos a otras funciones. Aunque las dos son muy parecidas existen algunas diferencias que nos lleva a usar una u otra.

Veamos el siguiente ejemplo:

```
irb(main):001:0> display_number_p = Proc.new{|num| puts num}
```

```
=> #<Proc:0x007fb55bd351e0@(irb):1>
```

```
irb(main):002:0> display_number_p.call
```

```

=> nil

irb(main):003:0> display_number_l = lambda{|num| puts num}

=> #<Proc:0x007fb55bd169e8@(irb):3 (lambda)>

irb(main):004:0> display_number_l.call

ArgumentError: wrong number of arguments (0 for 1)

from (irb):3:in `block in irb_binding'

from (irb):4:in `call'

from (irb):4

from /.rbenv/versions/2.1.1/bin/irb:4:in `<main>'
```

Primeramente declaramos en la variable `display_number_p` un `Proc` que acepta un parámetro, pero al momento de ejecutar el `Proc` no proveemos del parámetro y la función es ejecutada sin problemas.

Después declaramos en la variable `display_number_l` un `lambda`, que al igual que el `Proc` permite recibir un parámetro, pero al momento de ejecutar el `lambda` sin proveer del parámetro tenemos el error `ArgumentError: wrong number of arguments (0 for 1)` es decir el `lambda` se queja por la falta del parámetro.

Otra de las diferencia entre un `Proc` y un `lambda` es la forma en como tratan el control de flujo, por ejemplo usemos la palabra clave `return`.

```
irb(main):001:0> def display_messages(&block)
```

```
irb(main):002:1> puts "Starting display_messages"

irb(main):003:1> block.call

irb(main):004:1> puts "display_messages ended"

irb(main):005:1> end

=> :display_messages

irb(main):006:0>

irb(main):007:0*> message_proc = Proc.new do

irb(main):008:1*> puts "Running inside the Proc"

irb(main):009:1> return

irb(main):010:1> end

=> #<Proc:0x007f827a8ab070@(irb):7>

irb(main):011:0>

irb(main):012:0*> message_lambda = lambda do

irb(main):013:1*> puts "Running inside the lambda"
```

```
irb(main):014:1>    return

irb(main):015:1> end

=> #<Proc:0x007f827acb0a58@(irb):12 (lambda)>

irb(main):016:0>

irb(main):017:0* display_messages &message_proc

Starting display_messages

Running inside the Proc

LocalJumpError: unexpected return

from (irb):9:in `block in irb_binding'

from (irb):3:in `call'

from (irb):3:in `display_messages'

from (irb):17

from /.rbenv/versions/2.1.1/bin/irb:11:in `<main>'

irb(main):018:0>
```

```
irb(main):019:0* display_messages &message_lambda
```

```
Starting display_messages
```

```
Running inside the lambda
```

```
display_messages ended
```

```
=> nil
```

Si observamos la definición del `display_messages` veremos que despliega un mensaje, después ejecuta el bloque anónimo que se le pase, ya sea un `Proc` o una `lambda` y finalmente despliega un mensaje.

Cuando ejecutamos `display_messages` y le pasamos un `Proc`, en Ruby 2.0 tenemos un error `LocalJumpError: unexpected return`, el cual nos indica que no podemos usar `return` dentro de un `Proc`. En versiones anteriores de Ruby no recibimos el mismo error, el efecto sería que el `return` afecta al método `display_messages` terminándolo, es decir nunca veríamos el mensaje `display_messages ended`.

En el caso de ejecutar `display_messages` con un `lambda` como parámetro, vemos que `return` no genera un error y que `return` solo afecta al código contenido dentro del `lambda`, es decir regresa el control de ejecución a `display_messages`.

## Clases

Desde el inicio de este capítulo se mencionó que Ruby es un lenguaje orientado a objetos y hasta este momento lo hemos comprobado con algunas de las secciones previas en las que trabajamos con objetos que podemos instanciar a partir de las clases disponibles en la librería estándar de Ruby.

En esta sección veremos como es que podemos crear nuestras propias clases y conoceremos lo que Ruby nos ofrece como lenguaje en este ámbito.

La palabra reservada `class` es la que nos permite definir una nueva clase. La convención de nombre para una clase en Ruby es que el nombre es capitalizado, es decir la primera letra es

mayúscula y el resto se escribe en minúscula, cuando el nombre de la clase es una palabra compuesta se aplica la misma regla.

```
irb(main):001:0> class Person

irb(main):002:1> end

=> nil

irb(main):003:0> person = Person.new

=> #<Person:0x007f892c83e7a0>

irb(main):004:0> person.class

=> Person
```

Una vez que tenemos nuestra clase definida es posible crear una instancia de la misma, tal y como se muestra en la línea 3. La llamada al método `new` es el que nos devuelve una instancia de la clase.

Vamos a otorgarle comportamiento a nuestra clase para que sea un poco más interesante.

```
irb(main):005:0> class Person

irb(main):006:1> def say_hello

irb(main):007:2> puts 'Hello'

irb(main):008:2> end
```

```
irb(main):009:1> end  
  
=> :say_hello  
  
irb(main):010:0> person.say_hello  
  
Hello  
  
=> nil
```

Aquí hay algo muy importante a notar, definimos el método `say_hello` en nuestra clase `Person`, pero sucedió de una forma muy específica de Ruby, reabrimos nuestra clase para cambiar la forma en como funciona, y podemos apreciar este concepto en la línea 10 donde no fue necesario crear una nueva instancia para que la variable `person` adquiriera la nueva funcionalidad. Esta característica es propia de los lenguajes dinámicos como Ruby, en donde podemos cambiar código en tiempo de ejecución.

El método `say_hello` fue agregado en nuestra clase como un método de instancia, es decir que solamente se puede utilizar el método sobre un objeto que sea instancia de nuestra clase.

Vamos a reabrir nuestra clase para darle un nuevo comportamiento.

```
irb(main):011:0> class Person  
  
irb(main):012:1> def full_name  
  
irb(main):013:2> puts 'John Doe'  
  
irb(main):014:2> end  
  
irb(main):015:1> end
```

```
=> :full_name
```

```
irb(main):016:0> person.full_name
```

```
John Doe
```

```
=> nil
```

Ahora agregamos el método `full_name` el cual imprime un nombre, pero no es muy útil ya que no importa cuantas instancias tengamos de `Person` siempre va a imprimir el mismo nombre. Para arreglar este problema necesitamos que nuestra clase pueda guardar datos. En este caso necesitamos de variables de instancia que nos van a permitir guardar información que va a ser visible únicamente dentro del contexto de cada instancia.

Para inicializar los datos vamos a necesitar de un constructor en nuestra clase que permita que le pasemos el nombre al momento de inicializar nuestra instancia.

```
irb(main):017:0> class Person
```

```
irb(main):018:1> def initialize(first_name, last_name)
```

```
irb(main):019:2> @first_name = first_name
```

```
irb(main):020:2> @last_name = last_name
```

```
irb(main):021:2> end
```

```
irb(main):022:1> end
```

```
=> :initialize
```

```
irb(main):023:0> person = Person.new 'Mario', 'Chavez'  
=> #<Person:0x007fb735370c28 @first_name="Mario", @last_name="Chavez">
```

Al inicializar una nueva instancia de `Person` las variables `@first_name` y `@last_name` se inicializan con los valores que pasamos en el constructor. Ahora vamos a redefinir el método `full_name` que escribimos antes para que utilice los valores de nuestras variables de instancia.

```
irb(main):024:0> class Person  
  
irb(main):025:1> def full_name  
  
irb(main):026:2> "#{@first_name} #{@last_name}"  
  
irb(main):027:2> end  
  
irb(main):028:1> end  
  
=> :full_name  
  
irb(main):029:0> person.full_name  
  
=> "Mario Chavez"
```

Ya vimos que nuestro objeto puede mantener estado, el cual hasta este momento no es solo accesible desde el método `full_name` pero que pasa si queremos acceder a los nombres por separado desde fuera de nuestro objeto? Para tal efecto necesitas crear un par de accesores o propiedades.

```
irb(main):029:0> class Person
```

```
irb(main):030:1> def first_name  
  
irb(main):031:2> @first_name  
  
irb(main):032:2> end  
  
irb(main):033:1> def last_name  
  
irb(main):034:2> @last_name  
  
irb(main):035:2> end  
  
irb(main):036:1> end  
  
=> :last_name  
  
irb(main):037:0> person.first_name  
  
=> "Mario"  
  
irb(main):038:0> person.last_name  
  
=> "Chavez"
```

Ambos accesores son de modo sólo lectura, en Ruby hay una forma más simple de escribirlos.

```
irb(main):039:0> class Person  
  
irb(main):040:1> attr_reader :first_name, :last_name
```

```
irb(main):041:1> end  
  
=> nil  
  
irb(main):042:0> person.last_name  
  
=> "Chavez"
```

Con la palabra clave `attr_reader` seguida de las símbolos de los nombres de los accesores se inyectan en nuestra clase la definición de los métodos para leer el valor de las variables.

Hasta este momento nuestra clase funciona prácticamente en modo de sólo lectura. El incluir funcionalidad para poder acceder y modificar datos internos en la clase es muy sencillo.

```
irb(main):043:0> class Person  
  
irb(main):044:1> def address=(value)  
  
irb(main):045:2> @address = value  
  
irb(main):046:2> end  
  
irb(main):047:1> def address  
  
irb(main):048:2> @address  
  
irb(main):049:2> end  
  
irb(main):050:1> end  
  
=> :address
```

```
irb(main):051:0> person.address = 'Ciudad de Mexico'
```

```
=> "Ciudad de Mexico"
```

```
irb(main):052:0> person.address
```

```
=> "Ciudad de Mexico"
```

```
irb(main):053:0> person
```

```
=> #<Person:0x007f9f6b3e9510 @first_name="Mario", @last_name="Chavez", @address="Ciudad de Mexico">
```

Obviamente tal como en el accesor de sólo lectura hay una forma simple en Ruby de implementar un accesor de lectura/escritura con `attr_accessor`, al igual que en el caso anterior simplemente listamos con símbolos los nombres de los accesores que deseamos agregar a nuestra clase.

```
irb(main):054:0> class Person
```

```
irb(main):055:1> attr_accessor :address
```

```
irb(main):056:1> end
```

```
=> nil
```

```
irb(main):057:0> person.address
```

```
=> "Ciudad de Mexico"
```

Hasta este momento todos los métodos que hemos escrito en nuestra clase son métodos de instancia, es decir solo están disponibles cuando creamos un objeto a partir de nuestra clase, pero

hay ocasiones en donde necesitamos métodos que estén disponibles desde la definición de la clase, es decir que sean métodos de clase.

```
irb(main):058:0> class Person

irb(main):059:1>   def self.say_hello

irb(main):060:2>     puts "Saying hello from no one"

irb(main):061:2>   end

irb(main):062:1> end

=> :say_hello

irb(main):063:0> Person.say_hello

Saying hello from no one

=> nil
```

Como podemos apreciar en el ejemplo, no fue necesario crear una instancia de `Person` para llamar al método `say_hello`. En este ejemplo introducimos una palabra clave `self`. `self` es una palabra clave un tanto compleja, por el momento nos será suficiente con saber que `self` es la referencia a la clase `Person`.

Otra forma de poder escribir este mismo código en Ruby es de la siguiente forma:

```
irb(main):064:0> class Person

irb(main):065:1>
```

```
irb(main):066:1*   class << self  
  
irb(main):067:2>     def say_hello  
  
irb(main):068:3>       puts "Saying hello from no one"  
  
irb(main):069:3>     end  
  
irb(main):070:2>   end  
  
irb(main):071:1>  
  
irb(main):072:1* end  
  
=> :say_hello  
  
irb(main):073:0> Person.say_hello  
  
Saying hello from no one  
  
=> nil
```

Esta notación tiene el mismo efecto que en la notación previa.

Ahora veamos el concepto de herencia en Ruby. Al ser Ruby un lenguaje orientado a objetos, es posible definir una clase padre y a partir de ésta definir clases hijas o heredadas. La única limitante que tiene Ruby con respecto a otros lenguajes de programación es que una clase hija solo puede tener herencia de una clase padre, pero veremos más adelante que esto no representa ningún problema.

```
irb(main):001:0> class Person
```

```
irb(main):002:1> attr_reader :first_name, :last_name

irb(main):003:1> attr_accessor :address

irb(main):004:1>

irb(main):005:1*> def initialize(first_name, last_name)

irb(main):006:2>   @first_name = first_name

irb(main):007:2>   @last_name = last_name

irb(main):008:2> end

irb(main):009:1>

irb(main):010:1*> def full_name

irb(main):011:2>   "#{first_name} #{last_name}"

irb(main):012:2> end

irb(main):013:1> end

=> :full_name

irb(main):014:0> class Employee < Person
```

```

irb(main):015:1> end

=> nil

irb(main):016:0> employee = Employee.new 'John', 'Doe'

=> #<Employee:0x007fa7a5cda678 @first_name="John", @last_name="Doe">

irb(main):017:0> employee.full_name

=> "John Doe"

```

En éste ejemplo vemos que usamos < para denotar que `Employee` hereda de `Person`. Podemos hacer modificaciones a las clase `Employee` pero esas modificaciones solo afectarán a las instancias que derivan de `Employee`.

```

irb(main):018:0> person = Person.new 'Jane', 'Doe'

=> #<Person:0x007fa7a5cbdff0 @first_name="Jane", @last_name="Doe">

irb(main):019:0> class Employee < Person

irb(main):020:1>   attr_accessor :area

irb(main):021:1> end

=> nil

irb(main):022:0> person.area = :finance

```

```
NoMethodError: undefined method `area=' for #<Person:0x007fa7a5cbdff0 @first_-
name="Jane", @last_name="Doe">
```

```
from (irb):22
```

```
from /.rbenv/versions/2.1.1/bin/irb:11:in `<main>'
```

```
irb(main):023:0> employee.area = :payroll
```

```
=> :payroll
```

Cambios en `Person` afectaran a `Employee` pero no al revés. Es posible en una clase hija el sobre-escribir un método de la clase padre.

```
irb(main):024:0> class Employee < Person
```

```
irb(main):025:1>   def full_name
```

```
irb(main):026:2>     "#{last_name}, #{first_name}"
```

```
irb(main):027:2>   end
```

```
irb(main):028:1> end
```

```
=> :full_name
```

```
irb(main):029:0> person.full_name
```

```
=> "Jane Doe"
```

```
irb(main):030:0> employee.full_name
```

```
=> "Doe, John"
```

En este caso reemplazamos totalmente la funcionalidad del método de la clase padre, pero es posible ejecutar el método original proveído por la clase padre y tener lógica adicional en el método de la clase hija.

```
irb(main):031:0> class Employee < Person
```

```
irb(main):032:1>   def full_name
```

```
irb(main):033:2>     super.sub ' ', ', '
```

```
irb(main):034:2>   end
```

```
irb(main):035:1> end
```

```
=> :full_name
```

```
irb(main):036:0>
```

```
irb(main):037:0*>
```

```
irb(main):038:0*> person.full_name
```

```
=> "Jane Doe"
```

```
irb(main):039:0> employee.full_name
```

```
=> "John, Doe"
```

Mediante el uso de la palabra clave **super** desde el método de la clase hija delegamos al método original de la clase padre y regresamos en control.

En muchos lenguajes hoy en día es posible crear **namespaces** para crear separaciones lógicas o agrupar de clases en base un contexto específico. Con Ruby podemos hacer uso de la palabra clave **module** para crear estos **namespaces**.

```
irb(main):001:0> module Geometry

irb(main):002:1>   class Point

irb(main):003:2>     attr_accessor :x, :y

irb(main):004:2>

irb(main):005:2*>     def initialize(x, y)

irb(main):006:3>       @x = x

irb(main):007:3>       @y = y

irb(main):008:3>     end

irb(main):009:2>   end

irb(main):010:1> end

=> :initialize
```

```
irb(main):011:0>

irb(main):012:0* point1 = Geometry::Point.new 1, 2

=> #<Geometry::Point:0x007f93fabba360 @x=1, @y=2>
```

Como vemos en el ejemplo una vez que ponemos a una clase dentro de un **namespace** para hacer referencia a ésta es necesario usara el **nombre completo** de la clase para poder hacer uso de la misma.

Pero los módulo en Ruby tienen una funcionalidad adicional donde es posible **"mezclarlos"** en una clase. El poder realizar un **mixin** en Ruby nos permite extender la funcionalidad de una clase combinando funcionalidad a diferencia de hacerlo por herencia y la limitación que ya se comentó en Ruby.

```
irb(main):012:0* point1 = Geometry::Point.new 1, 2

irb(main):013:0> point1.to_s

=> "#<Geometry::Point:0x007fc7a1c51118>"

irb(main):014:0> module CustomToS

irb(main):015:1>   def to_s

irb(main):016:2>     "Geometry::Point #: ({x},#{y})"

irb(main):017:2>   end

irb(main):018:1> end
```

```
=> :to_s

irb(main):019:0> module Geometry

irb(main):020:1>   class Point

irb(main):021:2>     include CustomToS

irb(main):022:2>   end

irb(main):023:1> end

=> Geometry::Point

irb(main):024:0> point1.to_s

=> "Geometry::Point #: (1,2)"
```

En este ejemplo tenemos la clase `Geometry::Point` de la cual creamos una instancia y llamamos el método `to_s`, el cual es el método estándar que todos los objetos de Ruby adquieren. Podemos escribir un modulo, en este caso `CustomToS` para reemplazar el método `to_s` y que nos muestre otro formato como resultado. La inclusión del módulo `CustomToS` fue simple con la palabra clave `include` en la definición de nuestra clase.

En este caso en particular el módulo `CustomToS` está muy ligado a nuestra clase `Geometry::Point` ya que tiene conocimiento de cuales son las propiedades que deseamos imprimir en la llamada a `to_s` así como tener explícitamente la definición de la clase `Geometry::Point`, esto es fácil de solucionar haciendo que el modulo `CustomToS` no esté tan ligado a ésta clase y se pueda usar de forma genérica en otras clases.

```
irb(main):025:0> module CustomToS
```

```
irb(main):026:1>   def to_s  
  
irb(main):027:2>     "#{self.class.name} #: #{data_to_s}"  
  
irb(main):028:2>   end  
  
irb(main):029:1> end  
  
=> :to_s  
  
irb(main):030:0> module Geometry  
  
irb(main):031:1>   class Point  
  
irb(main):032:2>     private  
  
irb(main):033:2>     def data_to_s  
  
irb(main):034:3>       "(#{x}, #{y})"  
  
irb(main):035:3>   end  
  
irb(main):036:2> end  
  
irb(main):037:1> end  
  
=> :data_to_s
```

```
irb(main):038:0> point1.to_s
```

```
=> "Geometry::Point #: (1, 2)"
```

Tenemos el mismo efecto con este cambio, pero nuestro módulo `CustomToS` se vuelve más genérico y puede reutilizarse en otras clases con funcionalidad distinta, primeramente porque en lugar de tener explícitamente el nombre de la clase para que estamos mostrando la información ahora pregunta a través de la palabra clave `self` el nombre de la misma. Segundo, el módulo ahora requiere de un contrato con la clase que use, este contrato espera que la clase donde se incluya el módulo responda al método `data_to_s`.

Así como pudimos incluir un módulo en una clase para proveer de mayor funcionalidad a nuestros objetos instancias a partir de dicha clase, podemos extender una clase con métodos de clase o estáticos a partir de un módulo.

```
irb(main):039:0> module PointInitializer
```

```
irb(main):040:1>   def new_centered(position)
```

```
irb(main):041:2>     self.new(position, position)
```

```
irb(main):042:2>   end
```

```
irb(main):043:1> end
```

```
=> :new_centered
```

```
irb(main):044:0>
```

```
irb(main):045:0*> module Geometry
```

```
irb(main):046:1>   class Point
```

```
irb(main):047:2>     extend PointInitializer

irb(main):048:2>   end

irb(main):049:1> end

=> Geometry::Point

irb(main):050:0> point2 = Geometry::Point.new_centered(3)

=> #<Geometry::Point:0x007fc7a1a29d68 @x=3, @y=3>

irb(main):051:0> point2.to_s

=> "Geometry::Point #: (3, 3)"
```

Haciendo uso de la palabra clave `extend` ahora nuestro módulo agrega métodos a la clase, en este caso agregamos un nuevo inicializador para `Geometry::Point` llamado `new_centered`.

Los módulos en Ruby son una herramienta para reutilizar código a través de la composición en lugar de la herencia clásica en un lenguaje orientado a objetos.

## *Metaprogramación*

Al inicio del libro se hizo mención que una de las características más importantes de Ruby como lenguaje es su habilidad para poder realizar metaprogramación.

La metaprogramación por sí sola requiere de un libro completo, es por eso que en esta sección solamente vamos a ver algunos ejemplos básicos.

Anteriormente se había hecho mención de que en Ruby todo es un objeto, una forma sencilla de probarlo es revisando la herencia de nuestras clases.

```
irb(main):052:0> Geometry::Point.ancestors  
  
=> [Geometry::Point, CustomToS, Object, Kernel, BasicObject]
```

Vemos que para nuestra clase `Geometry::Point` en algún punto de la herencia tiene a `Object` como parent. Vamos a usar `Object` de forma directa para crear un objeto.

```
irb(main):053:0> an_object = Object.new  
  
=> #<Object:0x007fc7a1a0d280>  
  
irb(main):054:0>  
  
irb(main):055:0*> def an_object.set_variable=(var)  
  
irb(main):056:1>   @instance_variable = var  
  
irb(main):057:1> end  
  
=> :set_variable=  
  
irb(main):058:0>  
  
irb(main):059:0*> def an_object.get_variable  
  
irb(main):060:1>   @instance_variable  
  
irb(main):061:1> end
```

```

=> :get_variable

irb(main):062:0>

irb(main):063:0* an_object.set_variable = 'New Object

=> "New Object"

irb(main):064:0> an_object.get_variable

=> "New Object"

```

En el ejemplo vemos como podemos crear una instancia de `Object` y le definimos 2 métodos `set_variable` y `get_variable` los cuales solo están disponibles para nuestra instancia, si creamos otro objeto y tratamos de usar alguno de estos métodos vemos que obtenemos un error.

```

irb(main):065:0> other_object = Object.new

=> #<Object:0x007fc7a1b4c5b0>

irb(main):066:0> other_object.set_variable = 'Other Object'

NoMethodError: undefined method `set_variable=' for #<Object:0x007fc7a1b4c5b0>

from (irb):66

from /.rbenv/versions/2.1.2/bin/irb:11:in `<main>'

```

Ya vimos que podemos crear objetos en tiempo de ejecución y agregarles funcionalidad al momento ahora vamos a crear una clase en tiempo de ejecución.

```
irb(main):068:0> Person = Class.new  
  
=> Person  
  
irb(main):069:0> def Person.who_ami  
  
irb(main):070:1>   self.name  
  
irb(main):071:1> end  
  
=> :who_ami  
  
irb(main):072:0> Person.who_ami  
  
=> "Person"
```

Vamos a trabajar un poco sobre nuestra nueva clase para darle un poco de funcionalidad.

```
irb(main):073:0*> Person.class_eval do  
  
irb(main):074:1*>   attr_accessor :name  
  
irb(main):075:1>  
  
irb(main):076:1*>   def reverse_name  
  
irb(main):077:2>     name.reverse  
  
irb(main):078:2>   end
```

```
irb(main):079:1> end

=> :reverse_name

irb(main):080:0>

irb(main):081:0*> person = Person.new

=> #<Person:0x007fc7a19549b0>

irb(main):082:0> person.name = 'John'

=> "John"

irb(main):083:0> person.reverse_name

=> "nhoJ"
```

En tiempo de ejecución hemos creado una clase y le hemos dado comportamiento a nuestra clase a través del métodos `class_eval`.

Con ambos ejemplo posiblemente ya nos dimos cuenta de que en Ruby es posible escribir programas que escriban programas en base a la información de su entorno, lo que convierte a Ruby en una herramienta muy poderosa pero también en una herramienta que hay que manejarla con cuidado.

Vamos volviendo a nuestra clase `Geometry::Point` y el módulo `CustomToS` para ver los siguientes ejemplos.

El módulo `CustomToS` contiene un contrato en donde la clase que incluya este módulo debe de tener el método `data_to_s`, ¿qué pasa si incluimos el módulo en una clase que no implemente el contrato?

```

irb(main):84:0> class Triangle

irb(main):85:1>   include CustomToS

irb(main):86:1> end

=> Triangle

irb(main):87:0>

irb(main):88:0*> t = Triangle.new

=> #<Triangle:0x007fc7a203a478>

irb(main):89:0> t.to_s

NameError: undefined local variable or method `data_to_s' for #<Triangle:
0x007fc7a203a478>

from (irb):27:in `to_s'

from (irb):89

from /.rbenv/versions/2.1.2/bin/irb:11:in `<main>'
```

Al momento de realizar la llamada a `to_s` tenemos un error de que el método o variable `data_to_s` no existe. Una forma de prevenir este error y avisar al desarrollador que tiene que seguir el contrato se muestra en el siguiente ejemplo.

```
irb(main):106:0> module CustomToS
```

```

irb(main):107:1>   def to_s

irb(main):108:2>     data = if self.respond_to?(:data_to_s, true)

irb(main):109:3>       data_to_s

irb(main):110:3>     else
irb(main):111:3*>       "Please implement method #data_to_s in your class #{self.-
class.name}"
irb(main):112:3>   end

irb(main):113:2>

irb(main):114:2*>   "#{self.class.name} #: #{data}"

irb(main):115:2>   end

irb(main):116:1> end

=> :to_s

irb(main):117:0> t.to_s

=> "Triangle #: Please implement method #data_to_s in your class Triangle"

```

En este caso primero preguntamos si la clase donde se incluyó nuestro módulo responde al método `data_to_s` antes de intentar hacer la llamada, esto lo logramos a través de `respond_to?`.

Ahora vamos cambiando un poco de contexto y enfocarnos a la clase `Geometry::Point`. Supongamos que queremos tener un método para incrementar el valor de `x` o de `y`, una solución para este problema es la siguiente.

```
irb(main):118:0* module Geometry

irb(main):119:1>   class Point

irb(main):120:2>     def increment(coordenate)
irb(main):121:3>       @x +=1 if coordenate == :x
irb(main):122:3>       @y +=1 if coordenate == :y
irb(main):123:3>     end
irb(main):124:2>   end

irb(main):125:1> end

=> :increment

irb(main):126:0>

irb(main):140:0* point = Geometry::Point.new(1, 2)

=> #<Geometry::Point:0x007fc7a1bd2908 @x=1, @y=2>

irb(main):127:0> point.to_s
```

```
=> "Geometry::Point #: (1, 2)"  
  
irb(main):128:0> point.increment(:x)  
  
=> nil  
  
irb(main):129:0> point.to_s  
  
=> "Geometry::Point #: (2, 2)"
```

Ahora que pasa si nos piden un método especializado para incrementar `x` o `y` estaríamos posiblemente pensando en agregar `increment_x` e `increment_y` pero vamos mejor tomando ventaja de la metaprogramación en Ruby.

En Ruby existe un método especial en nuestras clases llamada `method_missing`, el cual es ejecutado cuando en un objeto tratamos de hacer una llamada a un método inexistente. Podemos tomar ventaja de esta característica de Ruby para nuestra implementación.

```
irb(main):130:0> module Geometry  
  
irb(main):131:1>   class Point  
  
irb(main):132:2>     def method_missing(method, *args, &block)  
  
irb(main):133:3>       last_char = method[-1]  
  
irb(main):134:3>       if ['x', 'y'].include?(last_char)  
  
irb(main):135:4>         send(:increment, last_char.to_sym)  
  
irb(main):136:4>       else
```

```
irb(main):137:4*      super

irb(main):138:4>      end

irb(main):139:3>      end

irb(main):140:2>    end

irb(main):141:1> end

=> :method_missing

irb(main):142:0>

irb(main):143:0* point.to_s

=> "Geometry::Point #: (2, 2)"

irb(main):144:0> point.increment_x

=> nil

irb(main):145:0> point.to_s

=> "Geometry::Point #: (3, 2)"

irb(main):146:0> point.increment_y
```

```

=> 3

irb(main):147:0> point.to_s

=> "Geometry::Point #: (3, 3)"

irb(main):148:0> point.increment_z

NoMethodError: undefined method `increment_z' for #<Geometry::Point:
0x007f9869ad33c0 @x=3, @y=2>

from (irb):132:in `method_missing'

from (irb):148

from /.rbenv/versions/2.1.2/bin/irb:11:in `<main>'

```

Ahora nuestra clase `Geometry::Point` entiende que es lo que tiene que hacer cuando se hace una llamada al método `increment_` con el nombre de alguna de las coordenadas y vemos que falla cuando tratamos de ejecutarlo con una coordenada que no existe.

La pieza clave en nuestra implementación de `method_missing` es el uso del método `send` el cual permite ejecutar de forma dinámica un método.

```
irb(main):149:0> point.send('to_s')
```

```
=> "Geometry::Point #: (3, 3)"
```

```
irb(main):150:0> point.send(:to_s)
```

```
=> "Geometry::Point #: (3, 3)"
```

Como vemos en el ejemplo `send` puede recibir el nombre del método a ejecutar como una cadena de texto o como un símbolo.

Como vimos en los ejemplos, el dinamismo de Ruby es increíble y usándolo de forma responsable podemos hacer programas que se adapten y funciones en diferentes circunstancias.

## Conclusiones.

En este primer capítulo conocimos un poco de la historia que hay detrás de Ruby, su concepción filosófica y técnicamente algunas de las cualidades del lenguaje.

Es imposible enseñar Ruby en un sólo capítulo pero lo aquí mostrado nos servirá durante el resto del libro para entender los ejemplos y conocer un poco.

# Capítulo 2 - Ruby on Rails, el desarrollo ágil de aplicaciones web

Ruby on Rails es un marco de trabajo para el desarrollo de aplicaciones. Ruby on Rails es considerado una herramienta que permite desarrollar aplicaciones web modernas de forma muy rápida.

Parte de esto se debe a que Ruby on Rails provee de una serie de convenciones que permiten al desarrollador enfocarse en la parte importante de la aplicación y no en los detalles que no agregan valor a la aplicación web.

A diferencia de los marcos de desarrollo que surgieron a finales y principios de los 90's, Ruby on Rails es un marco de desarrollo que nació a partir de aplicaciones reales y como vamos a ver un poco más adelante, fue el parteaguas del paradigma de desarrollo web, inclusive podríamos hablar de una época pre y post Ruby on Rails.

En el capítulo anterior vimos como es que el lenguaje Ruby está diseñado para que los desarrolladores sean felices y Ruby on Rails no se aleja mucho de la misma ideología.

## Historia

Ruby on Rails cumple 10 años en este 2014, en términos de software podemos concluir que es una herramienta madura en su versión 4.0.

Fue en el 2003 cuando David Heinemeier Hansson inició con el desarrollo de Ruby on Rails. Como se mencionó anteriormente Ruby on Rails - o simplemente Rails como lo mencionaré de forma indistinta en el resto del libro - nació a partir de un producto real, de Basecamp<sup>1</sup> el producto estrella de la entonces 37 Signals<sup>2</sup>.

David inició el desarrollo de Basecamp en Ruby y encontró patrones que no eran específicos a Basecamp pero que podían reutilizarse para el desarrollo de aplicaciones web. Esto sucede en una época en donde empresas como Microsoft y las extintas Sun Microsystems y Bea Logic em-

---

<sup>1</sup> <https://basecamp.com>

<sup>2</sup> <http://37signals.com>

pujaban para ser los líderes del desarrollo web visual, tratando de emular la experiencia del desarrollo de aplicaciones para escritorio hacia el Web.

La primera versión pública de Rails<sup>3</sup> la anunció David el 25 de Julio del 2004, fue la version 0.5.0 y la llamó "The end of vaporware!". Con el anuncio David describió a Rails como:

Ruby on Rails es un marco de desarrollo web para Ruby. Rails incluye una solución para cada letra de **MVC**: ActionPack para el Controlador y las Vistas, ActiveRecord para el Modelo.

Al ser un marco de desarrollo **Full Stack**, Rails permite el concepto de **Don't Repeat Yourself** o **DRY**, así como favorece a la introspección y extensiones al momento de ejecución en lugar de archivos de configuración, esto posteriormente lo conocimos como **Convention over Configuration**.

Para el 15 de diciembre del 2005 Ruby on Rails vio el anuncio de la versión 1.0<sup>4</sup>.

Ruby on Rails fue la inspiración para desarrolladores en otros lenguajes como Java, Groovy, PHP, etc. para crear marcos de desarrollo parecidos o modelados bajo las ideas de Ruby on Rails.

## Beneficios

Uno de los beneficios principales de Ruby on Rails es el concepto de **Convención sobre configuración** el cual nos elimina temas como el decidir como estructurar nuestra aplicación de Ruby on Rails.

Esto hace que la anatomía de una aplicación de Rails sea entre un 80% y 90% parecida en todas las aplicaciones de Rails. La curva de aprendizaje para desarrolladores que se integran a una aplicación de Rails en la que ya hay un avance en su desarrollo es menor.

Un punto importante en Rails es que con la configuración por omisión es posible empezar el desarrollo y no perder tiempo en esos detalles, más adelante si es necesario se puede ir ajustando la configuración.

---

<sup>3</sup> <http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/107370>

<sup>4</sup> <http://weblog.rubyonrails.org/2005/12/13/rails-1-0-party-like-its-one-oh-oh/>

Ruby on Rails provee de una serie de generadores que nos permiten prototipar y realizar un desarrollo acelerado de conceptos e ideas. Un ejemplo de que tan rápido se puede desarrollar con Ruby on Rails es el concurso Rails Rumble<sup>5</sup> donde sólo se cuenta con 48 horas para desarrollar una aplicación.

Como parte del desarrollo de aplicaciones con Rails, también se ofrece la posibilidad de realizar pruebas automáticas para las diferentes áreas de nuestra aplicación. Las pruebas son un medio para dar certidumbre y asegurarse que aún con cambios futuros la aplicación tendrá un comportamiento mínimo esperado.

Se dice que Rails agrega mucha *Magia* al desarrollo de aplicaciones, pero no es *Magia*, simplemente Rails nos evita de tener que tomar toda una serie de decisiones en nuestra aplicación, por ejemplo ¿Cómo nos vamos a conectar a nuestra base de datos?, ¿Cómo organizamos el código en nuestro proyecto?, etc. El concepto de *Magia* desaparece una vez que entendemos los patrones y convenciones de una aplicación de Rails.

## Rails y patrones de diseño

Como se mencionó anteriormente Ruby on Rails adoptó algunos patrones de diseño que ayudan a que todas las aplicaciones de Rails tengan la misma anatomía. Los siguientes conceptos forman parte intrínseca de la raíz de Rails y sirvieron para ayudar a la popularidad del marco de trabajo.

### MVC

Ruby on Rails popularizó el patrón de MVC - Model View Controller - formulado por Trygve Reenskaug<sup>6</sup> en la década de los 70's. El patrón describe la forma en como se debería de realizar la interacción de un usuario de un sistema y un sistema.

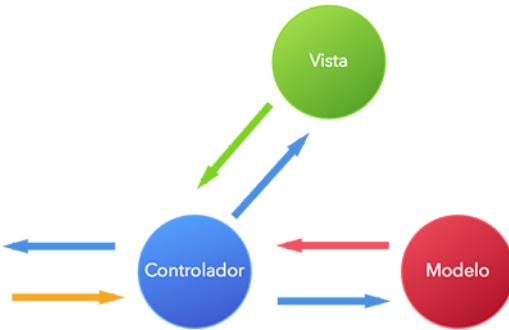
Desde el punto de vista del sistema, éste se divide en 3 componentes principales: el Modelo, la Vistas y el Controlador. Cada componente tiene responsabilidades únicas. El trabajo de los 3 componentes en conjunto permiten responder a usuario del sistema.

---

<sup>5</sup> <http://railsrumble.com>

<sup>6</sup> [http://en.wikipedia.org/wiki/Trygve\\_Reenskaug](http://en.wikipedia.org/wiki/Trygve_Reenskaug)

# MVC



1. **Modelo:** Representa datos en forma de un sólo objeto o un grupo de objetos, en el caso de Rails, un modelo es un objeto o una colección de objetos de **ActiveRecord**, que como vamos a ver más adelante es otro patrón de diseño.
  2. **Vista:** Es la representación visual del modelo, es decir es la interfase gráfica que le permite al usuario visualizar la información del modelo. En el caso de Rails una vista es una plantilla de **HTML** y código de Ruby empotrado que al ser procesada genera el código de **HTML** final que es enviado al navegador.
  3. **Controlador:** Tiene la responsabilidad de recibir una solicitud del usuario, determinar si se requiere de datos del modelo y pasar los datos al contexto de la vista para que se genere el **HTML** que será enviado al navegador.

## *ActiveRecord*

ActiveRecord es un patrón que ayuda a la interacción de una aplicación con una base de datos. De acuerdo a Martin Fowler<sup>7</sup>, **ActiveRecord** es un objeto que representa un registro en una tabla o vista, encapsula el acceso a la base de datos y puede contener lógica.

En Rails un Modelo es un objeto de **ActiveRecord** con el comportamiento anteriormente mencionado.

<sup>7</sup> <http://www.martinfowler.com/eaaCatalog/activeRecord.html>

## *REST*

**REST** significa *Transferencia de Estado Representacional*, en términos simples **REST** es un estilo de arquitectura para sistemas distribuidos como el **WWW**. **REST** provee entre otras cosas de consistencia en el estilo de la arquitectura a través de algunas restricciones para dicha arquitectura.

El concepto de **REST** fue introducido por Roy Thomas Fielding<sup>8</sup> en el 2000 como parte de su tesis doctoral.

Cada restricción definida en **REST** trae consigo beneficios desde portabilidad, visibilidad, estabilidad y escalabilidad.

1. Cliente/Servidor: A través de esta restricción se busca el separar las responsabilidades de la interfase gráfica con el proceso y almacenamiento de información. Lo que permite que ambas partes puedan evolucionar de forma independiente.
2. Sin estado: La interacción del cliente/servidor debe de no tener estado, es decir cada solicitud al servidor debe contener toda la información necesaria para que el servidor desde su contexto entienda de que se trata la solicitud.
3. Cacheable: Cada respuesta del servidor debe de ser marcada como *cacheable* o no y el cliente debe de decidir si reutiliza esa respuesta o necesita una nueva.
4. Interfase uniforme: Es una forma de simplificar la interfase a través de la que se conectan el cliente/servidor y a la vez desacoplar la implementación de los servicios o procesos que intervienen.
5. Un sistema capas: Permite la organización en capas de componentes que no *ven* más allá de su capa inmediata y que permite manejar la complejidad de un sistema agregando o eliminando componentes.
6. Código bajo demanda: Es posible que el servidor envíe código al cliente para modificar o extender su comportamiento.

Hay mucho más de que hablar sobre **REST** pero por el momento con lo que se menciona en esta sección es suficiente; más adelante en el transcurso del libro volveremos a tocar el tema y utilizaremos elementos arquitecturales de **REST** para ver de que forma se integran en la ideología de Rails.

---

<sup>8</sup> <http://www.ics.uci.edu/>

## Rails en el mundo real

Durante una época, Ruby on Rails fue muy popular entre *Startups*, muchas de las cuales el día de hoy son empresas perfectamente establecidas y reconocidas.

El ejemplo más representativo, sobre todo porque fue a partir de esta aplicación de como nació Ruby on Rails, es Basecamp<sup>9</sup>. Aplicación para organizar proyectos y equipos de trabajo.

Otro ejemplo representativo es Github(<http://github.com>), hoy en día una de las *piedras angulares* del desarrollo de *Open Source*. Github se describe a así mismo como la red social de los desarrolladores de software.

Siguiendo con las aplicaciones y servicios para desarrolladores, otro ejemplo de empresas que usan Ruby on Rails es Heroku<sup>10</sup>. Un servicio de hosting en la nube totalmente automático. Nació precisamente para aplicaciones de Ruby y posteriormente agrego soporte a otros lenguajes/frameworks.

Shopify<sup>11</sup> es un servicio que provee de tiendas de comercio electrónico a través de las cuales es posible tener una tienda lista en cuestión de minutos. De acuerdo a Shopify su servicio es utilizado por más de 150,000 tiendas.

En la línea de comercio electrónico se encuentra también SpreeCommerce<sup>12</sup>, la cual es una solución completa de comercio electrónico Open Source utilizada en miles de tiendas en todo el mundo.

500px<sup>13</sup> es una aplicación para fotógrafos donde pueden presentar su trabajo, recibir retroalimentación, publicar portafolios y a través del servicio *Prime* vender fotografías.

Otros ejemplos más son:

- [Indiegogo<sup>14</sup>](#)
- [Goodreads<sup>15</sup>](#)
- [Hulu<sup>16</sup>](#)

---

<sup>9</sup> <http://basecamp.com>

<sup>10</sup> <http://Heroku.com>

<sup>11</sup> <http://shopify.com>

<sup>12</sup> <https://spreecommerce.com/>

<sup>13</sup> <https://spreecommerce.com/>

<sup>14</sup> <https://www.indiegogo.com/>

<sup>15</sup> <http://www.goodreads.com/>

- [ScribD<sup>17</sup>](https://es.scribd.com)
- [CrunchBase<sup>18</sup>](https://www.crunchbase.com)
- [Square<sup>19</sup>](https://squareup.com)
- [AirBnB<sup>20</sup>](https://www.airbnb.com)
- [ThemeForest<sup>21</sup>](http://themeforest.net)

La lista de empresas que hoy en día utilizan Ruby on Rails puede ser interminable, los ejemplos antes mencionados son sólo una muestra de quienes utilizan esta marco de trabajo desde hace ya varios años en negocios exitosos.

## Nuestra primera aplicación de Ruby on Rails.

El primer paso para poder iniciar a desarrollar aplicaciones en Ruby on Rails es tener instalado Ruby en nuestro equipo de cómputo. Ya vimos en el capítulo anterior cómo realizar una instalación de Ruby.

Con Ruby instalado y listo para usar, es necesario instalar Ruby on Rails. Es muy sencillo realizar la instalación de Rails, esto gracias a la herramienta **gem** que existe en toda instalación de Ruby.

**gem** es una herramienta que hace la instalación de librerías de Ruby sea sumamente sencilla. **gem** utiliza directorios donde los desarrolladores registran sus librerías. El directorio más popular es Rubygems<sup>22</sup>.

Para instalar Ruby on Rails es tan sencillo como ejecutar el siguiente comando:

```
$ gem install rails
```

El comando instalará la versión más reciente de Rails, que al momento de escribir éste capítulo es la versión 5.0.0.1.

---

<sup>17</sup> <https://es.scribd.com>

<sup>18</sup> <https://www.crunchbase.com>

<sup>19</sup> <https://squareup.com>

<sup>20</sup> <https://www.airbnb.com>

<sup>21</sup> <http://themeforest.net>

<sup>22</sup> <http://rubygems.org>

Una vez que Rails está instalado en nuestro ambiente de desarrollo ha llegado el momento de crear nuestra primera aplicación de Rails.

Nuestra aplicación en éste capítulo va a ser muy sencilla. Vamos a crear una aplicación que permita a un equipo de desarrolladores registrar las respuestas de las 3 preguntas del *Standup meeting* de la metodología de *SCRUM*:

- ¿Qué hice hoy?
- ¿Qué voy a hacer mañana?
- ¿Hay algo que me impida cumplir mis objetivos?

En el caso de nuestra aplicación vamos a registrar el nombre del ingeniero que está realizando la entrada y en un área de texto la respuesta a las 3 preguntas.

Llamaremos a nuestra aplicación *Standup Meeting Log*.

Rails incluye un generador que permite crear toda la estructura de una aplicación de Rails de forma muy sencilla.

```
$ rails new standup_meeting_log
```

`standup_meeting_log` es un parámetro requerido al comando y representa al nombre de la aplicación de Rails. Es estándar que al crear una aplicación el nombre de la misma sea en minúsculas y si el nombre de conforma de más de una palabra, éstas se separan con un `_`.

Al finalizar la ejecución del comando tendremos un folder con el mismo nombre de nuestra aplicación y cuyo contenido es una aplicación completa de Rails, vacía, pero completa.

## *Anatomía de una aplicación de Rails*

Una aplicación de Ruby on Rails esta compuesta de varios folders y archivos que son generados al momento de crear una nueva aplicación.

```
$ ls
```

```
Gemfile      Procfile      Rakefile      bin      config.ru      lib
public       tmp          Gemfile.lock  README.md  app        config      db
log          test         vendor
```

El folder más importante es **app**. Éste folder contiene la estructura del código que es específico a nuestra aplicación.

Ahí podemos encontrar los Modelos, Vistas y Controladores además de recursos adicionales como Javascripts, Hojas de estilo e imágenes.

En una aplicación moderna de Rails en este folder colocaremos código que se encuentra fuera del patrón MVC pero que es necesario para nuestra aplicación.

**lib** es utilizado para código que no es exclusivo a nuestra aplicación y que en algún momento puede ser extraído en forma de una librería o gema como las conocemos en Ruby.

En **config** encontramos los archivos de configuración de Ruby on Rails así como archivos que se ejecutan al momento que nuestra aplicación inicia. Estos archivos de configuración contienen todo lo necesario para que nuestra aplicación desde el momento cero esté lista para iniciar y responder peticiones de un navegador. La configuración inicial nos permite enfocarnos en la funcionalidad de nuestra aplicación y no en configuraciones iniciales.

Todas nuestras pruebas automáticas se van a encontrar en el folder de **test**.

El archivo **Gemfile** es el manifiesto de las gemas de las cuales depende nuestra aplicación de Rails. Ahí podemos especificar nombre de las gemas y versiones específicas entre otras cosas.

Finalmente el archivo **config.ru** es el archivo que permite a nuestra aplicación de Rails iniciar encima de **Rack**. Rack es un API unificada, mínima y simple que permite la creación de aplicaciones de Ruby que requieren de gestión de HTTP.

Del resto de los folders y del contenido nos ocuparemos conforme avancemos en el contenido del libro.

## *Ejecutar una aplicación de Rails*

El que al momento de ejecutar el generador de aplicaciones de Rails se crea una aplicación vacía, es decir no tiene ninguna lógica ni código específico de nuestras necesidades, pero si se crea la configuración inicial completa de una aplicación de Rails.

Esto nos permite ejecutar nuestra aplicación un instante después de su creación, no tenemos que realizar ni especificar ninguna configuración adicional.

Estando dentro del folder de nuestra aplicación vamos a ejecutar el servidor web de Rails con el siguiente comando:

```
$ rails s
```

```
=> Booting Puma
```

```
=> Rails 5.0.0.1 application starting in development on http://localhost:3000
```

```
=> Run `rails server -h` for more startup options
```

```
Puma starting in single mode...
```

```
* Version 3.6.0 (ruby 2.3.1-p112), codename: Sleepy Sunday Serenity
```

```
* Min threads: 5, max threads: 5
```

```
* Environment: development
```

```
* Listening on tcp://localhost:3000
```

```
Use Ctrl-C to stop
```

Del resultado de ejecutar el comando `rails s` podemos ver que nuestra aplicación ha sido lanzada en el servidor web **Puma** y que está disponible en el puerto 3000 de **localhost**. Al abrir nuestro navegador en **http://localhost:3000** debemos de ver la siguiente pantalla.



Para detener al servidor web solamente hay que presionar **Ctrl-C**.

## *Los ambientes de ejecución de Rails*

Cuando ejecutamos nuestra aplicación vacía en la sección anterior, la segunda línea de la respuesta de ejecutar `rails s` nos indica que nuestra aplicación esta iniciando en modo de **development** o desarrollo.

En Rails tenemos por omisión 3 ambientes de ejecución: desarrollo, pruebas y producción. La gran diferencia entre los 3 ambientes es la configuración inicial de nuestra aplicación.

- Desarrollo: En este ambiente no existen optimizaciones que mejoren el desempeño de nuestra aplicación. Está configurado para que nos de el mayor detalle posible cuando existe un error y que la bitácora - o `log` - de Rails capture todo detalle. De igual forma cada vez que realizamos un cambio no es necesario detener y reiniciar el servidor para que los cambios sean visibles.
- Pruebas: Es un ambiente especial muy parecido a Desarrollo, el cual es utilizado para ejecutar pruebas de forma automática contra nuestra aplicación - es algo que vamos a ver en capítulos posteriores -. La característica principal es que la información en la base de datos del ambiente de pruebas generalmente se borra toda la información de la base de datos en cada ejecución.
- Producción: Este es el modo donde nuestra aplicación es optimizada para ejecutarse con el mejor desempeño posible para responder a solicitudes de usuarios reales.

Para nuestro caso en este momento con saber que estamos ejecutando nuestra aplicación en modo de desarrollo es suficiente.

### *Uso de scaffold*

Para iniciar con el desarrollo de ésta aplicación de demo utilizaremos uno de los generadores de Rails llamada **scaffold**, el cual nos da un punto de partida con los 3 elementos básicos de los patrones de Rails: modelo, controlador y vista.

El **scaffold** de igual forma generará otros artefactos como rutas y migraciones necesarias para una aplicación de Rails.

Para ejecutar el generador, desde el folder de nuestra aplicación ejecutamos:

```
$ bin/rails g scaffold entry name:string notes:text
```

La respuesta de este comando nos hace ver que creamos de forma automática una serie de archivos para nosotros. Pero vamos primero a analizar el comando que acabamos de ejecutar.

**bin/rails g** es la forma de indicarle a Rails que queremos hacer uso de uno de los generadores disponibles por omisión en Rails, si queremos ver la lista de los generadores solamente ejecutamos:

```
$ bin/rails g
```

**scaffold** es el nombre del generador que deseamos utilizar y el resto de los valores son sus parámetros. **entry** representa el nombre del recurso que deseamos generar, para nuestro ejemplo es una **entrada** a la bitácora de las **Standup Meetings**. **name** y **notes** son los atributos de cada entrada. **string** y **text** representan los tipos de datos que **name** y **notes** respectivamente van a guardar.

Vamos analizando más a detalle los archivos generados por el comando.

## Migraciones

Una migración en Rails la podemos ver como un mecanismo para controlar las versiones de cambios en el esquema de nuestra base de datos. Es decir cada cambio en la estructura de la

base de datos va a acompañado de un numero de versión que nos permite saber la secuencia de como se tienen que aplicar los cambios en orden.

Además Rails mantiene en la base de datos el numero de la última version aplicada de forma que es imposible aplicar el cambio en mas de una ocasión de forma accidental. Este mecanismo nos permite la recreación de bases de datos de una manera simple.

En nuestro caso el archivo `db/migrate/20150216172244_create_entries.rb` fue generado.

Simplemente con el nombre del archivo podemos empezar a ver ciertos puntos interesantes. El primero, es que el archivo tiene como parte del nombre un `timestamp` es decir tiene la fecha y hora de creación, esto es lo que nos ayuda versionar y a Rails a entender el orden en que una serie de migraciones deben de ser aplicadas.

El segundo punto es que como parte del nombre dice `entries` y no `entry` como especificamos con el comando `scaffold`. Veremos que Rails hace uso de la pluralización de palabras para definir los nombres de forma automática para ciertos elementos de la aplicación. A esto lo conocemos como **Convención sobre Configuración**. El cambio en el nombre indica que Rails va a crear una tabla `entries` para guardar la información.

Si abrimos el archivo nos encontramos con lo siguiente:

```
class CreateEntries < ActiveRecord::Migration

  def change

    create_table :entries do |t|

      t.string :name

      t.text :notes

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

```
end
```

```
end
```

```
end
```

Confirmamos que el nombre de la tabla es `entries` y que tiene 2 columnas, `name` y `notes`. También podemos apreciar que la sintaxis es una sintaxis especial de Rails y no son comandos de **SQL**. Es una sintaxis agnóstica al motor de la base de datos, esto permite técnicamente el poder reemplazar un motor de base de datos por otro bajo ciertas circunstancias.

Para aplicar la migración a la base de datos ejecutamos el siguiente comando:

```
$ rake db:migrate
```

El resultado de ésta operación nos indica que la tabla `entries` fue creada, pero demos un paso atrás, nunca especificamos el motor de base de datos, ¿Cómo es esto posible?.

Al momento de que creamos nuestra aplicación, no especificamos motor de base de datos, por este motivo Rails tomó la decisión de utilizar `sqlite`, un motor super ligero, autónomo y muy sencillo de utilizar. Esto lo podemos confirmar si abrimos el archivo `config/database.yml`

Por el momento no hay que preocuparnos por el motor de base de datos, cuando avancemos en los capítulos veremos como podemos cambiar el motor de base de datos.

Una vez ejecutada nuestra migración, se crea un nuevo archivo `db/schema.rb`. El cual se mantiene actualizado según vayamos creando más migraciones en nuestra aplicación.

Con respecto al comando `rake` vamos a utilizarlo muy seguido en el desarrollo de aplicaciones de Rails, por lo tanto nos estaremos familiarizando con él en el transcurso del libro.

## Modelos

Otro de los archivos creados por el generador de `scaffold` es el modelo `Entry`. Este es un modelo de `ActiveRecord` que nos permite realizar un mapeo entre una clase con atributos y una tabla con columnas. El archivo fue creado en `app/models/entry.rb`.

Al abrirlo nos encontramos con lo siguiente:

```
class Entry < ApplicationRecord
```

```
end
```

No tiene contenido. Simplemente define la clase y define que hereda de  `ApplicationRecord` . Pero antes de explicar que pasa, vamos a lanzar la consola de Rails para realizar algunos experimentos.

```
$ rails console
```

Una vez dentro de la consola vamos a ejecutar los siguientes comandos:

```
irb(main):001:0> Entry.all
```

```
Entry Load (2.5ms)  SELECT "entries".* FROM "entries"
```

```
=> #<ActiveRecord::Relation []>
```

```
irb(main):002:0> Entry.create name: 'Pedro Páramo', notes: 'Primera entrada'
```

```
(0.1ms)  begin transaction
```

```
SQL (0.4ms)  INSERT INTO "entries" ("name", "notes", "created_at",  
"updated_at") VALUES (?, ?, ?, ?)  [[{"name": "Pedro Páramo"}, {"notes": "Primera  
entrada"}, {"created_at": "2015-02-16 20:23:30.846879"}, {"updated_at":  
"2015-02-16 20:23:30.846879"}]
```

```
(1.3ms)  commit transaction
```

```
=> #<Entry id: 1, name: "Pedro Páramo", notes: "Primera entrada", created_at:  
"2015-02-16 20:23:30", updated_at: "2015-02-16 20:23:30">
```

```
irb(main):003:0> Entry.all

Entry Load (0.3ms)  SELECT "entries".* FROM "entries"

=> #<ActiveRecord::Relation [#<Entry id: 1, name: "Pedro Páramo", notes: "Primera
entrada", created_at: "2015-02-16 20:23:30", updated_at: "2015-02-16 20:23:30">>]
```

```
irb(main):004:0>
```

`Entry.all` realiza una consulta a la base de datos por todas las `entries`. Y regresa un `ActiveRecord::Relation` vacío.

`Entry.create name: 'Pedro Páramo', notes: 'Primera entrada'` crea una `entry` con los parámetros indicados.

Luego volvemos a ejecutar `Entry.all` y esta vez el objeto `ActiveRecord::Relation` contiene un elemento, el que creamos hace un momento.

Hagamos un par de pruebas más:

```
irb(main):004:0> entry = Entry.first

Entry Load (0.3ms)  SELECT  "entries".* FROM "entries"  ORDER BY "entries"."id"
ASC LIMIT 1
```

```
=> #<Entry id: 1, name: "Pedro Páramo", notes: "Primera entrada", created_at:
"2015-02-16 20:23:30", updated_at: "2015-02-16 20:23:30">>
```

```
irb(main):005:0> entry.name
```

```
=> "Pedro Páramo"
```

```
irb(main):006:0> entry.notes
```

```
=> "Primera entrada"
```

Ahora realizamos una consulta por el primer `entry` en la base de datos y lo asignamos a una variable. Despu s desplegamos el valor de los atributos `name` y `notes`.

Todo esto sin que la definici n de la clase `Entry` tenga m s c digo que simplemente indicar que hereda de `ApplicationRecord`.

Y es precisamente esta herencia la que nos permite realizar consultas a la base de datos, crear registros y consultar el valor de los atributos.

## Rutas y URLs

El generador `scaffold` tambi n modifico el archivo `config/routes.rb`, agregando al inicio del mismo la siguiente l nea:

```
resources :entries
```

El efecto que este cambio tiene lo podemos ver con uno de los comandos de `rake`.

```
$ rake routes
```

El cu l imprime una tabla como la siguiente:

| Prefix     | Verb | URI Pattern                 | Controller#Action |
|------------|------|-----------------------------|-------------------|
| entries    | GET  | /entries(.:format)          | entries#index     |
|            | POST | /entries(.:format)          | entries#create    |
| new_entry  | GET  | /entries/new(.:format)      | entries#new       |
| edit_entry | GET  | /entries/:id/edit(.:format) | entries#edit      |

```

entry GET      /entries/:id(.:format)          entries#show

PATCH /entries/:id(.:format)      entries#update

PUT   /entries/:id(.:format)      entries#update

DELETE /entries/:id(.:format)     entries#destroy

```

O si ejecutamos nuestro servidor de Rails y navegamos a <http://localhost:3000/rails/info/routes> podemos ver la misma información.

| Routes                                      |           |                             |                   |
|---|-----------|-----------------------------|-------------------|
| Routes match in priority from top to bottom |           |                             |                   |
| Helper                                      | HTTP Verb | Path                        | Controller#Action |
| Path / Uri                                  |           | Path Match                  |                   |
| entries_path                                | GET       | /entries(.:format)          | entries#index     |
|   | POST      | /entries(.:format)          | entries#create    |
| new_entry_path                              | GET       | /entries/new(.:format)      | entries#new       |
| edit_entry_path                             | GET       | /entries/:id/edit(.:format) | entries#edit      |
| entry_path                                  | GET       | /entries/:id(.:format)      | entries#show      |
|   | PATCH     | /entries/:id(.:format)      | entries#update    |
|   | PUT       | /entries/:id(.:format)      | entries#update    |
|   | DELETE    | /entries/:id(.:format)      | entries#destroy   |

Vamos a enfocarnos en las columnas **Verb**, **URI Pattern** y **Controller#Action**.

**URI Pattern** nos indica que **URL** ofrece nuestra aplicación de Rails para navegar la información, en el caso de `/entries` nos dice que podemos navegar a <http://localhost:3000/entries>. Sin embargo el patrón de `/entries` aparece duplicado, pero si vemos la columna de **Verb** observaremos que para uno el verbo es **GET** y para el otro es **POST**.

Para entender que pasa aquí, es necesario conocer los verbos que forman parte del estándar de Hypertext transfer protocol<sup>23</sup> los cuales dictan desde el navegador web la forma en como queremos acceder a un recurso, en este caso **entry**.

Para **GET** le indicamos al servidor web que queremos consultar información de ese recurso. En el caso de **POST** le indicamos al servidor web que deseamos crear un recurso de tipo **entry** y que le estamos enviando todos los parámetros necesarios.

---

<sup>23</sup> <http://es.wikipedia.org/wiki/HypertextTransferProtocol>

Adicionalmente si observamos la columna `Controller#Action` veremos que cada par de patrón/verbo están relacionados con una acción en un controlador. En nuestro caso la relación va con el controlador `Entries` y las acciones `index` y `create` respectivamente.

Es necesario detenernos un momento y darnos cuenta que los nombres tienen gran relevancia en Rails. Tenemos un modelo llamado `entry` que hace uso de una tabla `entries` y que las URLs para acceder a la información del mismo apuntan a `/entries` y un controlador `Entries` responsable para cada URL definida.

Nada de esto es una casualidad, es parte de como Rails está diseñado y de como todas las piezas que conforman una aplicación de Rails se conectan entre sí.

## Controladores y vistas

Ahora que entendemos que en cada URL de nuestra aplicación existe un controlador que responde con una acción, vamos a examinar el controlador generado por el generador `scaffold`.

El controlador lo encontramos en el archivo `app/controllers/entries_controller.rb` al abrirlo, lo primero que notamos es que hay 6 métodos definidos en la parte pública de la clase - los métodos definidos antes de la palabra clave `private` -.

```
class EntriesController < ApplicationController

  before_action :set_entry, only: [:show, :edit, :update, :destroy]

  GET /entries

  GET /entries.json

  def index

    @entries = Entry.all
```

```
end
```

```
GET /entries/1
```

```
GET /entries/1.json
```

```
def show
```

```
end
```

```
GET /entries/new
```

```
def new
```

```
@entry = Entry.new
```

```
end
```

```
GET /entries/1/edit
```

```
def edit
```

```
end
```

```
POST /entries
```

```
POST /entries.json
```

```
def create
```

```
@entry = Entry.new(entry_params)
```

```
respond_to do |format|
```

```
  if @entry.save
```

```
    format.html { redirect_to @entry, notice: 'Entry was successfully created.' }
```

```
    format.json { render :show, status: :created, location: @entry }
```

```
  else
```

```
    format.html { render :new }
```

```
    format.json { render json: @entry.errors, status: :unprocessable_entity }
```

```
    end

end

end

PATCH/PUT /entries/1

PATCH/PUT /entries/1.json

def update

  respond_to do |format|

    if @entry.update(entry_params)

      format.html { redirect_to @entry, notice: 'Entry was successfully updated.' }

      format.json { render :show, status: :ok, location: @entry }

    else

      format.html { render :edit }

      format.json { render json: @entry.errors, status: :unprocessable_entity }

    end
  end
end
```

```
    end

end

end

DELETE /entries/1

DELETE /entries/1.json

def destroy

@entry.destroy

respond_to do |format|

  format.html { redirect_to entries_url, notice: 'Entry was successfully de-
stroyed.' }

  format.json { head :no_content }

end

end
```

```
private
```

Use callbacks to share common setup or constraints between actions.

```
def set_entry
```

```
  @entry = Entry.find(params[:id])
```

```
end
```

Never trust parameters from the scary internet, only allow the white list through.

```
def entry_params
```

```
  params.require(:entry).permit(:name, :notes)
```

```
end
```

```
end
```

Estos métodos son: [index](#), [show](#), [new](#), [edit](#), [create](#), [update](#) y [destroy](#). Cada método se mapea a una acción definidas en las rutas. De hecho y en forma de documentación en la definición de cada método existe un comentario que muestra como debe de ser la URL con la cual se accede a tal acción.

Cada acción tiene un propósito general y una o más posibles respuestas estándar. Las cuales obviamente se ajustan a las necesidades de nuestra aplicación.

- **index:** Es una acción que responde con el listado del recurso al que pertenece el controlador. La acción responde con una vista llamada `index.html.erb` que se encuentra en el folder `app/views/<nombre del controlador>` en el caso de nuestro controlador `EntriesController` la vista está en `app/views/entries/index.html.erb`
- **show:** La acción busca un registro existente para mostrar la información del recurso en modo de sólo lectura. La acción responde con la vista `show.html.erb`.
- **new:** Es la acción que despliega un formulario para crear un nuevo recurso, en nuestro caso un nuevo `Entry`. La acción responde con la vista `new.html.erb`.
- **edit:** Esta acción busca un registro existente y despliega un formulario con los datos del recurso para que puedan ser modificados. La acción responde con la vista `edit.html.erb`.
- **create:** La acción es invocada cuando se llena el formulario en la acción `new`, los datos del mismo son enviados y están disponibles para la acción a través de un objeto llamado `params`. Si el recurso es creado exitosamente el navegador es redireccionado a la acción `index`, si hay algún problema la vista de `new` es presentada en el navegador con la información del recurso y los mensajes de error para darle la oportunidad al usuario a corregirlos.
- **update:** Es la contraparte de `edit`, cuando se modifica un recurso la información del formulario es enviada a la acción `update`, ésta información está disponible en el objeto `params`. Si el recurso es modificado con éxito entonces el navegador es redireccionado a la acción `show` para mostrar la nueva información del recurso. Si existe algún problema, la acción responde con la vista `edit` mostrando la información del recurso así como los mensajes de error para darle la oportunidad al usuario de corregirlos.
- **destroy:** Es la acción que nos permite eliminar un registro específico. Una vez eliminado el registro la acción redirige el navegador a la acción de `index`.

Finalmente nos queda conocer un poco sobre las vistas y el papel que juegan en una aplicación de Rails. Como ya se mencionó anteriormente, las vistas se encuentran en el folder `app/views/<nombre del controlador>`, en el caso de nuestro ejemplo encontraremos 4 vistas para `EntriesController`: `index`, `new` y `edit`.

Cada una lleva el nombre de la acción primaria donde es utilizada en el controlador.

Las vistas llevan una extensión muy peculiar: `.html.erb`. Esto indica que las vistas son archivos de `html` y la parte `erb` hace notar que tienen incrustado código de Ruby, el cual es interpretado antes de enviar la respuesta final en puro `html` al navegador.

Veamos el ejemplo de la vista `index.html.erb`.

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

```
<h1>Listing Entries</h1>
```

```
<table>
```

```
<thead>
```

```
<tr>
```

```
<th>Name</th>
```

```
<th>Notes</th>
```

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

```
</tr>
```

```
</thead>
```

```
<tbody>
```

```
<% @entries.each do |entry| %>

<tr>

<td><%= entry.name %></td>

<td><%= entry.notes %></td>

<td><%= link_to 'Show', entry %></td>

<td><%= link_to 'Edit', edit_entry_path(entry) %></td>

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

</tr>

<% end %>

</tbody>

</table>

<br>
```

```
<%= link_to 'New Entry', new_entry_path %>
```

Vamos a encontrar secciones delimitadas por `<% %>`, el código dentro de estos indicadores es código de Ruby. También hay que notar que el controlador en la acción `index` define una variable de clase `@entries` y que esta variable está disponible para la vista de forma automática.

Otro elemento que nos vamos a encontrar en una vista son los `helpers`, los cuales sólo están disponibles en el contexto de las vista y que ayudan a la generación de código `html`, como por ejemplo `link_to` que sirve para crear links.

Si vemos el caso de las vistas `new` y `edit` notaremos que de contenido solo tienen código de Ruby con una llamada al `helper render` y como parámetro tiene `form`. En este caso `render` nos permite realizar una llamada a una vista `parcial`. Una vista parcial es un fragmento que por sí sólo no representa la respuesta de una acción de un controlador, pero que puede ser código que se puede reutilizar e incrustar en otras vistas.

Vista `new.html.erb`

```
<h1>New Entry</h1>
```

```
<%= render 'form' %>
```

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

Vista `_form.html.erb`

```
<%= form_for(@entry) do |f| %>
```

```
<% if @entry.errors.any? %>
```

```
<div id="error_explanation">
```

```
<h2><%= pluralize(@entry.errors.count, "error") %> prohibited this entry from  
being saved:</h2>  
  
<ul>  
  
<% @entry.errors.full_messages.each do |message| %>  
  
<li><%= message %></li>  
  
<% end %>  
  
</ul>  
  
</div>  
  
<% end %>  
  
<div class="field">  
  
<%= f.label :name %><br>  
  
<%= f.text_field :name %>  
  
</div>
```

```

<div class="field">

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

<%= f.text_area :notes %>

</div>

<div class="actions">

<%= f.submit %>

</div>

<% end %>

```

## Ejecutando nuestra aplicación

Ya que conocemos un poco más del código que se genera con el uso de `scaffold` es tiempo de ejecutar nuestra aplicación y ver el resultado de nuestro arduo trabajo.

```
$ bin/rails s
```

Apuntamos nuestro navegador a `http://localhost:3000/entries` y veremos que tenemos una solución completa para ver, crear, editar y eliminar - `CRUD` - registros de tipo `Entry`.

| Listing Entries           |   |                   |
|---------------------------|---|-------------------|
| Name                      | Notes   |                   |
| Pedro Páramo              | Primera entrada   | Show Edit Destroy |
| Mario Chavez              | - Que estoy haciendo? - Que voy a hacer? - Hay algo que me detiene? | Show Edit Destroy |
| <a href="#">New Entry</a> |   |                   |

Obviamente nuestra aplicación no es muy *atractiva* visualmente y hay varias cosas que aun podemos mejorar, pero si tomamos en cuenta que con un par de comandos llegamos a este punto no suena nada mal.

## Conclusiones

Durante éste capítulo conocimos un poco de la historia y los principios que hay detrás de Ruby on Rails. De igual forma tuvimos la oportunidad de crear una aplicación muy simple en cuestión de minutos, lo que nos permite darnos una pequeña idea de la mejora en productividad que nos ofrece un marco de trabajo como Rails.

La idea detrás del capítulo es simplemente familiarizarnos un poco con el marco de trabajo, ya que en los capítulos siguiente estaremos trabajando en una aplicación más grande en donde vamos a ver una serie de aspectos más avanzados en el desarrollo de aplicaciones Web.

La información de ayuda sobre Rails la podemos encontrar en muchos sitios en Internet, pero hay 3 recursos mínimos que son indispensables para cualquier desarrollador.

- [Guías de Ruby on Rails<sup>24</sup>](#)
- [API de Ruby on Rails<sup>25</sup>](#)
- [Código fuente de Rails<sup>26</sup>](#)

---

<sup>24</sup> <http://guides.rubyonrails.org/>

<sup>25</sup> <http://api.rubyonrails.org>

<sup>26</sup> <https://github.com/rails/rails>

# Capítulo 3 - Pruebas automáticas en Ruby

Como desarrolladores cada implementamos una nueva funcionalidad o hacemos cambios a una ya existente recurrimos a realizar pruebas para asegurarnos que el código que escribimos funciona tal y como esperamos.

Ésta prueba puede ser tan sencilla como ejecutar nuestro código con una serie de parámetros de pruebas y observar si el resultado obtenido es el esperado.

Probar de esta forma es muy simple cuando nuestro programa es muy pequeño y somos los únicos que hacemos cambios. Pero es una situación insostenible fuera de éstas condiciones.

## Pruebas automáticas

Escribir pruebas automáticas para nuestros programas implica cambiar un poco nuestra forma de desarrollar software.

Pero antes de entender el por qué cambia nuestra forma de desarrollar software vamos a entender un poco que son las pruebas automáticas.

En términos simples, una prueba automática no es otra cosa que escribir código cuya única finalidad es probar que nuestro código de cierta funcionalidad en nuestra aplicación se comporte tal y como queremos.

El escribir pruebas automáticas nos ofrece una serie de beneficios como el ayudarnos a detectar errores introducidos por nuevos cambios en nuestro programa, si bien las pruebas no van a detectar todos los errores posibles, es más sencillo detectar efectos colaterales.

Otro de los beneficios es el poder probar de forma relativamente rápida que nuestra aplicación se comporta tal y como esperamos con pruebas que son repetitivas y que siempre se ejecutan con las mismas condiciones a diferencia de realizar pruebas de forma manual.

A final de cuentas el objetivo de escribir pruebas automáticas es lograr a que nos ayude a tener una aplicación más mantenible y que nos de un cierto nivel de seguridad el publicar cambios a producción.

## Pruebas automáticas y Ruby on Rails

Ruby on Rails favorece el escribir pruebas automáticas para nuestras aplicaciones. El escribir este tipo de pruebas está de alguna forma en el *DNA* del marco de trabajo.

En la aplicaciones de ejemplo que creamos en el capítulo pasado si observamos los folder creados por el generador vamos a encontrar el folder de *test*. Éste folder alberga el código base para empezar a escribir pruebas pruebas automáticas.

Inclusive el generador de *scaffold* generó un par de archivos para escribir pruebas para el modelo *entry* y el controlador *entries*.

Ruby on Rails, desde la versión 4.0 en adelante, utiliza **Minitest**, el cuál en una librería que ayuda a escribir y ejecutar pruebas automáticas en nuestras aplicaciones.

### Anatomía de una prueba automática

Una prueba automática esta comprendida de un patrón de 3 partes:

1. Configuración para el ambiente de nuestra prueba
2. Ejecutar lo que queremos probar
3. Verificar que el resultado del paso anterior es el resultado esperado

La forma en como podemos identificar que debe ir en cada una de las 3 partes anteriormente mencionadas es respondiendo las siguientes preguntas:

1. ¿Cómo configuro la prueba?
2. ¿Qué es lo que estoy probando?
3. ¿Cuál es la respuesta que espero?

Pasemos a nuestro primer ejemplo, el cuál no s ayudará a darnos una idea un poco más clara de como implementar una prueba automática.

Dado que nuestro objetivo final es trabajar con Ruby on Rails, vamos a utilizar **Minitest** como nuestra herramienta de prueba.

Empecemos en crear un archivo llamado `uppercase_test.rb`, en el archivo vamos a iniciar con el siguiente código:

```
require 'minitest/autorun'

class UppercaseTest < Minitest::Test

  def test_transform_text_to_uppercase
```

Configuración de la prueba

```
    downcase_text = 'abcd'
```

Ejecución

```
    uppercase_text = downcase_text.upcase
```

Verificación

```
    assert_equal 'ABCD', uppercase_text
```

```
  end
```

```
end
```

Antes de analizar que es lo que va a hacer nuestra prueba vamos observando algunos detalles importantes. La primera línea le indica a Ruby que deseamos cargar la librería de **Minitest** en nuestro programa y al mismo tiempo **Minitest** queda preparado para ejecutar nuestra prueba.

Cada grupo de pruebas afines van dentro la cual tiene como padre a la clase **Minitest::Test**. Ésta clase de **Minitest** define una serie de métodos llamados *afirmaciones* que nos va a permitir validar el resultado en nuestras pruebas.

En la clase de prueba definimos métodos con un nombre que nos indique la naturaleza de la prueba. Éstos métodos deben de tener como prefijo la palabra *test*.

Dentro del método, como podemos ver, seguimos las 3 reglas descritas anteriormente, contamos con una configuración para nuestra prueba, la ejecución de lo que queremos probar - en muchos de los casos a esto también se le conoce como el *sujeto bajo prueba* - y finalmente la verificación.

En el caso de la verificación **assert\_equal** es una de las afirmaciones de **Minitest**, la cual nos sirve para indicar que esperamos que 2 valores sean iguales. El patrón en todas las verificaciones es que primero indicamos el valor que esperamos, como en nuestro caso '**'ABCD'**' y posteriormente va el resultado de la ejecución.

Si el resultado de **assert\_equal** es verdadero entonces nuestra prueba va a ser exitosa:

```
$ ruby uppercase_test.rb -v
```

```
Run options: -v --seed 52540
```

Running:

```
UppercaseTest#test_transform_text_to_uppercase = 0.00 s = .
```

```
Finished in 0.001480s, 2027.0270 runs/s, 2027.0270 assertions/s.
```

```
1 runs, 1 assertions, 0 failures, 0 errors, 0 skips
```

Si el resultado de `assert_equal` es falso entonces nuestra prueba va a fallar. Vamos a cambiar nuestra expectativa en nuestra prueba de la siguiente forma:

```
require 'minitest/autorun'
```

```
class UppercaseTest < Minitest:: Test
```

```
def test_transform_text_to_uppercase
```

Configuración de la prueba

```
downcase_text = 'abcd'
```

Ejecución

```
uppercase_text = downcase_text.upcase
```

## Verificación

```
assert_equal 'ZXCV', uppercase_text
```

```
end
```

```
end
```

Y ejecutamos nuestra prueba:

```
$ ruby uppercase_test.rb -v
```

```
Run options: -v --seed 23704
```

Running:

```
UppercaseTest#test_transform_text_to_uppercase = 0.00 s = F
```

```
Finished in 0.001634s, 1835.9853 runs/s, 1835.9853 assertions/s.
```

1) Failure:

```
UppercaseTest#test_transform_text_to_uppercase [uppercase_test.rb:7]:
```

```
Expected: "ZXCV"
```

```
Actual: "ABCD"
```

```
1 runs, 1 assertions, 1 failures, 0 errors, 0 skips
```

Ahora **Minitest** nos indica que la prueba falló, pero además nos da información de que prueba en particular, que línea de código y que es lo que se esperaba y cual es el valor actual.

### *Sintaxis Spec*

La sintaxis que utilizamos para escribir nuestra prueba se le conoce como **Test::Unit**, esto debido a que es una sintaxis compatible con una herramienta, del mismo nombre, para realizar pruebas automáticas en Ruby, pero que ya no se utiliza más. **Minitest** adoptó la sintaxis por cuestiones de compatibilidad.

Pero hay otra sintaxis llamada **Spec** que popularizó otra herramienta para pruebas automáticas en Ruby llamada **RSpec**. **Minitest** también adoptó la sintaxis de forma que las pruebas de **Minitest** pueden tener una u otra sintaxis.

Vamos a reescribir nuestra prueba en la sintaxis **RSpec**:

```
require 'minitest/autorun'
```

```
describe 'Uppercase test' do
```

```
it 'transform text to uppercase' do
```

Configuración de la prueba

```
downcase_text = 'abcd'
```

Ejecución

```
uppercase_text = downcase_text.upcase
```

Verificación

```
uppercase_text.must_equal 'ABCD'
```

```
end
```

```
end
```

Para ejecutarla hacemos lo mismo que con la otra sintaxis:

```
$ ruby uppercase_test.rb -v
```

```
Run options: -v --seed 28714
```

Running:

```
Uppercase test#test_0001_transform text to uppercase = 0.00 s = .
```

```
Finished in 0.001480s, 2027.0270 runs/s, 2027.0270 assertions/s.
```

```
1 runs, 1 assertions, 0 failures, 0 errors, 0 skips
```

El resultado es lo mismo, no hay diferencia para [Minitest](#). Pero la diferencia de sintaxis si es significativa, aunque el espíritu de la prueba sigue siendo el mismo.

La decisión de que sintaxis utilizar, creo que no es técnica, pero si cuestión de gusto personal del desarrollador o de las políticas definidas por el equipo de desarrollo.

## Pruebas automáticas como una filosofía

Desde que inicie en el mundo de Ruby comprendí que ser Rubysta significa más que únicamente escribir código de Ruby, significa también el adquirir habilidades para escribir pruebas como parte del proceso de desarrollo, hasta el punto en que ya no haces diferencia entre escribir código con funcionalidad para la aplicación y escribir pruebas para el mismo.

Esta idea de ser Rubysta y escribir pruebas es intrínseca de igual forma en Ruby on Rails, ya que como vimos, desde el momento que generamos una aplicación, el marco de trabajo también genera el código y la configuración necesaria para empezar a escribir pruebas.

Escribir pruebas en Ruby es un aspecto técnico pero también es un aspecto filosófico de la comunidad.

Entre los Rubystas es común el uso de la técnica **Test Driven Development** o **TDD**. Esta técnica fue desarrollada por Kent Beck<sup>1</sup> en la década de los 90's como parte de las herramientas de **Extreme Programming**.

**TDD** se basa en el razonamiento de que a través de escribir las pruebas primero antes de realizar de implementación y de hacer lo mínimo necesario para hacer que las pruebas pases, uno como desarrollador obtiene los siguientes beneficios:

- El trabajar con confianza
- Trabajar en una serie de avances alcanzables y medibles que nos den noción de progreso en lugar de atacar problemas grandes de un sólo golpe
- Asegurarse de que nuestro código se apega a los requerimientos
- Respaldarnos de que nuestras pruebas nos ayuden a preservar la integridad de nuestro código.

Un conjunto de pruebas en un proyecto de software se le conoce como un **Suite** de pruebas.

La forma en como **TDD** es llevado a cabo es a través del concepto **Red–Green–Refactor**.

**Red–Green–Refactor** implica seguir los siguientes pasos:

1. Escribir una prueba de acuerdo a la especificación sin escribir código para hacerla pasar aún. Éste es el paso **Red** donde nuestra prueba va a fallar.
2. Escribir la implementación en el código de producción para hacer que nuestra pase. Éste es el paso **Green** donde nuestra prueba va a pasar.
3. **Refactor** significa revisar el código de producción escrito y realizar una serie de transformaciones para *limpiarlo* y mejorarlo en calidad, siempre con la confianza de que los cambios realizados van a estar respaldados por la prueba que escribimos.

El ciclo de **TDD** consiste en seguir estos 3 pasos de forma repetitiva durante nuestro proceso de desarrollo de software.

---

<sup>1</sup> [http://en.wikipedia.org/wiki/Kent\\_Beck](http://en.wikipedia.org/wiki/Kent_Beck)

## Ejemplo de TDD

Vamos a realizar un pequeño ejercicio de **TDD**.

Nuestro requerimiento nos pide escribir una clase que contenga un método que pueda recibir como parámetro un arreglo de números no ordenados. El método debe de regresar un nuevo arreglo con los números ordenados de forma ascendente.

Iniciemos escribiendo nuestra primera prueba, para tal efecto vamos a crear el archivo `sort_test.rb`

```
require 'minitest/autorun'

class SortTest < Minitest::Test

  def test_unsorted_array_is_being_sorted_example1

    unsorted_array = [4, 8, 7, 6, 1, 5]

    sorter = Sorter.new unsorted_array

    assert_equal [1, 4, 5, 6, 7, 8], sorter.sortAscending

  end
end
```

```
end
```

El resultado que obtenemos de ejecutar nuestra prueba es **Red** es decir, nuestra prueba falló.

```
$ ruby sort_test.rb -v
```

```
Run options: -v --seed 32345
```

Running:

```
SortTest#test_unsorted_array_is_being_sorted_example1 = 0.00 s = E
```

```
Finished in 0.001247s, 801.9246 runs/s, 0.0000 assertions/s.
```

1) Error:

```
SortTest#test_unsorted_array_is_being_sorted_example1:
```

```
NameError: uninitialized constant SortTest::Sorter
```

```
sort_test.rb:34:in `test_unsorted_array_is_being_sorted_example1'
```

```
1 runs, 0 assertions, 0 failures, 1 errors, 0 skips
```

El error que obtenemos es que no existe una constante llamada `SortTest::Sorter`. Vamos a hacer que nuestra prueba pase.

```
require 'minitest/autorun'
```

```
class Sorter
```

```
  def initialize(array)
```

```
    end
```

```
    def sortAscending
```

```
      [1, 4, 5, 6, 7, 8]
```

```
    end
```

```
  end
```

```
class SortTest < Minitest::Test

  def test_unsorted_array_is_being_sorted_example1

    unsorted_array = [4, 8, 7, 6, 1, 5]

    sorter = Sorter.new unsorted_array

    assert_equal [1, 4, 5, 6, 7, 8], sorter.sortAscending

  end

end
```

Si ejecutamos nuestra prueba después de agregar la clase `Sort` nuestra prueba debe de pasar y estará en modo **Green**.

```
$ ruby sort_test.rb -v
```

```
Run options: -v --seed 2319
```

Running:

```
SortTest#test_unsorted_array_is_being_sorted_example1 = 0.00 s = .
```

```
Finished in 0.001556s, 642.6735 runs/s, 642.6735 assertions/s.
```

```
1 runs, 1 assertions, 0 failures, 0 errors, 0 skips
```

Pero obviamente sólo hicimos lo mínimo necesario para hacer la que primera prueba pasara. Y como podemos ver, el código de la clase `Sort` no sirve para nada más que hacer que esa prueba pase. Vamos a comprobarlo agregando una prueba adicional.

```
require 'minitest/autorun'
```

```
class Sorter
```

```
  def initialize(array)
```

```
  end
```

```
  def sortAscending
```

```
[1, 4, 5, 6, 7, 8]

end

end

class SortTest < Minitest::Test

def test_unsorted_array_is_being_sorted_example1

  unsorted_array = [4, 8, 7, 6, 1, 5]

  sorter = Sorter.new unsorted_array

  assert_equal [1, 4, 5, 6, 7, 8], sorter.sortAscending

end

def test_unsorted_array_is_being_sorted_example2
```

```
unsorted_array = [6, 2, 9, 8, 0, 4]

sorter = Sorter.new unsorted_array

assert_equal [0, 2, 4, 6, 8, 9], sorter.sortAscending

end

end
```

Al ejecutar nuestras pruebas, vemos que la segunda prueba falla:

```
$ ruby sort_test.rb -v
```

```
Run options: -v --seed 27951
```

Running:

```
SortTest#test_unsorted_array_is_being_sorted_example1 = 0.00 s = .
```

```
SortTest#test_unsorted_array_is_being_sorted_example2 = 0.00 s = F
```

```
Finished in 0.001739s, 1150.0863 runs/s, 1150.0863 assertions/s.
```

### 1) Failure:

```
SortTest#test_unsorted_array_is_being_sorted_example2 [sort_test.rb:53]:
```

```
Expected: [0, 2, 4, 6, 8, 9]
```

```
Actual: [1, 4, 5, 6, 7, 8]
```

```
2 runs, 2 assertions, 1 failures, 0 errors, 0 skips
```

Es momento de realizar nuestra implementación para ordenar un arreglo de números. En nuestro ejemplo vamos a utilizar un algoritmo muy sencillo llamado **Simple Card Sort**<sup>2</sup> no es un algoritmo eficiente ni en tiempo ni en memoria, pero nos dará el resultado que deseamos.

La implementación se hará con una pequeña modificación al algoritmo original, en donde en lugar de marcar el espacio de un número menor con el número mayor en el arreglo, vamos a eliminar la posición del número menor una vez que ha sido copiado al nuevo arreglo ordenado.

```
require 'minitest/autorun'
```

---

<sup>2</sup> <http://courses.cs.vt.edu/csonline/Algorithms/Lessons/SimpleCardSort/index.html>

```
class Sorter

  def initialize(array)

    @array = array.clone

    @new_array = []

  end

  def sort_descending(array = @array)

    return if array.empty?

    min = nil

    min_index = nil

    array.each_with_index do |item, index|

      if min.nil? || item < min

        min = item

        min_index = index

      end

    end

    @new_array = array.map { |item| item == min ? nil : item }

    @new_array.insert(min_index, min)

  end

  def sort_descending!(array = @array)

    sort_descending(array)

    @array = @new_array

  end

end
```

```
min = item

min_index = index

end

end

array.delete_at min_index

@new_array << min

sortAscending array

@new_array

end

end

class SortTest < Minitest::Test
```

```
def test_unsorted_array_is_being_sorted_example1

unsorted_array = [4, 8, 7, 6, 1, 5]

sorter = Sorter.new unsorted_array

assert_equal [1, 4, 5, 6, 7, 8], sorter.sortAscending

end

def test_unsorted_array_is_being_sorted_example2

unsorted_array = [6, 2, 9, 8, 0, 4]

sorter = Sorter.new unsorted_array

assert_equal [0, 2, 4, 6, 8, 9], sorter.sortAscending
```

```
end
```

```
end
```

Después de finalizar nuestra implementación, ejecutamos nuestro **Suite** de pruebas y vemos como ambas pruebas pasan. Nuestra implementación ha sido un éxito.

```
$ ruby sort_test.rb -v
```

```
Run options: -v --seed 36137
```

```
Running:
```

```
SortTest#test_unsorted_array_is_being_sorted_example2 = 0.00 s = .
```

```
SortTest#test_unsorted_array_is_being_sorted_example1 = 0.00 s = .
```

```
Finished in 0.001227s, 1629.9919 runs/s, 1629.9919 assertions/s.
```

```
2 runs, 2 assertions, 0 failures, 0 errors, 0 skips
```

Hasta éste momento hemos únicamente ejecutado los pasos 1 y 2 de **TDD**, vamos a realizar el tercero que implica *refactorización*, es decir vamos a ver si podemos mejorar nuestra implementa-

ción actual, pero lo vamos a hacer con la confianza de que tenemos un **Suite** de pruebas que nos respalden.

```
require 'minitest/autorun'

class Sorter

  def initialize(array)

    @array = array.clone

    @new_array = []

  end

  def sortAscending(array = @array)

    return if array.empty?

    minimum_value = nil

    value_index = nil
```

```
array.each_with_index do |item, index|
  minimum_value, value_index = [item, index] if minimum?(item, minimum_value)
end

array.delete_at value_index

@new_array << minimum_value

sortAscending array

@new_array

end

private

def minimum?(value, compare_to)
```

```
compare_to.nil? || value < compare_to

end

end

class SortTest < Minitest::Test

def test_unsorted_array_is_being_sorted_example1

unsorted_array = [4, 8, 7, 6, 1, 5]

sorter = Sorter.new unsorted_array

assert_equal [1, 4, 5, 6, 7, 8], sorter.sortAscending

end

def test_unsorted_array_is_being_sorted_example2
```

```
unsorted_array = [6, 2, 9, 8, 0, 4]

sorter = Sorter.new unsorted_array

assert_equal [0, 2, 4, 6, 8, 9], sorter.sortAscending

end

end
```

En nuestro paso de *refactorización* renombramos algunas variables para hacerlas más explícitas y movimos código a métodos privados en nuestra clase `Sort`. En teoría mejoramos nuestra implementación.

Al ejecutar nuestro `Suite` de pruebas vemos que nuestros cambios no afectaron al resultado esperado.

```
$ ruby sort_test.rb -v
```

```
Run options: -v --seed 15605
```

Running:

```
SortTest#test_unsorted_array_is_being_sorted_example1 = 0.00 s = .
```

```
SortTest#test_unsorted_array_is_being_sorted_example2 = 0.00 s = .
```

```
Finished in 0.001364s, 1466.2757 runs/s, 1466.2757 assertions/s.
```

```
2 runs, 2 assertions, 0 failures, 0 errors, 0 skips
```

Unas semanas después tenemos un reporte que dice que al ordenar números nuestra aplicación tarda *mucho tiempo*. Nuestra implementación no es la mejor, tenemos que hacer algo al respecto, afortunadamente tenemos pruebas que nos van a ayudar en esta *refactorización*.

```
require 'minitest/autorun'
```

```
class Sorter
```

```
  def initialize(array)
```

```
    @array = array
```

```
  end
```

```
def sortAscending

  @array.sort

end

end

class SortTest < Minitest::Test

  def test_unsorted_array_is_being_sorted_example1

    unsorted_array = [4, 8, 7, 6, 1, 5]

    sorter = Sorter.new unsorted_array

    assert_equal [1, 4, 5, 6, 7, 8], sorter.sortAscending

  end

```

```
def test_unsorted_array_is_being_sorted_example2

  unsorted_array = [6, 2, 9, 8, 0, 4]

  sorter = Sorter.new unsorted_array

  assert_equal [0, 2, 4, 6, 8, 9], sorter.sortAscending

end

end
```

Acabamos de mejorar nuestra implementación, es momento de ejecutar nuestras pruebas automáticas.

```
$ ruby sort_test.rb -v
```

```
Run options: -v --seed 56895
```

Running:

```
SortTest#test_unsorted_array_is_being_sorted_example1 = 0.00 s = .
```

```
SortTest#test_unsorted_array_is_being_sorted_example2 = 0.00 s = .
```

```
Finished in 0.001281s, 1561.2802 runs/s, 1561.2802 assertions/s.
```

```
2 runs, 2 assertions, 0 failures, 0 errors, 0 skips
```

Todo pasa correctamente. Aún y cuando nuestra clase `Sorter` recibió un cambio bastante drástico. Es por eso que es importante que nuestras pruebas sólo se enfoquen en el resultado del sujeto bajo prueba y no de la forma en como se realizó la implementación.

## Conclusiones

Escribir pruebas automáticas se escucha más sencillo de lo que realmente es, en especial si como desarrolladores nunca habíamos hecho algo como esto. La pregunta recurrente de gente que llega por primera vez al concepto de pruebas automáticas es ¿Cómo decido qué es lo que tengo que probar? y la verdad la respuesta no es simple.

Generalmente toma tiempo el poder determinar que es lo que hay probar. Hay ejercicios como `Katas`<sup>3</sup> que ayudan a crear la costumbre de realizar pruebas para resolver problemas.

La otra parte viene de la experiencia desarrollando software bajo estos principios y experimentar de forma directa los beneficios de escribir pruebas.

A través de el resto de los capítulos del libro vamos a desarrollar una aplicación para entender los conceptos de desarrollar una aplicación con Ruby on Rails en donde vamos a necesitar de los conceptos de escribir pruebas automáticas y realizar `refactorizaciones` conforme nuestra aplicación se va completando capítulo a capítulo.

---

<sup>3</sup> <http://codekata.com/>

# Capítulo 4 - Desarrollando una aplicación real en Ruby on Rails

A partir de este capítulo empezaremos el desarrollo de una aplicación de Ruby on Rails realista, es decir, utilizaremos las técnicas de desarrollo ágil para ir agregando funcionalidad de forma vertical y de manera incremental.

Durante el desarrollo de la aplicación exploraremos áreas de Ruby on Rails, donde extendemos un poco la funcionalidad original del marco de trabajo, con la finalidad de que nuestro código sea un poco más mantenible, objetivo que lograremos también con el uso de pruebas automáticas para toda la funcionalidad de la aplicación.

El desarrollo lo iniciaremos bajo el concepto de *afuera hacia adentro*, es decir trabajemos a partir de historias de usuarios que nos permitirán validar que estamos construyendo la funcionalidad necesaria para nuestra aplicación. Al realizar el desarrollo de esta forma, nos adentraremos en el concepto de **Behaviour Driven Development** o **BDD**.

**BDD** es otra técnica de pruebas de automáticas que nos permite enfocarnos en definir, probar y entregar funcionalidad es completas en una aplicación. Para tal efecto traeremos otras herramientas de pruebas para incorporarlas a lo que ya conocemos de **Minitest**.

## La aplicación

Durante el resto del libro trabajaremos en desarrollar una aplicación para la publicación de imágenes. A través de la aplicación un usuario se va a poder registrar y empezar a publicar y compartir imágenes. Por me dio de la misma aplicación un usuario va a poder descubrir, comentar y agregar a favoritas las imágenes que más le agraden.

Todas las imágenes para poder mostrarse en pantalla, requerirán de procesamiento posterior al momento de publicación, éste procesamiento se realizará a través de *background jobs*.

La aplicación nos permitirá explorar características específicas de Ruby on Rails que nos ayudaran a entender mejor como utilizarlo para nuestros proyectos.

## Control de código fuente

El código fuente de la aplicación lo manejaremos con **Git** que hoy en día es uno de los administradores de código fuente más populares. Si no conoces de **Git** es buen momento para hacer una pausa y revisar el libro Pro Git<sup>1</sup>.

Como servicio de repositorio de código fuente utilizaremos Github<sup>2</sup>, cada capítulo estará contenido en su propia **branch** o **rama** para que sea fácil identificar que el trabajo realizado capítulo a capítulo.

El repositorio en Github para la aplicación es OnePx<sup>3</sup>.

## Iniciando con la aplicación

Ha llegado el momento de iniciar nuestra aplicación de Ruby on Rails. Como paso inicial crearemos un proyecto nuevo de Rails, al cuál vamos a realizarle algunos ajustes iniciales que nos ayudaran durante el desarrollo de nuestra aplicación.

Ruby on Rails es agnóstico al motor de base de datos que se utilice. Dos de los motores de base de datos más populares son **MySQL** y **Postgresql**. Siendo éste último el motor de base de datos que vamos a utilizar en nuestro desarrollo.

## *La base de datos*

En versiones recientes de Rails el soporte de funcionalidad nativa de **Postgresql** ha permeado en la funcionalidad de Rails mismo, un ejemplo de esto es el soporte para el tipo de datos **Json** y **HStore** que permiten almacenar información de forma no estructurada, similar a la funcionalidad que obtenemos de bases de datos **NoSQL**.

Por este motivo vamos a utilizar **Postgresql** como base de datos para nuestra aplicación. El como instalar **Postgresql** se sale del alcance del libro pero en el wiki de Postgresql.org hay guías<sup>4</sup> de instalación para diferentes sistemas operativos.

---

<sup>1</sup> <http://progit.org/book>

<sup>2</sup> <http://github.com>

<sup>3</sup> <http://www.github.com/mariochavez/onepx>

<sup>4</sup> [https://wiki.postgresql.org/wiki/Detailed\\_installation\\_guides](https://wiki.postgresql.org/wiki/Detailed_installation_guides)

En el caso de los usuarios de [OSX](#) la instalación es muy sencilla con [Postgresapp](#)<sup>5</sup>.

La versión de [Postgresql](#) que estaremos utilizando es 9.4 con el soporte de la extensión [HStore](#)<sup>6</sup>.

## *Creando el proyecto*

Como ya vimos anteriormente, vamos a iniciar con el comando `rails` que nos permitirá crear el esqueleto inicial de nuestra aplicación. A diferencia de como lo utilizamos anteriormente vamos a agregar una opción para indicarle a Rails que configure automáticamente la base de datos para nosotros.

El nombre de nuestro proyecto será [OnePx](#) - como ya mencionamos anteriormente -, por lo que un nombre adecuado para Rails será `onepx`.

```
$ rails new one_px -d postgresql
```

Una vez que el generador terminó de crear nuestro proyecto, nuestro primer paso va a ser realizar nuestro primer `commit` de [Git](#) para empezar a crear historia de nuestro proyecto.

```
$ cd one_px
```

```
$ git init
```

```
$ git add .
```

```
$ git commit -m "Project bootstrap"
```

El comando `git init` inicializa un repositorio de [Git](#) en el folder de nuestra aplicación, el segundo comando `git add .` agrega todos los archivos de proyecto a control de código fuente de [Git](#) y finalmente `git commit -m "Project bootstrap"` guarda el estado de nuestros archivo en ese momento específico con el mensaje `Project bootstrap`.

---

<sup>5</sup> <http://postgresapp.com>

<sup>6</sup> <http://www.postgresql.org/docs/9.4/static/hstore.html>

Si ejecutamos el comando `git log` nos muestra el historial de cambios que hemos registrado en nuestro control de código fuente.

```
$ git log
```

```
commit 740f58d8278edb17ad385f29446ccd0d5cc2b698
```

```
Author: Mario Alberto Chávez <mario.chavez@gmail.com>
```

```
Date: Tue Nov 3 14:59:34 2015 -0600
```

#### Proyect bootstrap

En este punto ya tenemos un proyecto de Rails listo para que podamos comenzar a trabajar en él, pero antes es necesario realizar algunos cambios y agregar algunas gemas que nos ayuden a trabajar de la mejor manera.

El primer cambio consiste configurar el archivo `config/database.yml` de acuerdo a nuestro servidor de [Postgresql](#). En el caso de mi servidor únicamente necesito especificar el `host` o la dirección del servidor, depende de como se configuró el servidor es posible que requiera de usuario y clave.

```
default: &default
```

```
adapter: postgresql
```

```
host: localhost
```

```
encoding: unicode
```

```
# For details on connection pooling, see rails configuration guide  
  
# http://guides.rubyonrails.org/configuring.html#database-pooling  
  
pool: 5
```

En la sección de `default` es donde se agrega la entrada de `host` como se muestra en el ejemplo anterior. Si fuera necesario agregar `username` y `password` lo podemos agregar en esta misma sección.

Una vez que realizamos los cambios, vamos a comprobar que nuestra aplicación de Rails se puede conectar a nuestro servidor de *Postgresql*.

```
$ rake db:create
```

`rake db:create` se conecta al servidor y crea una base de datos con el nombre especificado en `database` en la sección de `development` del archivo `config/database.yml`

Si la configuración es correcta el comando se ejecuta sin mostrar ningún mensaje.

Vamos a realizar otro `commit` para enviar el cambios a nuestro sistema de control de cambios.

```
$ git add config/database.yml
```

```
$ git commit -m "Database configuration"
```

El siguiente ajuste consiste en configurar el archivo `Gemfile`. Este archivo es un manifiesto de las dependencias de nuestra aplicación. En el archivo `Gemfile` especificamos las gemas que son requeridas en nuestro proyecto y en algunos casos inclusive podemos indicar versiones específicas o mínimas de las gemas requeridas.

Al abrir el archivo `Gemfile` vemos que ya define el uso de ciertas gemas. Éstas son las gemas mínimas requeridas para una aplicación de Ruby On Rails, pero como veremos en el desarrollo de nuestra aplicación iremos agregando algunas más que nos ayudaran en ciertas áreas de nuestro proyecto.

Vemos que algunas gemas se encuentran agrupadas en secciones delimitadas por la palabra `group` y que en ésta agrupación indica los ambientes en los cuales las gemas van a estar activas.

Por ejemplo un `group` de `:development` y/o `:test` indica que las gemas estarán activas únicamente esos ambientes.

El archivo `Gemfile` es utilizado por el comando `bundle`. `bundle` es un comando parte de *Bundler*, y se encarga de leer el manifesto, resolver las dependencias de las gemas, así como sus versiones y finalmente instalar las gemas que no se encuentran disponibles en nuestro ambiente.

*Bundler* se conecta con el directorio RubyGems<sup>7</sup> para resolver las dependencias e instalar las gemas necesarias para cumplir con las especificaciones del manifiesto.

Una vez que el comando `bundle` finaliza, el archivo `Gemfile.lock` es creado automáticamente y a diferencia del archivo `Gemfile`, `Gemfile.lock` contiene el listado específico de las gemas, dependencias y versiones que nuestra aplicación utilizará.

Vamos a realizar una reorganización del archivo `Gemfile`, primeramente para eliminar los comentarios, agregar las gemas que vamos a necesitar en la primera etapa de nuestro proyecto y finalmente organizar las gemas por orden alfabético. Éste último paso no es esencialmente necesario, ya que no hace diferencia en el funcionamiento de nuestra aplicación.

Nuestro archivo `Gemfile` queda de la siguiente forma:

```
source 'https://rubygems.org'

gem 'coffee-rails', '~> 4.1.0'

gem 'jquery-rails'

gem 'turbolinks', github: 'rails/turbolinks'

gem 'pg'

gem 'pretty_formatter'
```

---

<sup>7</sup> <http://rubygems.org>

```
gem 'puma'

gem 'rails', '4.2.4'

gem 'sass-rails', '~> 5.0'

gem 'uglifier', '>= 1.3.0'

group :development do

  gem 'pry-rails'

  gem 'spring'

  gem 'web-console', '~> 2.0'

end

group :development, :test do

  gem 'byebug'

  gem 'minitest-rails'
```

```
gem 'minitest-rails-capybara'

gem 'selenium-webdriver'

end
```

Como vemos en el contenido del archivo, podemos declarar las dependencias a nuestras librerías indicando la versión en la que estamos interesados en utilizar o como en el caso de [turbo-links](#) podemos indicar que deseamos utilizar la librería directamente desde su repositorio en [GitHub](#).

Una vez modificado nuestro [Gemfile](#), sólo resta ejecutar el comando [bundle](#) para los cambios entren en efecto.

```
$ bundle
```

Al final de la ejecución de [bundle](#) esperamos ver la confirmación de que las dependencias se instalaron y configuraron correctamente.

```
Bundle complete! 17 Gemfile dependencies, 69 gems now installed.
```

```
Use bundle show [gemname] to see where a bundled gem is installed.
```

Uno de los cambios a notar que realizamos en nuestro [Gemfile](#), es la adición de la gema de [Puma](#)<sup>8</sup>. [Puma](#) es un servidor web rápido y concurrente.

Utilizaremos [Puma](#) ya que es un servidor propio para poner aplicaciones de Ruby on Rails en producción, a diferencia de [WEBrick](#), el servidor web que utilizan las aplicaciones de Rails por omisión, el cuál sólo es apto para desarrollo.

Hacemos el cambio de servidor web en este momento, debido a que es común olvidar este paso y poner aplicaciones de Rails con [WEBrick](#) en producción con lo que obtenemos un bajo desempeño.

---

<sup>8</sup> <http://puma.io>

Otra de las gemas que agregamos a nuestra aplicación es `minitest-rails`, que como ya vimos `Minitest` es una herramienta para desarrollar y ejecutar pruebas automáticas en una aplicación. En este caso `minitest-rails` hace una integración más estrecha con nuestro proyecto de Rails.

`minitest-rails` incluye un generador que nos ayuda a configurar el ambiente de pruebas.

Es común que gemas en Rails incluyan generadores. Para conocer los generadores disponibles en nuestra aplicación podemos ejecutar el siguiente comando:

```
$ rails g
```

Conforme avancemos con el desarrollo de nuestra aplicación haremos uso de más de uno de los generadores listados. Por ahora vamos a ejecutar el generador de `Minitest`.

```
$ rails g minitest:install
```

Una alerta nos indicará que el archivo `test/test_helper.rb` ya existe y que el generador lo reemplazará, presionamos `Y` para indicar que sí se reemplace.

Ahora deseamos indicarle a Rails que queremos que utilice a `Minitest` como la herramienta de pruebas por omisión, por lo que vamos a crear un archivo inicializador y especificaremos ahí la configuración.

Rails cuenta con varios archivos de inicialización, los cuales son utilizados para configurar una aplicación de Rails para trabajar de cierta forma. Los archivos de inicialización los encontramos en `config/initialization`.

Vamos a crear el archivo `config/initialization/generators.rb` y le agregamos el siguiente contenido:

```
Rails.application.config.generators do |g|  
  
  g.test_framework :minitest, spec: false, fixture: true  
  
end
```

Finalmente abrimos el archivo `test/test_helper.rb` que agregó el generador de `Minitest` y le quitamos el comentario a la línea que dice `# require "minitest/rails/capybara"`.

Con esto activaremos `Capybara`<sup>9</sup> para realizar pruebas de tipo de integración, es decir crearemos pruebas que simularan la presencia de un usuario en el navegador mientras hace uso de nuestra aplicación.

Para hacer uso de `Capybara` requerimos de tener instalada la última versión de `Firefox`.

En este punto ha llegado el momento de realizar un nuevo `commit`.

```
$ git add .
```

```
$ git commit -m "Initial configuration"
```

## Registro de usuarios, nuestra primera funcionalidad

Ha llegado el momento de trabajar en nuestra primera funcionalidad `Users signup`.

Antes de proceder a escribir una línea de código vamos escribiendo una especificación de como debe de operar la funcionalidad.

### *Escribiendo nuestras primeras pruebas*

Un usuario nuevo debe de poder navegar a la `URL /registro` de nuestra aplicación, en ésta dirección al usuario se le mostrará un formulario para poder ingresar los siguientes datos y completar su registro:

- Correo electrónico, solamente un nuevo registro por correo electrónico es posible.
- Contraseña, una contraseña de mínimo 8 caracteres.

---

<sup>9</sup> <https://jnicklas.github.io/capybara/>

- Confirmación de contraseña, la cual debe coincidir con la contraseña ingresada anteriormente.

El formulario contará con un botón llamado *Crear cuenta* que al ser presionado se intentará registrar al usuario, en caso de error se deberán mostrar los mensajes correspondientes para que usuario pueda corregirlos y volver a intentar.

En caso de éxito se redirigirá al usuario a `/` donde se le mostrará un mensaje de bienvenida.

Con esta información vamos creando nuestra primera prueba de integración con la cuál partiremos para desarrollar la lógica necesaria en nuestra aplicación.

Pero antes vamos a crear una *rama* o *branch* en *git* para alojar la implementación de esta funcionalidad.

```
$ git branch user-signup
```

```
$ git checkout user-signup
```

```
Switched to branch 'user-signup'
```

Ahora vamos a ejecutar uno de los generadores de *Minitest* para que nos cree el esqueleto de la prueba.

```
$ rails g minitest:feature UserSignup
```

```
create  test/features/user_signup_test.rb
```

Abrimos el archivo que se generó para nosotros y vemos que ya tiene una prueba de ejemplo.

Para ejecutar la prueba hay que primeramente ejecutar el comando `rake db:create` y `RAILS_ENV=test rake db:create`, es necesario ya que nuestra base de datos especificada en `config/database.yml` aún no existe en *Postgresql*. Esto es solamente requerido de hacer una vez al inicio del proyecto.

```
$ rake test
```

```
Run options: --seed 6567
```

```
# Running:
```

```
E
```

```
Finished in 0.005855s, 170.8004 runs/s, 0.0000 assertions/s.
```

```
1) Error:
```

```
UserSignupTest#test_sanity:
```

```
NameError: undefined local variable or method `root_path' for #<UserSignupTest: 0x007f84b52d54d8>
```

```
test/features/user_signup_test.rb:5:in `block in <class:UserSignupTest>'
```

```
1 runs, 0 assertions, 0 failures, 1 errors, 0 skips
```

La prueba falló, pero era de esperarse, vamos a eliminarla del archivo y crear una nueva que siga la especificación descrita anteriormente.

```
require "test_helper"

class UserSignupTest < Capybara::Rails::TestCase

  test "navigate to /signup and see a registration form" do

    visit signup_path

    assert_content page, 'Crear una cuenta'

    assert_selector page, '#new_user'

  end

end
```

Nuestra primera prueba solamente indica que podemos navegar ala *URL* indicada y que vamos ver el mensaje *Crear una cuenta* además de el *HTML* de la página tendrá el selector `#new_user`.

La operación descrita en el párrafo anterior está especificada en código a través de `visit`, `page.has_content?` y `page.has_selector?`. Todos son métodos de `Capybara` que nos ayudan a automatizar la navegación y verificación de funcionalidad en el navegador.

Vamos a ejecutar nuevamente la prueba y veamos que pasa.

```
$ rake test
```

```
Run options: --seed 24547
```

```
# Running:
```

```
E
```

```
Finished in 0.005059s, 197.6847 runs/s, 0.0000 assertions/s.
```

```
1) Error:
```

```
UserSignupTest#test_navigate_to_/signup_and_see_a_registration_form:
```

```
NameError: undefined local variable or method `signup_path' for #<UserSignupTest: 0x007f84b56e3610>
```

```
test/features/user_signup_test.rb:5:in `block in <class:UserSignupTest>'
```

```
1 runs, 0 assertions, 0 failures, 1 errors, 0 skips
```

El error es claro, no está definido el método `signup_path` en nuestra aplicación. Si recordamos en el capítulo 2 hablamos de que las rutas en una aplicación de Rails nos permiten definir puntos para navegar, así mismo mencionábamos que podemos nombrar las rutas para poder hacer referencia a ellas con los métodos `_path` y `_url`.

Entonces el error indica que queremos navegar a una ruta que aún no hemos definido. Vamos a abrir el archivo `config/routes.rb` y agregar la ruta junto con su nombre. El archivo va a contener comentarios de como podemos definir rutas, vamos a eliminarlos todos de modo que nuestro archivo quede como sigue:

```
Rails.application.routes.draw do

  get '/registro' => 'registration#new', as: :signup

end
```

Aquí definimos la ruta `/registro` de acuerdo a nuestra especificación, e indicamos que el controlador `Registration` a través de la acción `new` responderá cuando un usuario navegue a este punto. De igual forma agregamos el nombre `signup` a nuestra ruta.

Vamos inspeccionando nuestra tabla de rutas para ver como quedó definida.

```
$ rake routes
```

| Prefix | Verb | URI Pattern | Controller#Action |
|--------|------|-------------|-------------------|
|--------|------|-------------|-------------------|

|        |     |                     |                  |
|--------|-----|---------------------|------------------|
| signup | GET | /registro(.:format) | registration#new |
|--------|-----|---------------------|------------------|

Todo se ve bien, vamos a ejecutar nuevamente nuestra prueba.

```
$ rake test
```

```
Run options: --seed 45963
```

```
# Running:
```

```
E
```

```
Finished in 0.014336s, 69.7548 runs/s, 0.0000 assertions/s.
```

```
1) Error:
```

```
UserSignupTest#test_navigate_to_/_signup_and_see_a_registration_form:
```

```
ActionController::RoutingError: uninitialized constant RegistrationController
```

```
test/features/user_signup_test.rb:5:in `block in <class:UserSignupTest>'
```

```
1 runs, 0 assertions, 0 failures, 1 errors, 0 skips
```

Ahora el error es diferente, **Minitest** nos indica que la constante **RegistrationController** no existe. Vamos a crear el controlador que nuestra prueba espera. Agregamos el archivo **app/controllers/registration\_controller.rb** con el siguiente contenido.

```
class RegistrationController < ApplicationController  
  
end
```

Ejecutamos una vez más nuestras pruebas.

```
$ rake test
```

```
Run options: --seed 32046
```

```
# Running:
```

```
E
```

```
Finished in 0.028430s, 35.1745 runs/s, 0.0000 assertions/s.
```

1) Error:

```
UserSignupTest#test_navigate_to_/signup_and_see_a_registration_form:
```

```
AbstractController::ActionNotFound: The action 'new' could not be found for RegistrationController
```

```
test/features/user_signup_test.rb:5:in `block in <class:UserSignupTest>'
```

Ahora el mensaje es diferente, la acción `new` no está definida en el controlador. Es momento de agregar un nuevo tipo de prueba a nuestra aplicación, en este caso va a ser una prueba específica a el controlador `RegistrationController` y su acción `new`.

Vamos a crear el archivo `test/controllers/registration_controller_test.rb` y agregamos el siguiente contenido.

```
require 'test_helper'

class RegistrationControllerTest < ActionController::TestCase

  test 'GET /new' do
    get :new

    assert_response :success
  end
end
```

En esta prueba de controlador solamente queremos asegurarnos que de responde a la acción `new` con éxito. Ejecutamos nuevamente nuestras pruebas y evaluamos el resultado.

```
$ rake test
```

```
Run options: --seed 24232
```

```
# Running:
```

```
EE
```

```
Finished in 0.025207s, 79.3445 runs/s, 0.0000 assertions/s.
```

```
1) Error:
```

```
UserSignupTest#test_navigate_to_/signup_and_see_a_registration_form:
```

```
AbstractController::ActionNotFound: The action 'new' could not be found for RegistrationController
```

```
test/features/user_signup_test.rb:5:in `block in <class:UserSignupTest>'
```

2) Error:

```
RegistrationControllerTest#test_GET_/new:
```

```
AbstractController::ActionNotFound: The action 'new' could not be found for RegistrationController
```

```
test/controllers/registration_controller_test.rb:5:in `block in <class:RegistrationControllerTest>'
```

```
2 runs, 0 assertions, 0 failures, 2 errors, 0 skips
```

Ahora en lugar de una, tenemos 2 pruebas que fallan. Vamos a enfocarnos en la prueba del controlador `RegistrationControllerTest#test_GET_/new`. Como vemos, el problema es que la acción `new` aún no está definida. Vamos a agregar la acción al controlador.

```
# app/controllers/registration_controller.rb

class RegistrationController < ApplicationController

  def new

  end

end
```

Ejecutamos nuestras pruebas nuevamente.

```
$ rake test
```

```
Run options: --seed 35018
```

```
# Running:
```

```
EE
```

```
Finished in 0.034473s, 58.0157 runs/s, 0.0000 assertions/s.
```

```
1) Error:
```

```
RegistrationControllerTest#test_GET_/new:
```

```
ActionView::MissingTemplate: Missing template registration/new, application/new  
with {:locale=>[:en], :formats=>[:html], :variants=>[], :han-  
dlers=>[:erb, :builder, :raw, :ruby, :coffee]}. Searched in:
```

```
* "/Users/mariochavez/Development/one_px/app/views"
```

```
test/controllers/registration_controller_test.rb:5:in `block in <class:RegistrationControllerTest>'
```

## 2) Error:

```
UserSignupTest#test_navigate_to_/signup_and_see_a_registration_form:
```

```
ActionView::MissingTemplate: Missing template registration/new, application/new  
with {:locale=>[:en], :formats=>[:html], :variants=>[], :han-  
dlers=>[:erb, :builder, :raw, :ruby, :coffee]}. Searched in:
```

```
* "/Users/mariochavez/Development/one_px/app/views"
```

```
test/features/user_signup_test.rb:5:in `block in <class:UserSignupTest>'
```

Las pruebas siguen fallando, pero vemos que el problema ahora es que el controlador no encuentra el *template new*.

Vamos a crear el archivo `app/views/registration/new.html.erb` y lo dejamos en blanco.

El directorio `registration` no existe, va a ser necesario crear uno.

Ejecutamos nuestras pruebas y vemos que ahora tenemos un resultado diferente.

```
$ rake test
```

```
Run options: --seed 708
```

```
# Running:
```

```
F.
```

```
Finished in 0.158450s, 12.6223 runs/s, 12.6223 assertions/s.
```

```
1) Failure:
```

```
UserSignupTest#test_navigate_to_/signup_and_see_a_registration_form [/Users/mariochavez/Development/one_px/test/features/user_signup_test.rb:7]:
```

```
Expected to include "Crear una cuenta".
```

```
2 runs, 2 assertions, 1 failures, 0 errors, 0 skips
```

Solamente una de las pruebas falla, la prueba de integración, el error es que no encuentra el texto *Crear una cuenta*. Vamos a modificar el *template new.html.erb* que creamos vacío anteriormente y agregamos el texto con un **h3**.

```
# app/views/registration.html.erb
```

```
<h3>Crear una cuenta</h3>
```

Ejecutamos las pruebas una vez más.

```
$ rake test
```

```
Run options: --seed 9070
```

```
# Running:
```

```
F.
```

```
Finished in 0.133935s, 14.9326 runs/s, 22.3989 assertions/s.
```

1) Failure:

```
UserSignupTest#test_navigate_to_/signup_and_see_a_registration_form [/Users/mariochavez/Development/one_px/test/features/user_signup_test.rb:8]:
```

```
expected to find css "#new_user" but there were no matches
```

```
2 runs, 3 assertions, 1 failures, 0 errors, 0 skips
```

El resultado es diferente, ahora no encuentra el selector `#new_user`. Vamos a modificar el *template* de nuevo y agregamos, de momento, un `div`.

```
# app/views/registration.html.erb
```

```
<h3>Crear una cuenta</h3>
```

```
<div id='new_user'></div>
```

Una vez más ejecutamos nuestras pruebas y comprobamos el resultado.

```
$ rake test
```

```
Run options: --seed 64851
```

```
# Running:
```

```
..
```

```
Finished in 0.150925s, 13.2516 runs/s, 13.2516 assertions/s.
```

```
2 runs, 2 assertions, 0 failures, 0 errors, 0 skips
```

Finalmente no tenemos errores, pero esto solamente indica que ocupamos agregar algunas pruebas más, ya que la funcionalidad que esperamos aún no está completa.

En retrospectiva, hasta este momento hemos estado escribiendo pruebas, donde describimos el resultado que esperamos de nuestro código y las vamos ejecutando conforme vamos agregando código que solucione el error que las pruebas reportan. Cómo se mencionó en el capítulo 3, ésta técnica se conoce como [Test Driven Development](#) o [TDD](#).

Antes de continuar, vamos a realizar una pause y crear un nuevo [commit](#) de nuestro código.

```
$ git add .
```

```
$ git commit -m "Sanity test for UserSignup feature"
```

Vamos agregar una serie de pruebas al archivo `test/features/user_signup_test.rb` donde completemos toda la funcionalidad espera de un usuario al registrarse.

```
# test/features/user_signup_test.rb

require 'test_helper'

class UserSignupTest < Capybara::Rails::TestCase

  test 'navigate to /signup and see a registration form' do
```

```
visit signup_path

assert_content page, 'Crear una cuenta'

assert_selector page, '#new_user'

end

test 'fill in form at /signup and see welcome message' do

  visit signup_path

  within('#new_user') do

    fill_in 'Correo electrónico', with: 'newuser@mail.com'

    fill_in 'Contraseña', with: 'password'

    fill_in 'Confirmar contraseña', with: 'password'
```

```
click_button 'Crear cuenta'

end

assert_content 'Bienvenido newuser@mail.com'

end

test 'dont fill in form at /signup, click button and see form errors' do

  visit signup_path

  within('#new_user') do

    click_button 'Crear cuenta'

  end

  assert_content 'Por favor corrija los siguientes errores'
```

```
assert_equal all('.error-field').count, 2

refute_content 'Bienvenido'

end

end
```

Las 2 nuevas pruebas de integración se encargan de verificar que si llenamos los datos del formulario de registro correctamente, el usuario va a poder registrarse y que si no lo hacemos así, entonces obtendremos un mensaje que nos indique que tenemos que completar la información.

Al ejecutar nuestras pruebas ambas fallan indicando que no encuentran el formulario.

```
$ rake test
```

```
Run options: --seed 32785
```

```
# Running:
```

```
.EE.
```

```
Finished in 0.174569s, 22.9136 runs/s, 11.4568 assertions/s.
```

1) Error:

```
UserSignupTest#test_dont_fill_in_form_at_signup,_click_button_and_see_form_errors:
```

```
Capybara::ElementNotFound: Unable to find button "Crear cuenta"
```

```
    test/features/user_signup_test.rb:29:in `block (2 levels) in <class:UserSignupTest>'
```

```
    test/features/user_signup_test.rb:28:in `block in <class:UserSignupTest>'
```

2) Error:

```
UserSignupTest#test_fill_in_form_at_signup_and_see_welcome_message:
```

```
Capybara::ElementNotFound: Unable to find field "Correo electrónico"
```

```
    test/features/user_signup_test.rb:15:in `block (2 levels) in <class:UserSignupTest>'
```

```
    test/features/user_signup_test.rb:14:in `block in <class:UserSignupTest>'
```

```
4 runs, 2 assertions, 0 failures, 2 errors, 0 skips
```

Por deducción obvia, ha llegado el momento de implementar el formulario dentro del `template app/views/registration/new.html`, pero para poder hacerlo tenemos la dependencia de necesitar un modelo que represente a un usuario en la base de datos.

### *El modelo User y contraseñas seguras*

El modelo `User` de acuerdo a la especificación general de ésta característica debe de:

- Contener un correo electrónico y ser único en la base de datos.
- Tener una contraseña, la cuál se debe de confirmar y debe de ser mínimo de 8 caracteres.

Con esta información hay que escribir pruebas para el modelo `User`, éstas las vamos a crear en el archivo `test/models/user_test.rb`.

```
# test/models/user_test.rb

require 'test_helper'

class UserTest < ActiveSupport::TestCase

  test 'valid user with valid attributes' do

    user = User.new.tap do |u|
```

```
    u.email = 'test@mail.com'

    u.password = 'pa$$word'

    u.password_confirmation = 'pa$$word'

end

assert_equal user.valid?, true

end

test 'invalid user with invalid attributes' do

  user = User.new

  refute_equal user.valid?, true

  refute_empty user.errors[:email]
```

```
refute_empty user.errors[:password]

end

test 'invalid user with duplicated email' do

  user = User.new.tap do |u|

    u.email = 'user@mail.com'

    u.password = 'pa$$word'

    u.password_confirmation = 'pa$$word'

  end

  refute_equal user.valid?, true

  refute_empty user.errors[:email]

  assert_empty user.errors[:password]
```

```
end

test 'invalid user with short password' do

  user = User.new.tap do |u|
    u.email = 'test@mail.com'
    u.password = 'pa$$'
    u.password_confirmation = 'pa$$'

  end

  refute_equal user.valid?, true

  assert_empty user.errors[:email]

  refute_empty user.errors[:password]

end
```

```
end
```

Hemos agregado 4 pruebas para el modelo `User` con la finalidad de asegurarnos que el modelo puede ser válido en algún momento, que puede ser inválido si no le asignamos valor a sus atributos y finalmente es inválido si no se cumplen las condiciones que mencionamos previamente.

Si ejecutamos nuestras pruebas, podemos ver que las relacionadas a `UserTest` fallan ya que Rails no encuentra la clase `User`.

```
$ rake test
```

```
Run options: --seed 29162
```

```
# Running:
```

```
.EE.EEEE
```

```
Finished in 0.167228s, 47.8389 runs/s, 11.9597 assertions/s.
```

```
1) Error:
```

```
UserSignupTest#test_fill_in_form_at_signup_and_see_welcome_message:
```

```
Capybara::ElementNotFound: Unable to find field "Correo electrónico"
```

```
  test/features/user_signup_test.rb:15:in `block (2 levels) in <class:User-
SignupTest>'
```

```
test/features/user_signup_test.rb:14:in `block in <class:UserSignupTest>'
```

2) Error:

```
UserSignupTest#test_dont_fill_in_form_at_signup,_click_button_and_see_form_er-
rors:
```

```
Capybara::ElementNotFound: Unable to find button "Crear cuenta"
```

```
  test/features/user_signup_test.rb:29:in `block (2 levels) in <class:User-
SignupTest>'
```

```
test/features/user_signup_test.rb:28:in `block in <class:UserSignupTest>'
```

3) Error:

```
UserTest#test_invalid_user_with_duplicated_email:
```

```
NameError: uninitialized constant UserTest::User
```

```
test/models/user_test.rb:24:in `block in <class:UserTest>'
```

4) Error:

```
UserTest#test_invalid_user_with_invalid_attributes:
```

```
NameError: uninitialized constant UserTest::User
```

```
test/models/user_test.rb:15:in `block in <class:UserTest>'
```

5) Error:

```
UserTest#test_valid_user_with_valid_attributes:
```

```
NameError: uninitialized constant UserTest::User
```

```
test/models/user_test.rb:5:in `block in <class:UserTest>'
```

## 6) Error:

```
UserTest#test_invalid_user_with_short_password:
```

```
NameError: uninitialized constant UserTest::User
```

```
test/models/user_test.rb:37:in `block in <class:UserTest>'
```

```
8 runs, 2 assertions, 0 failures, 6 errors, 0 skips
```

Vamos solucionar este problema. Para el caso de crear un modelo vamos a hacer uso de los generadores de Rails, ya que aparte de crear el modelo, es necesario crear una migración.

Si recordamos lo mencionado en el capítulo 2, una migración contiene instrucciones para generar parte del esquema de la base de datos. Las instrucciones de la migración son convertidas a **SQL** válido para el motor de base de datos que nuestra aplicación está utilizando, pero no solamente eso, Rails nos ayuda a tener organizados los cambios que vamos realizando a la estructura de la base de datos, registrando que migraciones ya hemos aplicado y cuales aún están pendientes.

Pero antes proceder a la creación de la migración y el modelo, es necesario saber que guardar contraseñas en la base de datos es algo que simplemente **está mal**. Cualquier persona que consiga acceso a la tabla de usuarios podría ver las contraseñas ahí guardadas. Esto es un problema **muy grave de seguridad**.

La forma correcta de guardar contraseñas es aplicando algún tipo de cifrado criptográfico que impida la obtención de contraseñas de una manera fácil. El problema de es que implementar

este tipo de mecanismos puede ser complicado y abierto a fallas si no somos expertos en la materia.

Por tal razón y entendiendo la necesidad de que las contraseñas puedan estar seguras en una aplicación de Rails, el marco de trabajo ofrece una funcionalidad llamada `has_secure_password`<sup>10</sup>.

La forma en como `has_secure_password` funciona, es que requiere y hace uso de la gema `bcrypt`, ya que ésta contiene el algoritmo necesario para cifrar criptográficamente las contraseñas.

`has_secure_password` requiere de un atributo en nuestro modelo que se debe de llamar `password_digest` de tipo `String`, ya que este atributo es el que almacena la contraseña cifrada. Los atributos `password` y `password_confirmation` que utilizamos en las pruebas, deben de ser atributos virtuales, es decir no están respaldados por una columna en la tabla de usuarios.

Para generar el modelo junto con su migración, en la terminar vamos a ejecutar el siguiente comando:

```
$ rails g model user email password_digest

invoke  active_record

create    db/migrate/20151105013845_create_users.rb

create    app/models/user.rb

invoke  minitest

conflict  test/models/user_test.rb

Overwrite /Users/mariochavez/Development/one_px/test/models/user_test.rb?
(enter "h" for help) [Ynaqdh] n
```

---

<sup>10</sup> <http://api.rubyonrails.org/classes/ActiveModel/SecurePassword/ClassMethods.html>

```
skip      test/models/user_test.rb
```

```
create    test/fixtures/users.yml
```

Al inspeccionar el resultado del comando vemos que ha generado una serie de archivos y que trató de generar el archivo `test/models/user_test.rb` que nosotros ya creamos, es importante decirle que no deseamos sobreescribirlo cuando el generador nos pregunte.

Vamos a buscar primeramente nuestro archivo de migración, el cual vamos a encontrar en `db/migrate`. El nombre de el archivo no va a ser el mismo que yo generé, esto debido a la primera parte del archivo es un `Timestamp` es decir contiene la fecha y hora en la que el archivo fue generado y posteriormente el nombre de la migración.

Al abrir el archivo vemos que dentro del método `change` existe código para crear la tabla `users` y agregar 2 columnas de tipo `String`. Adicionalmente existe una definición para `timestamps`, ésta definición generará las columnas `created_at` y `updated_at` las cuales Rails controla y registra ahí la fecha y hora en que un modelo fue creado y actualizado por última vez.

Vamos a agregar algunos cambios. Para las columnas `email` y `password_digest` vamos a agregar la restricción a nivel de base de datos para que no acepten valores nulos, esto lo logramos con `null: false`.

Para el caso de `:email` también queremos agregar un índice único en la base de datos, de forma que no permita correos electrónicos repetidos. Esto lo logramos agregando un `add_index`.

Nuestro archivo de migraciones debe quedar como se muestra a continuación.

```
# db/migrate/20151105013845_create_users.rb
```

```
class CreateUsers < ActiveRecord::Migration
```

```
def change
```

```
  create_table :users do |t|
```

```
    t.string :email, null: false
```

```
t.string :password_digest, null: false

t.timestamps null: false

end

add_index :users, :email, unique: true

end

end
```

Ya con los cambios realizados, ejecutamos el comando para aplicar el cambio de esquema a la base de datos.

```
$ rake db:migrate

== 20151105013845 CreateUsers: migrating =====

-- create_table(:users)

-> 0.0076s

-- add_index(:users, :email, {:unique=>true})
```

```
-> 0.0031s
```

```
-- 20151105013845 CreateUsers: migrated (0.0108s) =====
```

Hay que abrir el archivo `Gemfile` y agregar la línea `gem bcrypt` antes de la línea `gem 'coffee-rails', '~> 4.1.0'`. Después de guardar el archivo ejecutamos el comando `bundle` para que `bcrypt` esté disponible en nuestra aplicación.

Ahora estamos listos para modificar el archivo `app/models/user.rb` que se creo con el generador de Rails anteriormente. Vamos a agregarle la definición de `has_secure_password` y posteriormente ejecutaremos nuestras pruebas para ver que ha cambiado.

```
# app/models/user.rb
```

```
class User < ActiveRecord::Base
```

```
    has_secure_password
```

```
end
```

```
$ rake test
```

Al ejecutar la pruebas obtenemos un mensaje de error un tanto raro `ActiveRecord::RecordNotUnique: PG::UniqueViolation: ERROR: duplicate key value violates unique constraint "index_users_on_email".`

El error nos indica que estamos violando la restricción de la base de datos para ingresar correos electrónicos duplicados, al parecer el error no tiene sentido ya, pero tiene que ver con uno de los archivos que creo el generador del modelo para nosotros: `test/fixtures/users.yml`.

Vamos a abrir el archivo y eliminar el contenido, más adelante en éste capítulo vamos a regresar al archivo y comentaremos un poco más de detalles sobre él.

Ejecutamos nuevamente nuestras pruebas, pero en ésta ocasión vamos restringiendo para ver sólo las pruebas de modelos con el siguiente comando:

```
$ rake test:models
```

```
Run options: --seed 24481
```

```
# Running:
```

```
FFF.
```

```
Finished in 0.031112s, 128.5692 runs/s, 192.8538 assertions/s.
```

```
1) Failure:
```

```
UserTest#test_invalid_user_with_short_password [/Users/mariochavez/Development/
one_px/test/models/user_test.rb:40]:
```

```
Expected true to not be equal to true.
```

2) Failure:

```
UserTest#test_invalid_user_with_invalid_attributes [/Users/mariochavez/Development/one_px/test/models/user_test.rb:16]:
```

```
Expected [] to not be empty.
```

3) Failure:

```
UserTest#test_invalid_user_with_duplicated_email [/Users/mariochavez/Development/one_px/test/models/user_test.rb:27]:
```

```
Expected true to not be equal to true.
```

```
4 runs, 6 assertions, 3 failures, 0 errors, 0 skips
```

Tenemos 3 fallas. Vamos corregir primeramente la prueba `UserTest#test_invalid_user_with_invalid_attributes` que en nuestro archivo `test/models/users_spec.rb` es la prueba de la línea 16.

```
# test/models/user_spec.rb
```

```
...
```

```
test 'invalid user with invalid attributes' do

    user = User.new

    refute_equal user.valid?, true

    refute_empty user.errors[:email]

    refute_empty user.errors[:password]

end

...
```

Primeramente nuestra prueba trata de crear un registro para `User` sin ningún parámetro.

`refute_equal user.valid?, true` comprueba que al realizar la operación el usuario no será válido. El método `valid?` nos regresa un verdadero o falso dependiendo del estado del objeto `user`.

En `ActiveRecord` los objetos tienen la propiedad `errors`, el cuál es un `Hash` especializado y en caso de que alguna validación falle, aquí es donde se registran los errores. Al pasar el `Symbol` de un atributo, podemos investigar los errores específicos para él, los errores son regresados en un `Array`.

Es por eso que `refute_empty user.errors[:email]` y `refute_empty user.errors[:password]` comprueban que el `Array` de errores para `:email` y `:password` no será un `Array` vacío.

En el caso de ésta prueba en particular, el error proviene de `refute_empty user.e-`  
`rrors[:email]` ya que `:email` no reporta errores.

Vamos a corregirla agregando una validación al model `User` en donde indiquemos que `:email` es requerido.

```
# app/models/user.rb

class User < ActiveRecord::Base

  has_secure_password

  validates :email, presence: true

end
```

Ejecutamos nuestras pruebas y vemos el resultado.

```
$ rake test:models

Run options: --seed 50865
```

```
# Running:
```

```
F..F
```

```
Finished in 0.036702s, 108.9846 runs/s, 217.9691 assertions/s.
```

1) Failure:

```
UserTest#test_invalid_user_with_duplicated_email [/Users/mariochavez/Development/  
one_px/test/models/user_test.rb:24]:
```

```
Expected true to not be equal to true.
```

2) Failure:

```
UserTest#test_invalid_user_with_short_password [/Users/mariochavez/Development/  
one_px/test/models/user_test.rb:37]:
```

```
Expected true to not be equal to true.
```

```
4 runs, 8 assertions, 2 failures, 0 errors, 0 skips
```

Ahora tenemos 2 pruebas fallando para el model `User`. Vamos a revisar la prueba `UserTest#test_invalid_user_with_duplicated_email`.

```
#test/models/user_spec.rb
```

...

```
test 'invalid user with duplicated email' do

    user = User.new.tap do |u|
        u.email = 'user@mail.com'
        u.password = 'pa$$word'
        u.password_confirmation = 'pa$$word'

    end

    refute_equal user.valid?, true

    refute_empty user.errors[:email]

    assert_empty user.errors[:password]

end
```

...

La prueba espera que al crear un registro de tipo `user` con un email de otro registro que ya exista en la base de datos obtengamos un error. `refute_empty user.errors[:email]` es el paso que

verifica que esperamos errores en el atributo `:email`, mientras que `assert_empty user.errors[:password]` indica que no esperamos errores en el atributo `:password`.

Vamos primeramente a agregar una validación en el modelo `User` para indicar que el correo electrónico debe de ser único.

```
#app/models/user.rb

class User < ActiveRecord::Base

  has_secure_password

  validates :email, presence: true, uniqueness: true

end
```

Para que nuestra prueba funcione, necesitamos que exista un registro en la base de datos de `test` antes de ejecutar nuestra prueba y que el registro contenga el correo electrónico `user@mail.com`.

Aquí es donde vamos a necesitar del archivo `test/fixtures/users.yml`. Este archivo es un archivo `fixture`, es decir podemos definir registros que ocupamos en nuestras pruebas para que se carguen de forma automática a la base de datos.

Vamos a agregar un registro que nos sirva para nuestro propósito.

```
#test/fixtures/users.yml

user:
  email: 'user@mail.com'
```

```
password_digest: 'pa$$word'
```

Ejecutamos nuevamente nuestras pruebas y observamos el resultado.

```
$ rake test:models
```

```
Run options: --seed 45120
```

```
# Running:
```

```
.F..
```

```
Finished in 0.071670s, 55.8117 runs/s, 167.4351 assertions/s.
```

1) Failure:

```
UserTest#test_invalid_user_with_short_password [/Users/mariochavez/Development/
one_px/test/models/user_test.rb:37]:
```

```
Expected true to not be equal to true.
```

Ya sólo nos queda una prueba más por corregir: `UserTest#test_invalid_user_with_short_password`. Volvamos al model de `User` y agreguemos una validación para cuando la contraseña es menor a 8 caracteres.

```
#app/models/user.rb
```

```
class User < ActiveRecord::Base

  has_secure_password

  validates :email, presence: true, uniqueness: true

  validates :password, length: { minimum: 8 }

end
```

Ejecutamos nuevamente las pruebas y comprobamos que nuestro modelo ya pasa todas la pruebas que definimos.

```
$ rake test:models
```

```
Run options: --seed 43505
```

```
# Running:
```

....

Finished in **0.078010s**, 51.2754 runs/s, 205.1015 assertions/s.

**4 runs, 16 assertions, 0 failures, 0 errors, 0 skips**

Pero si ejecutamos todas las pruebas, no únicamente las de modelos, vemos que aún nos queda trabajo por hacer.

```
$ rake test
```

Run options: --seed 14816

# Running:

.E.E....

Finished in **0.231639s**, 34.5365 runs/s, 77.7072 assertions/s.

1) Error:

```
UserSignupTest#test_dont_fill_in_form_at_signup,_click_button_and_see_form_errors:
```

```
Capybara::ElementNotFound: Unable to find button "Crear cuenta"
```

```
    test/features/user_signup_test.rb:29:in `block (2 levels) in <class:UserSignupTest>'
```

```
    test/features/user_signup_test.rb:28:in `block in <class:UserSignupTest>'
```

2) Error:

```
UserSignupTest#test_fill_in_form_at_signup_and_see_welcome_message:
```

```
Capybara::ElementNotFound: Unable to find field "Correo electrónico"
```

```
    test/features/user_signup_test.rb:15:in `block (2 levels) in <class:UserSignupTest>'
```

```
    test/features/user_signup_test.rb:14:in `block in <class:UserSignupTest>'
```

```
8 runs, 18 assertions, 0 failures, 2 errors, 0 skips
```

Vamos a trabajar sobre la prueba `UserSignupTest#test_dont_fill_in_form_at_signup,_click_button_and_see_form_errors`. El motivo de su falla es `Capybara::ElementNotFound: Unable to find button "Crear cuenta"`, no encuentra el botón y esto se debe a que no hemos creado el formulario.

El formulario debe de existir en el `template app/views/registration.new.html`. En este caso vamos a hacer uso del `Helper form_for` de Rails para que nos ayude a crear el formulario y agregaremos el botón.

```
# app/views/registration.new.html
```

```
<h3>Crear una cuenta</h3>
```

```
<%= form_for @user do |form| %>
```

```
<%= form.submit %>
```

```
<% end %>
```

Ejecutamos pruebas y veamos qué sucede.

```
$ rake test
```

```
Run options: --seed 3515
```

```
# Running:
```

EEE....E

Finished in 0.162412s, 49.2573 runs/s, 98.5146 assertions/s.

1) Error:

UserSignupTest#test\_fill\_in\_form\_at\_/signup\_and\_see\_welcome\_message:

ActionView::Template::Error: First argument in form cannot contain nil or be empty

app/views/registration/new.html.erb:2:in `\_\_app\_views\_registration\_new\_html\_erb\_\_\_\_1820466596605678274\_70283571008660'

test/features/user\_signup\_test.rb:12:in `block in <class:UserSignupTest>'

2) Error:

```
UserSignupTest#test_dont_fill_in_form_at_signup,_click_button_and_see_form_errors:
```

```
ActionView::Template::Error: First argument in form cannot contain nil or be empty
```

```
    app/views/registration/new.html.erb:2:in `__app_views_registration_new_html_erb____1820466596605678274_70283571008660'
```

```
    test/features/user_signup_test.rb:26:in `block in <class:UserSignupTest>'
```

3) Error:

```
UserSignupTest#test_navigate_to_signup_and_see_a_registration_form:
```

```
ActionView::Template::Error: First argument in form cannot contain nil or be empty
```

```
    app/views/registration/new.html.erb:2:in `__app_views_registration_new_html_erb____1820466596605678274_70283571008660'
```

```
    test/features/user_signup_test.rb:5:in `block in <class:UserSignupTest>'
```

4) Error:

```
RegistrationControllerTest#test_GET_/new:
```

```
ActionView::Template::Error: First argument in form cannot contain nil or be empty
```

```
app/views/registration/new.html.erb:2:in `__app_views_registration_new_html_erb____1820466596605678274_70283571008660'
```

```
test/controllers/registration_controller_test.rb:5:in `block in <class:RegistrationControllerTest>'
```

```
8 runs, 16 assertions, 0 failures, 4 errors, 0 skips
```

Todas las pruebas fallan con el mensaje `ActionView::Template::Error: First argument in form cannot contain nil or be empty`. Esto se debe a que cuando agregamos el `Helper` de `form_for` indicamos que utilizará la variable `@user` pero en ningún lugar inicializamos la variable.

Para solucionar este problema, hay que abrir el archivo del `RegistrationController` y en la acción `new` inicializamos la variable.

```
# app/controllers/registration_controller.rb

class RegistrationController < ApplicationController

  def new

    @user = User.new
```

```
end
```

```
end
```

Ejecutemos nuevamente las pruebas.

```
$ rake test
```

```
Run options: --seed 36832
```

```
# Running:
```

```
EEEE....
```

```
Finished in 0.140074s, 57.1126 runs/s, 114.2252 assertions/s.
```

1) Error:

```
RegistrationControllerTest#test_GET_/new:
```

```
ActionView::Template::Error: undefined method `users_path' for #<#<Class:  
0x007fd8568dfca8>:0x007fd8568de998>  
  
app/views/registration/new.html.erb:2:in `__app_views_registration_new_htm-  
l_erb____1820466596605678274_70283537012920'  
  
test/controllers/registration_controller_test.rb:5:in `block in <class:Regis-  
trationControllerTest>'
```

2) Error:

```
UserSignupTest#test_dont_fill_in_form_at_signup,_click_button_and_see_form_er-  
rors:
```

```
ActionView::Template::Error: undefined method `users_path' for #<#<Class:  
0x007fd8568dfca8>:0x007fd855af1268>
```

```
app/views/registration/new.html.erb:2:in `__app_views_registration_new_htm-  
l_erb____1820466596605678274_70283537012920'
```

```
test/features/user_signup_test.rb:26:in `block in <class:UserSignupTest>'
```

3) Error:

```
UserSignupTest#test_fill_in_form_at_signup_and_see_welcome_message:
```

```
ActionView::Template::Error: undefined method `users_path' for #<#<Class:  
0x007fd8568dfca8>:0x007fd8522e3150>
```

```
    app/views/registration/new.html.erb:2:in `__app_views_registration_new_htm-  
l_erb____1820466596605678274_70283537012920'
```

```
test/features/user_signup_test.rb:12:in `block in <class:UserSignupTest>'
```

4) Error:

```
UserSignupTest#test_navigate_to_signup_and_see_a_registration_form:
```

```
ActionView::Template::Error: undefined method `users_path' for #<#<Class:  
0x007fd8568dfca8>:0x007fd855a0b0d8>
```

```
    app/views/registration/new.html.erb:2:in `__app_views_registration_new_htm-  
l_erb____1820466596605678274_70283537012920'
```

```
test/features/user_signup_test.rb:5:in `block in <class:UserSignupTest>'
```

```
8 runs, 16 assertions, 0 failures, 4 errors, 0 skips
```

No avanzamos mucho, todas las pruebas fallan, pero ahora es con otro error `ActionView::Template::Error: undefined method `users_path' for form_for` asume que siendo una formulario para el modelo `User`, queremos que el formulario se publique hacia un controlador de usuarios, pero en nuestro caso deseamos que se publique al controlador `RegistrationController` en la acción de `create`.

Vamos a indicarle a `form_for` éste cambio y vamos a crear una ruta para la acción.

```
# config/routes.rb
```

```
Rails.application.routes.draw do
```

```
  get '/registro' => 'registration#new', as: :signup
```

```
  post '/registro' => 'registration#create', as: :registration
```

```
end
```

```
# app/views/registration/new.html.erb
```

```
<h3>Crear una cuenta</h3>
```

```
<%= form_for @user, url: registration_path do |form| %>
```

```
  <%= form.submit %>
```

```
<% end %>
```

Ejecutamos las pruebas y vemos que todavía fallan con el problema de que no encuentra el botón **Crear cuenta**. El formulario y el botón existen, el problema es que el botón no tiene el texto correcto.

Para agregar el texto correcto al botón vamos a utilizar el *API* de internacionalización de Ruby on Rails. Primeramente vamos a abrir el archivo **config/application.rb** y dentro de él buscamos la línea `# config.i18n.default_locale = :de`, le quitamos el `#` y modificamos el valor `:de` a `:es`.

En el siguiente paso es necesario descarga de Internet el archivo **es.yml**<sup>11</sup> y lo ponemos en el directorio **config/locales**. En ese mismo directorio vamos a crear el archivo **config/locales/es.views.yml** el cuál vamos a utilizar para poner las traducciones en español de los texto que utiliza nuestra aplicación en los **templates**.

```
# config/locales/es.views.yml

es:

registration:

new:

    title: 'Crear una cuenta'

helpers:

submit:

user:
```

---

<sup>11</sup> <https://raw.githubusercontent.com/svenfuchs/rails-i18n/master/rails/locale/es.yml>

```
create: 'Crear cuenta'
```

Observando el archivo `config/locales/es.views.yml` nos percatamos de que tiene una estructura anidad de llaves y valores.

La parte que nos interesa de momento, es la sección de `helpers:`. Vemos que tenemos la llave `submit:` que Rails busca para todos los botones `submit` de los formularios, pero luego agregamos la llave del model y la acción de `create:` con el texto que deseamos que el botón muestre. Rails reconoce ésta estructura y colocará el mensaje correcto.

Antes de ejecutar nuestras pruebas, si observamos el archivo `app/views/registration/new.html` vemos que al inicio tiene una etiqueta `h3` con un mensaje, vamos a modificar el archivo para que utilice el archivo de traducciones que acabamos de agregar. Es por eso que al principio existe la llave `registration.new.title` con el texto `Crear una cuenta`.

Para indicarle a Rails que utilice el texto del archivo es muy sencillo, solamente tenemos que utilizar el `Helper t` con la última sección de la llave de traducción y Rails hará el resto.

```
# app/view/registration/new.html.erb

<h3><%= t('.title') %></h3>

<%= form_for @user, url: registration_path do |form| %>

  <%= form.submit %>

<% end %>
```

Vamos a ejecutar nuestras pruebas una vez más.

```
$ rake test
```

```
Run options: --seed 19864
```

```
# Running:
```

```
.....EE.
```

```
Finished in 0.263649s, 30.3433 runs/s, 68.2725 assertions/s.
```

```
1) Error:
```

```
UserSignupTest#test_dont_fill_in_form_at_signup,_click_button_and_see_form_errors:
```

```
AbstractController::ActionNotFound: The action 'create' could not be found for  
RegistrationController
```

```
    test/features/user_signup_test.rb:29:in `block (2 levels) in <class:User-  
SignupTest>'
```

```
    test/features/user_signup_test.rb:28:in `block in <class:UserSignupTest>'
```

## 2) Error:

```
UserSignupTest#test_fill_in_form_at_signup_and_see_welcome_message:
```

```
Capybara::ElementNotFound: Unable to find field "Correo electrónico"
```

```
test/features/user_signup_test.rb:15:in `block (2 levels) in <class:UserSignupTest>'
```

```
test/features/user_signup_test.rb:14:in `block in <class:UserSignupTest>'
```

```
8 runs, 18 assertions, 0 failures, 2 errors, 0 skips
```

Siguen fallando 2, pero el mensaje de la prueba en la que estamos trabajando ahora cambió a `AbstractController::ActionNotFound: The action 'create' could not be found for RegistrationController`. El mensaje nos indica que se presionó el botón del formulario pero que el controlador `RegistrationController` no tiene aún la acción de `create`.

Hay que recordar que en esta prueba lo que estamos esperando es enviar el formulario vacío y que Rails nos muestre los mensajes de error en el formulario para que el usuario pueda corregirlos.

```
# test/features/user_signup_test.rb
```

```
...
```

```
test 'dont fill in form at /signup, click button and see form errors' do
```

```
  visit signup_path
```

```
within('#new_user') do

  click_button 'Crear cuenta'

end

assert_content 'Por favor corrija los siguientes errores'

assert_equal all('.error-field').count, 2

refute_content 'Bienvenido'

end

...
```

Teniendo esta información presente, vamos a agregar una prueba al controlador [Registration-Controller](#).

```
# test/controllers/registration_controller_test.rb

require 'test_helper'
```

```
class RegistrationControllerTest < ActionController::TestCase

  test 'GET /new' do
    get :new

    assert_response :success

  end

  test 'POST /create without params' do
    post :create

    assert_response :success

  end
end
```

Ejecutamos la prueba, pero sólo para el controladores y vemos el resultado.

```
$ rake test:controllers
```

```
Run options: --seed 10953
```

```
# Running:
```

```
E.
```

```
Finished in 0.199141s, 10.0431 runs/s, 5.0216 assertions/s.
```

```
1) Error:
```

```
RegistrationControllerTest#test_POST_/_create_without_params:
```

```
AbstractController::ActionNotFound: The action 'create' could not be found for  
RegistrationController
```

```
test/controllers/registration_controller_test.rb:11:in `block in <class:Reg-  
istrationControllerTest>'
```

```
2 runs, 1 assertions, 0 failures, 1 errors, 0 skips
```

La prueba que acabamos de agregar falla. Vamos a hacer que pase agregando la acción `create`.

```
# app/controllers/registration_controller.rb

class RegistrationController < ApplicationController

  def new

    @user = User.new

  end

  def create

    @user = User.new

    render :new

  end

end
```

Ejecutamos nuestras pruebas de controladores y ya pasan todas.

```
$ rake test:controllers
```

```
Run options: --seed 32010
```

```
# Running:
```

```
..
```

```
Finished in 0.185931s, 10.7567 runs/s, 10.7567 assertions/s.
```

```
2 runs, 2 assertions, 0 failures, 0 errors, 0 skips
```

Ahora ejecutamos todas las pruebas y vemos el resultado.

```
$ rake test
```

```
Run options: --seed 37870
```

```
# Running:
```

..FE.....

Finished in 0.351153s, 25.6298 runs/s, 56.9552 assertions/s.

1) Failure:

UserSignupTest#test\_dont\_fill\_in\_form\_at\_signup,\_click\_button\_and\_see\_form\_errors [/Users/mariochavez/Development/one\_px/test/features/user\_signup\_test.rb:32]:

Expected to include "Por favor corrija los siguientes errores".

2) Error:

UserSignupTest#test\_fill\_in\_form\_at\_signup\_and\_see\_welcome\_message:

Capybara::ElementNotFound: Unable to find field "Correo electrónico"

test/features/user\_signup\_test.rb:15:in `block (2 levels) in <class:UserSignupTest>'

```
test/features/user_signup_test.rb:14:in `block in <class:UserSignupTest>'
```

```
9 runs, 20 assertions, 1 failures, 0 skips
```

Nuestra prueba `UserSignupTest#test_dont_fill_in_form_at_signup,_click_button_and_`  
`see_form_errors` sigue fallando pero ahora el mensaje es diferente, dice que no encuentra el  
texto `Expected to include "Por favor corrija los siguientes errores".`, vamos por  
buen camino.

El código que agregamos a la acción `RegistrationController#create` no activa las validacio-  
nes del modelo `User`, vamos a corregir esa parte y después vamos agregar lógica en el `template`  
de `new` para que muestre el mensaje error.

```
# app/controllers/registration_controller.rb

class RegistrationController < ApplicationController

  def new

    @user = User.new

  end

  def create

    @user = User.new

  end
end
```

```
  @user.valid?  
  
    render :new  
  
  end  
  
end  
  
# app/views/registration/new.html.erb  
  
<h3><%= t('.title') %></h3>  
  
<%= form_for @user, url: registration_path do |form| %>  
  
  <%-- if @user.errors.any? %>  
  
    <div class='form-error-message'><%= t('form.error.message') %></div>  
  
  <% end %>  
  
<%= form.submit %>
```

```
<% end %>

# config/locales/es.views.yml

es:

registration:

new:

title: 'Crear una cuenta'

helpers:

submit:

user:

create: 'Crear cuenta'

form:
```

```
error:
```

```
message: 'Por favor corrija los siguientes errores'
```

Ejecutamos pruebas nuevamente y evaluamos.

```
$ rake test
```

```
Run options: --seed 38311
```

```
# Running:
```

```
.....EF.
```

```
Finished in 0.222217s, 40.5009 runs/s, 94.5022 assertions/s.
```

1) Error:

```
UserSignupTest#test_fill_in_form_at_/signup_and_see_welcome_message:
```

```
Capybara::ElementNotFound: Unable to find field "Correo electrónico"
```

```
test/features/user_signup_test.rb:15:in `block (2 levels) in <class:UserSignupTest>'
```

```
test/features/user_signup_test.rb:14:in `block in <class:UserSignupTest>'
```

## 2) Failure:

```
UserSignupTest#test_dont_fill_in_form_at_signup,_click_button_and_see_form_errors [/Users/mariochavez/Development/one_px/test/features/user_signup_test.rb:33]:
```

Expected: 0

Actual: 2

```
9 runs, 21 assertions, 1 failures, 1 errors, 0 skips
```

La prueba sigue fallando, pero ahora el mensaje es diferente. El error proviene de la verificación `assert_equal all('.error-field').count, 2` donde se espera que 2 campos de entrada en el formulario tengan la clase de CSS `.error-field`.

Para hacer que la verificación pase, vamos a agregar los campos al formulario.

```
# app/views/registration/new.html.erb
```

```
<h3><%= t('.title') %></h3>

<%= form_for @user, url: registration_path do |form| %>

<%-- if @user.errors.any? %>

<div class='form-error-message'><%= t('form.error.message') %></div>

<% end %>

<div class='field <%= set_error_class(form.object, :email) %>'>

<%= form.label :email %>

<%= form.text_field :email %>

</div>

<div class='field <%= set_error_class(form.object, :password) %>'>

<%= form.label :password %>

<%= form.password_field :password %>

</div>

<div class='field <%= set_error_class(form.object, :password_confirmation) %>'>
```

```

<%= form.label :password_confirmation %>

<%= form.password_field :password_confirmation %>

</div>

<div class='action'>

<%= form.submit %>

</div>

<% end %>

```

Completamos nuestro formulario con los campos para `:email`, `:password` y `:password_confirmation`. Para el formulario utilizamos los **Helpers** de Rails `:text_field`, `:label` y `:password_field` todos generan el **HTML** necesario para hacer que el formulario sea funcional.

Cada pareja de etiqueta y campo de entrada los colocamos dentro de una etiqueta `div`. Vemos que la propiedad de `class` de cada uno de los `divs` contiene código de Ruby `set_error_class(form.object, :password)`. El método `set_error_class` simplemente verifica si el atributo del modelo `User` contiene errores, si es así entonces agrega la clase de `CSS .error-field`. El método está implementado en el módulo `ApplicationHelper`.

```

# app/helpers/application_helper.rb

module ApplicationHelper

  def set_error_class(model, field)

    return 'error-field' if model.errors[field].any?

```

```
    end
```

```
end
```

Este módulo, está disponible para todas las **templates** de Rails y generalmente lo utilizamos para agregar métodos con lógica un tanto compleja para evitar *infectar* las **templates** con mucha lógica.

Nuevamente ejecutamos las pruebas y evaluamos resultados.

```
$ rake test
```

```
Run options: --seed 9970
```

```
# Running:
```

```
.....E.
```

```
Finished in 0.249314s, 36.0990 runs/s, 88.2420 assertions/s.
```

### 1) Error:

```
UserSignupTest#test_fill_in_form_at_signup_and_see_welcome_message:  
  
Capybara::ElementNotFound: Unable to find field "Correo electrónico"  
  
    test/features/user_signup_test.rb:15:in `block (2 levels) in <class:User-  
SignupTest>'  
  
    test/features/user_signup_test.rb:14:in `block in <class:UserSignupTest>'  
  
9 runs, 22 assertions, 0 failures, 1 errors, 0 skips
```

Finalmente sólo tenemos una prueba que falla. El problema, como vemos, es que no encuentra el campo *Correo electrónico*, aún y cuando ya lo agregamos al formulario.

El campo está ahí, el problema es que la etiqueta muy probablemente dice *Email*. Vamos a agregar traducciones al modelo y los atributos del modelo a través del archivo `config/locales/es.-models.yml`, el cuál no existe pero vamos a crear.

```
# config/locales/es.models.yml
```

```
es:
```

```
activerecord:
```

```
models:
```

```
    user: 'Usuario'
```

```
attributes:
```

```
user:
```

```
email: 'Correo electrónico'
```

```
password: 'Contraseña'
```

```
password_confirmation: 'Confirmar contraseña'
```

Desafortunadamente al ejecutar las pruebas vemos que la prueba en la que estamos trabajando sigue fallando exactamente igual. El problema es que [Capybara](#) utiliza [Rack::Test](#) que es un navegador [headless](#) es decir sin pantalla visible y [Rack::Test](#) por alguna razón no funciona bien con contenido con tildes y acentos.

Por ejemplo, lo que [Capybara "ve"](#) es lo siguiente:

**Crear una cuenta**

```
Correo electrÃ³nico   
ContraseÃ±a   
Confirmar contraseña   
Crear cuenta
```

En nuestro archivo [Gemfile](#) agregamos la gema [selenium-webdriver](#) pero no lo hemos utilizado aún. Para poder utilizarlo necesitamos de tener instalada la versión más reciente de [Firefox](#).

Para activar [selenium](#) es necesario abrir el archivo [test/test\\_helper.rb](#) y al final del mismo agregar la siguiente línea de código [Capybara.default\\_driver = :selenium](#).

Volvemos a ejecutar nuestras pruebas y vemos que ahora el error es diferente. De paso durante la ejecución de las pruebas pudimos observar que se abrió una pantalla de [Firefox](#) y vimos como [Capybara](#) cargó y llenó el formulario.

```
$ rake test
```

```
Run options: --seed 28445
```

```
# Running:
```

```
.....F....
```

```
Finished in 7.045032s, 1.2775 runs/s, 3.2647 assertions/s.
```

```
1) Failure:
```

```
UserSignupTest#test_fill_in_form_at_/signup_and_see_welcome_message [/Users/mariochavez/Development/one_px/test/features/user_signup_test.rb:22]:
```

```
Expected to include "Bienvenido newuser@mail.com".
```

```
9 runs, 23 assertions, 1 failures, 0 errors, 0 skips
```

Ahora dice que no encuentra el mensaje `Bienvenido newuser@mail.com`, esto indica que se llenó el formulario con información correcta y se envió al controlador, pero el controlador no realizó lo que esperábamos.

Si abrimos el controlador `RegistrationController` vemos que aún no hemos implementado código para el caso de que el modelo `User` sea válido.

Abrimos el archivo `test/controllers/registration_controller_test.rb` y agregamos la siguiente prueba.

```
# test/controllers/registration_controller_test.rb

...

test 'POST /create with valid params' do
  post :create, user: { email: 'newuser@mail.com',
    password: 'pa$$word',
    password_confirmation: 'pa$$word' }

  assert_redirected_to root_path
end

...
```

La prueba consiste en enviar un `POST` al método de `create` del controlador, pero en esta ocasión con parámetros válidos para registrar un usuario.

La comprobación de que el resultado es correcto consiste en redireccionar el usuario al `root_path`.

`root_path` es la ruta raíz en las aplicaciones de Rails, a la cuál accedemos cuando navegamos a `/`, ejemplo <http://localhost:3000/>.

Al ejecutar las pruebas de controlador vemos que la prueba falla.

```
$ rake test:controllers
```

```
Run options: --seed 39453
```

```
# Running:
```

```
.E.
```

```
Finished in 0.187971s, 15.9599 runs/s, 10.6399 assertions/s.
```

1) Error:

```
RegistrationControllerTest#test_POST_/_create_with_valid_params:
```

```
NameError: undefined local variable or method `root_path' for #<RegistrationControllerTest:0x007fb9b899b280>
```

```
test/controllers/registration_controller_test.rb:21:in `block in <class:RegistrationControllerTest>'
```

```
3 runs, 2 assertions, 0 failures, 1 errors, 0 skips
```

Nos dice que la `root_path` no está definido. Para definirlo abrimos el archivo `config/routes.rb` y agregamos la siguiente definición `root to: 'home#index'` como la primera línea en el bloque `do`.

La definición indica que al navegar a `/` queremos que el `HomeController` a través de la acción `index` nos responda.

Ejecutamos nuevamente las pruebas para observar el resultado de nuestro cambio.

```
$ rake test:controllers
```

```
Run options: --seed 19596
```

```
# Running:
```

```
.F.
```

```
Finished in 0.204505s, 14.6695 runs/s, 14.6695 assertions/s.
```

1) Failure:

```
RegistrationControllerTest#test_POST_/create_with_valid_params [/Users/mari-  
ochavez/Development/one_px/test/controllers/registration_controller_test.rb:21]:
```

```
Expected response to be a <redirect>, but was <200>
```

```
3 runs, 3 assertions, 1 failures, 0 errors, 0 skips
```

El error nos indica que nuestra prueba esperaba la redirección, pero lugar de eso obtuvimos el código [200](#) como respuesta. Esta respuesta implica que es hora de agregar la lógica para crear el registro del usuario.

```
# app/controllers/registration_controller.rb

class RegistrationController < ApplicationController

  def new

    @user = User.new

  end

  def create
```

```
  @user = User.new secure_params

  if @user.save

    return redirect_to root_path

  end

  render :new

end

private

def secure_params

  params.require(:user)

  .permit(:email, :password, :password_confirmation)

end
```

```
end
```

La acción `create` ahora iniciativa un objeto a partir del modelo `User` con los parámetros recibidos. El método de `secure_params` es necesario para definir explícitamente cuales parámetros el modelo `User` va a aceptar a partir de la información recibida del formulario y de esta manera prevenir problemas de seguridad en donde se pueda inyectar información maliciosa.

Después de inicializar la instancia el objeto es guardado a la base de datos, si las validaciones no generan error y se pudo guardar el registro, entonces `save` regresa un `true`, en caso contrario regresa `false`.

Cuando el resultado de guardar es `true`, entonces el controlador realiza la redirección a `root_path`.

Al correr la pruebas observamos que la prueba en la que estamos trabajando ya no falla, pero ahora falla la prueba anterior que estaba funcionando hasta antes de el último cambio en el controlador.

```
$ rake test:controllers
```

```
Run options: --seed 62327
```

```
# Running:
```

```
..E
```

```
Finished in 0.189432s, 15.8368 runs/s, 15.8368 assertions/s.
```

1) Error:

```
RegistrationControllerTest#test_POST_/_create_without_params:
```

```
ActionController::ParameterMissing: param is missing or the value is empty: user
```

```
app/controllers/registration_controller.rb:18:in `secure_params'
```

```
app/controllers/registration_controller.rb:7:in `create'
```

```
test/controllers/registration_controller_test.rb:11:in `block in <class:RegistrationControllerTest>'
```

```
3 runs, 3 assertions, 0 failures, 1 errors, 0 skips
```

El motivo de la falla es debido a la introducción de `secure_params`. Hay que hacer un ajuste a la prueba para que quede de la siguiente forma.

```
# test/controllers/registration_controller_test.rb
```

```
...
```

```
test 'POST /create without params' do
```

```
  post :create, user: { email: '' }
```

```
    assert_response :success
```

```
end
```

```
...
```

`user: { email: '' }` es requerido, ya que con la ejecución de `params.require(:user)...` en el cuerpo del método `secure_params` se espera que todas las llamadas a la acción `create` incluya un `Hash` de parámetros con la llave `:user`.

Nuevamente ejecutamos nuestras pruebas y comprobamos.

```
$rake test:controllers
```

```
Run options: --seed 25044
```

```
# Running:
```

```
...
```

```
Finished in 0.181506s, 16.5284 runs/s, 22.0379 assertions/s.
```

```
3 runs, 4 assertions, 0 failures, 0 errors, 0 skips
```

Las pruebas de controladores pasan. Ahora vamos a ejecutar todas las pruebas.

```
$ rake test
```

```
Run options: --seed 5573
```

```
# Running:
```

```
.....E.....
```

```
Finished in 4.784174s, 2.0902 runs/s, 5.0165 assertions/s.
```

1) Error:

```
UserSignupTest#test_fill_in_form_at_/signup_and_see_welcome_message:
```

```
ActionController::RoutingError: uninitialized constant HomeController
```

Tenemos el fallo en la misma prueba que originó los cambios en el `RegistrationController`. El error indica que `HomeController` no existe. Si recuerdan en el archivo de rutas indicamos que al navegar a `/ HomeController#index` debe de responder.

Hay que crear el `HomeController`.

```
# test/controllers/home_controller_test.rb

require 'test_helper'

class HomeControllerTest < ActionController::TestCase

  test 'GET /' do

    get :index

    assert_response :success

  end

end

# app/controllers/home_controller.rb
```

```
class HomeController < ApplicationController

  def index

  end

end

# app/views/home/index.html.erb

<h2>Home</h2>
```

Como vemos en el código, agregamos el archivo de pruebas para el `HomeController` en donde verificamos que al realizar la llamada a `index` obtenemos una respuesta de éxito.

En el código de `HomeController#index` no ejecutamos ninguna instrucción, de momento esto es correcto.

Y en el `template` de `index` sólo desplegamos un mensaje, de momento esto es suficiente.

Ejecutamos las pruebas y verificamos el resultado.

```
$ rake test
```

```
Run options: --seed 13272
```

```
# Running:
```

```
.....F..
```

```
Finished in 6.563716s, 1.6759 runs/s, 3.9612 assertions/s.
```

#### 1) Failure:

```
UserSignupTest#test_fill_in_form_at_signup_and_see_welcome_message [/Users/mariochavez/Development/one_px/test/features/user_signup_test.rb:22]:
```

```
Expected to include "Bienvenido newuser@mail.com".
```

```
11 runs, 26 assertions, 1 failures, 0 errors, 0 skips
```

La prueba continua fallando pero ahora nos indica que no encuentra el texto `Bienvenido newuser@mail.com`.

Vamos a realizar un último ajuste a `RegistrationController#create` para que al momento de crear de forma exitosa el registro de `User` y al redireccionar coloque el mensaje esperado en el objeto `Flash`.

`Flash` es un objeto especial de los controladores que permiten el intercambio de información entre ellos, a costa de utilizar memoria del servidor para tal acción.

`Flash` es efímero, es decir si yo estoy en el controlador **A** y coloco información en '`Flash`', posteriormente redireccono al controlador **B**, al consultar `Flash` ahí voy a encontrar el mensaje enviado por **A**, pero si hago una redirección más, `Flash` pierde ese mensaje.

Para hacer que la prueba pase vamos a modificar los siguientes archivos.

```
# config/locales/es.controllers.yml

es:

registration:

create:

welcome: 'Bienvenido %{email}'

# app/controllers/registration_controller.rb

...

def create

@user = User.new secure_params

if @user.save

return redirect_to root_path,

notice: t('.welcome', email: @user.email)
```

```
    end

    render :new

  end

  ...

# app/views/layouts/application.html.erb

<!DOCTYPE html>

<html>

  <head>

    <title>OnePx</title>

    <%= stylesheet_link_tag 'application', media: 'all', 'data-turbolinks-track' => true %>

    <%= javascript_include_tag 'application', 'data-turbolinks-track' => true %>

    <%= csrf_meta_tags %>
```

```
</head>

<body>

<%-- if flash.notice.present? %>

<div class='flash-notice'>

<%= flash.notice %>

</div>

<%-- end %>

<%= yield %>

</body>

</html>
```

Entre los cambios que realizamos, creamos el archivo `config/locales/es.controllers.yml` y agregamos la estructura de llaves para el mensaje de la traducción a `es.registration.create.-welcome`.

El mensaje '`Bienvenido %{email}`' incluye la notación `%{email}` que implica que al momento de llamar la traducción vamos a poder pasar el parámetro `email` y éste se va a interpolar con el mensaje de `Bienvenido`.

En la acción `create` de `RegistrationController` al realizar la acción de redirección colocamos la llave `notice` del objeto `Flash` con la traducción y el parámetro `email`.

Finalmente y para desplegar el contenido de `flash.notice`, si es que está presente, modificamos el archivo `app/views/layouts/application.html.erb` para desplegar el mensaje.

`app/views/layouts/application.html.erb` es el `template` maestro de nuestra aplicación. Todos los `templates` que hemos creado anteriormente de incrustan dentro de `app/views/layouts/application.html.erb`, específicamente en la definición de `yield`. Más adelante en el libro tenemos con más detalle a `app/views/layouts/application.html.erb`.

Una vez más ejecutamos las pruebas y en esta ocasión todas pasan.

```
$ rake test
```

```
Run options: --seed 17306
```

```
# Running:
```

```
.....
```

```
Finished in 4.822289s, 2.2811 runs/s, 5.3916 assertions/s.
```

```
11 runs, 26 assertions, 0 failures, 0 errors, 0 skips
```

Finalmente lo hemos logrado. Hemos completado nuestra primer característica que permite a un usuario registrarse en nuestras aplicación, pero lo más importante es que lo hemos realizado siguiente la técnica de **Test Driven Development**, es decir hemos creado las pruebas para los diferentes componentes primero y luego de forma iterativa hemos implementado su funcionalidad.

Es momento de realizar un **commit** de nuestro código fuente y sentirnos bien con nosotros por lo que hemos alcanzado.

```
$ git add .
```

```
$ git commit -m "Users can register"
```

```
[user-signup 6e456fd] Users can register
```

```
18 files changed, 333 insertions(+), 18 deletions(-)
```

```
create mode 100644 app/controllers/home_controller.rb
```

```
create mode 100644 app/views/home/index.html.erb
```

```
rewrite app/views/layouts/application.html.erb (88%)
```

```
rewrite app/views/registration/new.html.erb (100%)
```

```
create mode 100644 config/locales/es.controllers.yml
```

```
create mode 100644 config/locales/es.models.yml
```

```
create mode 100644 config/locales/es.views.yml
```

```
create mode 100644 config/locales/es.yml
```

```
create mode 100644 test/controllers/home_controller_test.rb
```

```
$ git push origin user-signup
```

## Conclusiones

En este capítulo hemos aprendido una gran cantidad de cosas sobre como iniciar un proyecto con Ruby on Rails pero también como realizar pruebas automáticas que nos ayuden a dar un nivel de confianza de que nuestra aplicación implementa la funcionalidad requerida.

Para recapitular, aprendimos los siguientes puntos:

- Crear una aplicación de Ruby on Rails
- Sobre **Gemfile** y **Bundler** para el manejo de dependencias
- Conocimos **Capybara** para realizar pruebas automáticas con el navegador
- Utilizar **Minitest** para probar modelos y controladores
- Trabajamos con **I18n** para realizar internacionalización en una aplicación de Rails
- Sobre los generadores código de Rails
- El uso de migraciones para generar y modificar esquema de una base de datos

# Capítulo 5 - Herramientas de apoyo en el desarrollo con Ruby on Rails.

En los capítulos 3 y 4 se han tocado temas relacionados con el uso de pruebas automáticas para ayudar en el desarrollo de una aplicación de Ruby on Rails.

Las pruebas son una parte fundamental para asegurar de que el desarrollo, por lo menos desde el punto de vista técnico, va por el camino adecuado.

Existen herramientas adicionales que podemos utilizar en nuestros proyectos para asegurarnos de que la calidad del código fuente cumple con ciertos requisitos básicos. Estas herramientas generalmente se ejecutan de manera automática cada ocasión que realizamos un *commit* y lo *empujamos* a nuestro repositorio de Github.

En este capítulo, como primer tema, configuraremos nuestra aplicación y nuestro repositorio de Github para que hagan uso de un servidor de integración continua.

El servidor se encargará de ejecutar de forma automática las pruebas de nuestra aplicación.

Cómo segundo tema vamos a conocer tres herramientas y la forma en cómo se van a integrar en nuestro flujo de desarrollo. En su conjunto, todas las herramientas nos ayudarán a realizar un análisis sintáctico, de lineamientos de código y calidad, y finalmente de seguridad; todo esto con el único objetivo de que nuestra aplicación cuente con un código fuente confiable.

## Integración continua

En el capítulo anterior escribimos pruebas para la primera característica que implementamos de nuestra aplicación.

Específica con las pruebas de integración vimos cómo es posible automatizar al navegador para que abra una nueva ventana y ejecute las pruebas simulando la acción de un usuario.

Para tal efecto utilizamos la gema *selenium-webdriver*, la cual require de que contemos con una instalación del navegador Firefox. Esto funciona perfectamente cuando ejecutamos las pruebas en nuestro computador, pero no va a funcionar al momento que deleguemos la ejecución de las

pruebas en el servidor de integración, por la sencilla razón de que estos servidores no cuentan con la interfase gráfica para ejecutar Firefox.

La solución es ejecutar nuestras pruebas en un navegador *headless*. Los navegadores *headless* utilizan el motor de renderización de los navegadores tradicionales, pero se ejecutan en un manejador de ventanas que no crea ventanas visibles.

El párrafo anterior quizás pueda sonar un poco confuso, pero quedará claro una vez que realicemos la implementación y veamos los resultados.

En nuestro proyecto vamos a utilizar *webkit* como nuestro navegador *headless*. *webkit* es el motor web sobre el cuál el navegador *Safari* está basado.

### *Configuración de webkit como servidor headless*

Para poder hacer uso de *webkit* es necesario que instalemos en nuestro computador el *Software Development Kit* o *SDK* de *Qt*.

Dependiendo de el sistema operativo que estemos utilizando, las instrucciones de cómo instalar el *SDK* de *Qt* van a variar.

Esta [guía de instalación](#) contiene instrucciones de instalación para diferentes sistemas operativos y diferentes versiones de los mismos.

Una vez instalado *Qt*, es necesario realizar algunos cambios al archivo `test/test_helper.rb`, los cambios son para indicarle que deseamos usar el navegador *headless*.

Vamos a crear un *branch* de *git* para los cambios en éste capítulo.

```
$ git branch tools-setup
```

```
$ git checkout tools-setup
```

```
Switched to branch 'tools-setup'
```

Pero antes de realizar el cambio, hay que agregar la gema `capybara-webkit` a nuestro archivo `Gemfile`. En el grupo `:development, :test` se encuentra la gema `selenium-webdriver`, la cual agregamos anteriormente para poder ejecutar nuestras pruebas funcionales; a partir de este mo-

mento ya no es necesaria y simplemente la reemplazamos con `capybara-webkit`, de manera que el grupo queda de la siguiente forma:

```
group :development, :test do

  gem 'byebug'

  gem 'minitest-rails'

  gem 'minitest-rails-capybara'

  gem 'capybara-webkit'

  gem 'launchy'

end
```

Como modificamos el archivo que sirve de manifiesto de las dependencias de la aplicación, es necesario ejecutar el comando `bundle` para que actualice las dependencias.

```
$ bundle
```

Ahora si estamos listos para realizar los cambios necesarios a `test/test_helper.rb`. Primera-mente vamos a buscar la línea `Capybara.default_driver = :selenium` y la reemplazamos con `Capybara.javascript_driver = :webkit`.

Posteriormente agregamos el siguiente bloque de configuración, con el cuál le indicamos al nave-gador `headless` que bloquee todos los llamados a URLs que no pertenezcan a nuestra apli-cación, por ejemplo si estamos cargando tipografías de algún sitio como `Google`, realmente no las ocupamos en nuestras pruebas y el descargarlas hace que las pruebas se ejecuten más lento.

De igual forma vamos a desactivas que descargue imágenes en el modo de pruebas, otra vez con la finalidad de acelerar la ejecución de las mismas.

Por tal motivo el archivo `test/test_helper.rb` queda de la siguiente forma:

```
ENV["RAILS_ENV"] = "test"

require File.expand_path("../config/environment", __FILE__)

require "rails/test_help"

require "minitest/rails"

# To add Capybara feature tests add gem "minitest-rails-capybara"

# to the test group in the Gemfile and uncomment the following:

require "minitest/rails/capybara"

# Uncomment for awesome colorful output

# require "minitest/pride"

class ActiveSupport::TestCase

# Setup all fixtures in test/fixtures/*.yml for all tests in alphabetical order.
```

```
fixtures :all

# Add more helper methods to be used by all tests here...

end

Capybara.javascript_driver = :webkit

Capybara::Webkit.configure do |config|

  config.block_unknown_urls

  config.skip_image_loading

end
```

Después de realizar los cambios ejecutamos nuestras pruebas automáticas para comprobar de que siguen funcionando sin problemas, pero también vamos a notar de que ya la ventana del navegador [Firefox](#) no se activa y que las pruebas en general se ejecutan más rápido.

```
$ rake test

Running via Spring preloader in process 79805

Run options: --seed 59452
```

```
# Running:
```

```
.....
```

```
Finished in 0.312306s, 35.2219 runs/s, 83.2518 assertions/s.
```

```
11 runs, 26 assertions, 0 failures, 0 errors, 0 skips
```

En este punto podemos realizar un *commit* y empujar los cambios al repositorio.

```
$ git add .
```

```
$ git commit -m "Change test_helper.rb to use webkit as a headless browser"
```

```
$ git push origin tools-setup
```

## *Configuración de TravisCI como servidor de integración continua*

Hoy en día en el mercado existen una gran variedad de servicios que ofrecen servidores de integración. La oferta en términos de costos y características varían de uno a otro.

Prácticamente todos ofrecen la opción de correr pruebas automáticas sin costo para proyectos *Open Source*. También posible que nosotros configuremos un servidor de pruebas de integración a través de proyectos *Open Source*.

De los servicios actuales de paga podemos encontrar:

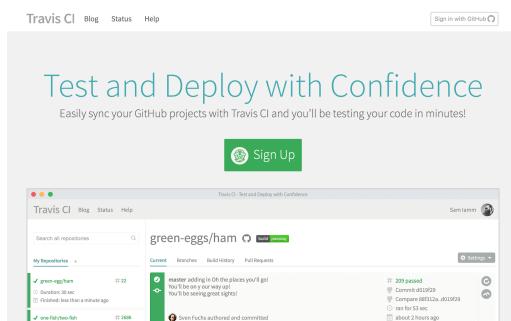
- [TravisCI](#)
- [Codeship](#)
- [CircleCI](#)
- [Gitlab](#)

Pero si deseamos configurar nuestro propio servicio de integración continua podemos utilizar

- [Gitlab](#), así como hay un servicio de paga, existe la posibilidad de descargar el código.
- [Jenkins](#)

Para el proyecto en el que estamos trabajando vamos a hacer uso de TravisCI, y como el código del proyecto es *Open Source* vamos a poder hacer uso de la versión gratuita de TravisCI.

El motivo de la elección de TravisCI sobre el resto es simple, es uno de los primeros servicios de este tipo que existió para las aplicaciones de Ruby. Hoy en día TravisCI ofrece integración continua para múltiples lenguajes y plataformas.



Los primero que vamos a hacer es crear una cuenta con TravisCI, para tal efecto debemos de contar con una cuenta en [Github](#). Al iniciar sesión en Github, tendremos que confirmar que queremos enlazar nuestra cuenta de Github con la cuenta de TravisCI.

Una vez que se haya creado la cuenta, vamos a tener acceso al [dashboard](#) de TravisCI.

En el *dashboard* en la parte de la izquierda, donde dice *My Repositories* hay un símbolo de +, al hacer *click* en él, TravisCI nos mostrará un listado de nuestros proyectos públicos en Github. Del listado en necesario buscar nuestro proyecto y *activar* el botón de la izquierda para que se ponga verde.



Ahora es necesario hacer *click* en el ícono de engrane para ver la configuración del proyecto.

En esta pantalla hay que activar la opción que dice *Build only if .travis.yml is present*.

Ahora hay que crear el archivo *.travis.yml* en el directorio raíz de nuestra aplicación. Este archivo contendrá información sobre nuestro proyecto, la cuál le ayudará a TravisCI a ejecutar las pruebas de forma automática.

El archivo es relativamente simple, al menos para esta parte del desarrollo.

```
#.travis.yml
```

```
language: ruby
```

```
cache: bundler
```

```
rvm:
```

```
- 2.3.0
```

```
bundler_args: --without production
```

```
before_script:
```

```
- RAILS_ENV=test bundle exec rake db:setup
```

Primeramente le indicamos a TravisCI que nuestro proyecto usa el lenguaje *Ruby*. En la segunda línea le pedimos a TravisCI que en la medida de lo posible realice un *cache* de las gemas de nuestra aplicación, de forma que trate de evitar realizar una instalación desde cero cada que se ejecuten nuestras pruebas.

En las 2 siguientes líneas le pasamos la versión de *Ruby* sobre la cuál deseamos probar nuestra aplicación, aquí es posible indicarle más una versión del lenguaje.

En las últimas 3 líneas, primeramente pedimos que no se instalen las gemas que son requisito únicamente en modo de producción y que antes de ejecutar nuestras pruebas configure la base de datos.

TravisCI nos proporciona *Postgresql* para el ambiente de nuestra aplicación.

Ahora realizamos un *commit* de *git* para mandar el archivo *.travis.yml*

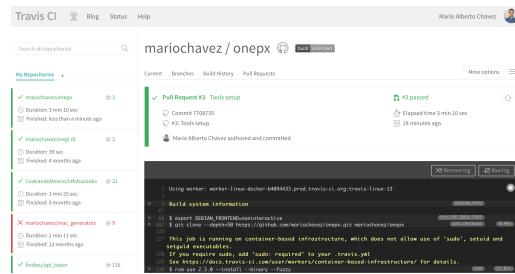
```
$ git add .
```

```
$ git commit -m "Add .travis.yml for Continous Integration"
```

```
$ git push origin tools-setup
```

Una vez que los cambios han sido enviados al repositorio en tan sólo unos segundos vamos a ver que el *dashboard* de TravisCI ya recibió la información y se está preparando para ejecutar nuestras pruebas.

Después de algunos minutos vamos a tener el resultado de la ejecución. Verde si todas la pruebas pasaron, Rojo si alguna prueba falló. El detalle de la ejecución va a estar disponible para que podamos inspeccionar el resultado.



Ahora ya contamos con un servidor independiente que se encarga de ejecutar nuestras pruebas cada vez que enviamos un [commit](#) a nuestro repositorio.

## Escribiendo código con estilo

Es importante que al escribir nuestras aplicaciones en [Ruby](#) tengamos un estilo de código que sea consistente a través de todos los archivos. Es aún más importante cuando el desarrollo es llevado no por una sola persona, pero por un equipo.

A través de la experiencia escribiendo aplicaciones en [Ruby](#), la misma comunidad de desarrolladores ha llegado a cierta conclusión de cómo deben ser las reglas para nuestro código, estas conclusiones han sido registradas en una [Guía de estilo de código](#).

La guía en términos generales la seguimos todos los que escribimos código en [Ruby](#), pero la guía es solamente eso, una guía y cada equipo puede darse la libertad de hacer ciertos ajustes y escribir el código un poco diferente.

El problema de escribir código con estilo consistente radica en la disciplina del desarrollador para seguir la guía - o la guía con los ajustes aceptados - al pie de la letra.

Por este motivo existe una herramienta llamada [Rubocop](#), la cual permite la revisión automática del código fuente de nuestro proyecto y nos notifica cuando el código no sigue un estilo predefinido.

[Rubocop](#) es lo que se conoce como un analizador de código estático, el cual reporta violaciones al estilo del código e inclusive puede corregir de forma automática ciertos de los problemas notificados. Así mismo permite la modificación de ciertos estilos del código de forma que nosotros podamos definir las excepciones a la guía de estilo que creamos pertinentes.

La configuración por omisión de *Rubocop* para realizar análisis del código y reportar violaciones a la guía de estilo se encuentra en este [archivo](#). Cada una de las entradas es un *Cop* y define una de las reglas de la guía.

Por ejemplo:

**StringLiterals:**

**EnforcedStyle: double\_quotes**

Es un *Cop* de estilo, el cuál se aplica a las cadenas de texto y define como debe de ser la declaración, por ejemplo **double\_quotes** indica que las cadenas de texto tienen que ir delimitadas por **"** y no por **'**.

Para instalar *Rubocop* en nuestro proyecto hay que agregar la gema correspondiente al archivo **Gemfile**, la gema va dentro del grupo de **:development, :test**.

```
# Gemfile

...
group :development, :test do
  gem 'byebug'
  gem 'minitest-rails'
  gem 'minitest-rails-capybara'
  gem 'capybara-webkit'
  gem 'launchy'
```

```
gem 'rubocop', require: false
```

```
end
```

Ejecutamos el comando de `bundle`.

```
$ bundle
```

Y listo *Rubocop* está disponible para nuestro proyecto. Si lo ejecutamos con el siguiente comando, vamos a ver la lista de las ofensas:

```
$ rubocop
```

```
...
```

```
test/test_helper.rb:13:7: C: Use nested module/class definitions instead of compact style.
```

```
class ActiveSupport::TestCase
```

```
^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
test/test_helper.rb:14:81: C: Line is too long. [82/80]
```

```
# Setup all fixtures in test/fixtures/*.yml for all tests in alphabetical order.
```

```
^^
```

```
test/test_helper.rb:22:1: C: Use 2 (not 3) spaces for indentation.
```

```
config.block_unknown_urls
```

^^^

```
37 files inspected, 73 offenses detected
```

73 ofensas en 37 archivos!. Parece que no tenemos una guía de código fuente muy definida, aunque quizás no sea del todo cierto.

Hay que tener en mente que *Rubocop* es sólo una herramienta que realiza una serie de recomendaciones sobre nuestro código de fuente, pero no tenemos que obedecerlo al 100%. Y es por eso que lo podemos personalizar para que se ajuste a nuestro estilo.

Para personalizar *Rubocop* hay que crear el archivo `.rubocop.yml` en el directorio raíz de nuestra aplicación.

En el archivo vamos a incluir algunas excepciones y algunos cambios de reglas. Las excepciones son principalmente para los archivos que *Rails* genera para nosotros al crear una nueva aplicación, estos archivos siguen cierta guía de estilo que puede ser o no compatible con la nuestra, es por eso que es mejor ignorarlos.

Los cambios a la regla van a ser en función a cómo es nuestro estilo de escribir código. Para conocer en qué consisten los cambios hay que consultar el [archivo de Cops](#) con sus descripciones.

```
#.rubocop.yml
```

```
AllCops:
```

```
Exclude:
```

- "vendor/\*\*/\*"
- "spec/fixtures/\*\*/\*"

– "tmp/\*\*/\*"

– "db/\*\*/\*"

– "bin/\*\*/\*"

– "config.ru"

– "Gemfile"

– "Rakefile"

– "config/\*\*/\*"

**TargetRubyVersion:** 2.3

**DisplayCopNames:** true

**ExtraDetails:** true

**Rails:**

**Enabled:** true

**Style/Encoding:**

**EnforcedStyle:** when\_needed

**Enabled:** true

**AutoCorrectEncodingComment:** "# encoding: utf-8"

**Style/FrozenStringLiteralComment:**

**EnforcedStyle:** when\_needed

**StringLiterals:**

**EnforcedStyle:** double\_quotes

**Style/Documentation:**

**Enabled:** false

Metrics/LineLength:

Exclude:

- "config/\*\*/\*"

Después de crear el archivo `.rubocop.yml` ejecutamos el comando `rubocop` y vemos que ahora el número de violaciones a la guía ha bajado.

```
$ rubocop
```

```
...
```

```
test/test_helper.rb:1:1: C: Style/FrozenStringLiteralComment: Missing frozen  
string literal comment.
```

```
ENV["RAILS_ENV"] = "test"
```

```
^
```

```
test/test_helper.rb:13:7: C: Style/ClassAndModuleChildren: Use nested module/  
class definitions instead of compact style.
```

```
class ActiveSupport::TestCase
```

```
^^^^^^^^^^^^^^^^^^^^^^^^
```

```
test/test_helper.rb:14:81: C: Metrics/LineLength: Line is too long. [82/80]
```

```
# Setup all fixtures in test/fixtures/*.yml for all tests in alphabetical order.
```

^^

```
test/test_helper.rb:22:1: C: Style/IndentationWidth: Use 2 (not 3) spaces for indentation.
```

```
config.block_unknown_urls
```

^^^

```
10 files inspected, 61 offenses detected
```

Ahora es momento de abrir cada uno de los archivos reportados por *Rubocop* y realizar los ajustes solicitados en las 61 violaciones o podemos hacer que *Rubocop* soluciones algunas de ellas por nosotros.

```
$ rubocop -a
```

...

```
test/test_helper.rb:22:1: C: [Corrected] Style/IndentationWidth: Use 2 (not 3) spaces for indentation.
```

```
config.block_unknown_urls
```

^^^

```
test/test_helper.rb:24:4: C: [Corrected] Style/IndentationConsistency: Inconsistent indentation detected.
```

```
config.skip_image_loading
```

```
~~~~~
```

```
10 files inspected, 62 offenses detected, 60 offenses corrected
```

El argumento `-a` en *Rubocop* le indica que trate de corregir de manera automática las violaciones reportadas. Como vemos en el resultado, nos indica que de 62 violaciones ha corregido 60, nada mal.

Vamos a ejecutar el comando de `rubocop` nuevamente para obtener el reporte de las violaciones que vamos a tener que corregir manualmente.

```
$ rubocop
```

```
...
```

```
Inspecting 10 files
```

```
.....C
```

```
Offenses:
```

```
test/test_helper.rb:14:7: C: Style/ClassAndModuleChildren: Use nested module/
class definitions instead of compact style.
```

```
class ActiveSupport::TestCase
```

```
~~~~~
```

```
test/test_helper.rb:15:81: C: Metrics/LineLength: Line is too long. [82/80]
```

```
# Setup all fixtures in test/fixtures/*.yml for all tests in alphabetical or-
der.
```

```
^^
```

```
10 files inspected, 2 offenses detected
```

De abajo hacia arriba, vemos que la primera violación es en relación a la longitud de la línea de código en el archivo `test/test_helper.rb`, esa es muy sencilla de corregir.

Sobre la segunda violación nos indica que en lugar de declarar la clase `TestCase` como:

```
class ActiveSupport::TestCase
```

```
...
```

```
end
```

La declaremos de la siguiente forma:

```
module ActiveSupport
```

```
  class TestCase
```

```
    ...
```

```
  end
```

```
end
```

Lo que produce el mismo resultado, pero cumple con la guía de estilo. Después del cambio ejecutamos nuevamente `rubocop` y vemos que nuestro código cumple con la guía de estilo al 100%.

```
$ rubocop
```

```
Inspecting 10 files
```

```
.....
```

```
10 files inspected, no offenses detected
```

Ahora ejecutamos nuestras pruebas para asegurarnos que los cambios en el estilo del código no crearon algún problema en la funcionalidad.

```
$ rake test
```

```
Running via Spring preloader in process 87229
```

```
Run options: --seed 55245
```

```
# Running:
```

```
.....
```

```
Finished in 0.334506s, 32.8843 runs/s, 77.7265 assertions/s.
```

```
11 runs, 26 assertions, 0 failures, 0 errors, 0 skips
```

Todo está en orden. Ahora como parte del flujo de desarrollo además de ejecutar nuestras pruebas es necesario ejecutar *Rubocop* para asegurarnos que nuestro código está en orden. Un poco más adelante vamos a ver como automatizar nuestro flujo.

Este es un buen momento para realizar un *commit* con nuestros cambios.

```
$ git add .
```

```
$ git commit -m "Add rubocop to check our code style. Fix code style."
```

```
$ git push origin tools-setup
```

## Eliminando los malos olores de nuestro código

Parte del trabajo de desarrollar una aplicación es obviamente que la aplicación permita a sus usuarios realizar una serie de acciones, pero es también parte de la responsabilidad del equipo de desarrollo el tratar de crear código fuente que sea fácil de entender, mantener y modificar.

Se escucha como una tarea sencilla, pero es más complicado de lo que suena. En inglés hay una definición llamada *Code smells* y se utiliza precisamente para identificar las partes del código que no permitan las 3 acciones mencionadas anteriormente.

Estos *smells* no son necesariamente cosas que están mal en nuestro y en muchas de las ocasiones son totalmente subjetivas. Pero em base a la experiencia que nos da el desarrollar software, es posible identificar esos lugares que pueden ser área de oportunidad para mejorar nuestro código.

Es aquí donde entra la herramienta *Reek*. *Reek* es una herramienta que analiza nuestro código fuente en busca de *code smells*, específicamente los problemas que detecta se encuentran en esta [lista](#).

Para integrar *Reek* en nuestro proyecto es necesario agregar su gema a la sección de `:development, :test` de nuestro *Gemfile*.

```
# Gemfile

...
group :development, :test do
  gem 'byebug'
  gem 'capybara-webkit'
  gem 'minitest-rails'
  gem 'minitest-rails-capybara'
```

```
gem 'launchy'

gem 'reek', require: false

gem 'rubocop', require: false

end
```

Una vez hecho esto, es necesario ejecutar el comando `bundle` para actualizar las gemas.

```
$ bundle
```

*Reek* al igual que *Rubocop* cuenta con un archivo de configuración, este archivo debe de existir en el directorio raíz de la aplicación y se debe de llamar `.reek`. En él podemos desactivar, excluir o modificar la forma en como *Reek* va a detectar problemas en el código.

```
#.reek
```

```
---
```

```
"app/controllers":
```

```
IrresponsibleModule:
```

```
enabled: false
```

```
NestedIterators:
```

```
max_allowed_nesting: 2
```

```
UnusedPrivateMethod:
```

```
    enabled: false
```

```
TooManyStatements:
```

```
    max_statements: 8
```

```
"app/helpers":
```

```
IrresponsibleModule:
```

```
    enabled: false
```

```
UtilityFunction:
```

```
    enabled: false
```

```
exclude_paths:
```

```
    - db
```

El archivo `.reek` mostrado arriba, cuenta con configuraciones por omisión que van a funcionar bien en una aplicación de Ruby on Rails.

Ejecutando el comando `reek` obtenemos el reporte de las áreas de oportunidad en nuestro código fuente.

```
$ reek

test/models/user_test.rb -- 1 warning:

[6, 25, 38]:UncommunicativeVariableName: UserTest has the variable name
'u' [https://github.com/troessner/reek/blob/master/docs/Uncommunicative-Variable-
Name.md]

1 total warning
```

Sólo un tipo de problema fue reportado en las líneas 6, 25 y 38 del archivo `test/models/user_test.rb`, el mensaje indica que estamos usando un nombre de variable poco comunicativo. Vamos a solucionar el problema cambiando el nombre de la variable de `u` a `model`.

Ejecutamos el comando `reek`.

```
$ reek

0 total warnings
```

Cero notificaciones, corregimos, en este caso, un problema que podría haber resultado en falta de claridad en nuestro código. Un poco más adelante vamos ver como podemos automatizar `Reek` en nuestro flujo de desarrollo.

Ahora es buen momento para realizar un `commit` de los cambios realizados.

```
$ git add .

$ git commit -m "Add reek to check our code smells. Fix code smells."

$ git push origin tools-setup
```

## Escribiendo código seguro

Otra actividad clave de escribir código para una aplicación, es el escribir código de manera segura. Generalmente los frameworks como Ruby on Rails ofrecen cierto nivel de protección, pero no cubren todos los casos posibles de problemas de seguridad.

Escribir código seguro no es una tarea sencilla, ya que código inseguro puede ser incluido en una aplicación por lo menos de 2 formas diferentes:

- Código que el desarrollador escribe
- Código proveniente de una gema o del mismo framework

Para asegurarnos que en nuestra aplicación estamos escribiendo código con un cierto nivel de seguridad vamos a hacer uso de [\*\*Brakeman\*\*](#). *Brakeman* es un buscador de [\*\*vulnerabilidades\*\*](#) diseñado para Ruby on Rails.

*Brakeman* ayuda a descubrir problemas de seguridad en diferentes áreas de nuestra aplicación.

Para utilizar *Brakeman* es muy sencillo, definimos el uso de la gema en el archivo **Gemfile** dentro del grupo **:development, :test**.

```
#Gemfile  
  
...  
  
group :development, :test do  
  
  gem 'brakeman', require: false  
  
  gem 'byebug'  
  
  gem 'capybara-webkit'  
  
  gem 'minitest-rails'
```

```
gem 'minitest-rails-capybara'

gem 'launchy'

gem 'reek', require: false

gem 'rubocop', require: false

end
```

Y ejecutamos el comando **bundle**.

```
$ bundle
```

Una vez completados los pasos anteriores solamente ejecutamos **brakeman**.

```
$ brakeman
```

```
...
```

```
+BRAKEMAN REPORT+
```

```
Application path: /Users/mariochavez/Development/one_px
```

```
Rails version: 4.2.6
```

```
Brakeman version: 3.2.1
```

Started at 2016-05-09 10:55:36 -0500

Duration: 0.243804 seconds

Checks run: BasicAuth, BasicAuthTimingAttack, ContentTag, CreateWith, CrossSiteScripting, DefaultRoutes, Deserialize, DetailedExceptions, DigestDoS, DynamicFinders, EscapeFunction, Evaluation, Execute, FileAccess, FileDisclosure, FilterSkipping, ForgerySetting, HeaderDoS, I18nXSS, JRubyXML, JSONEncoding, JSONParsing, LinkTo, LinkToHref, MailTo, MassAssignment, MimeTypeDoS, ModelAttrAccessible, ModelAttributes, ModelSerialize, NestedAttributes, NestedAttributesBypass, NumberToCurrency, QuoteTableName, Redirect, RegexDoS, Render, RenderDoS, RenderInline, ResponseSplitting, RouteDoS, SQL, SQLCVEs, SSLVerify, SafeBufferManipulation, SanitizeMethods, SelectTag, SelectVulnerability, Send, SendFile, SessionManipulation, SessionSettings, SimpleFormat, SingleQuotes, SkipBeforeFilter, StripTags, SymbolDoSCVE, TranslateBug, UnsafeReflection, ValidationRegex, WithoutProtection, XMLDoS, YAMLParsing

+SUMMARY+

+-----+-----+

| Scanned/Reported | Total |

+-----+-----+

|             |   |  |
|-------------|---|--|
| Controllers | 3 |  |
|-------------|---|--|

|        |   |  |
|--------|---|--|
| Models | 1 |  |
|--------|---|--|

|           |   |  |
|-----------|---|--|
| Templates | 3 |  |
|-----------|---|--|

|        |   |  |
|--------|---|--|
| Errors | 0 |  |
|--------|---|--|

|                   |       |  |
|-------------------|-------|--|
| Security Warnings | 0 (0) |  |
|-------------------|-------|--|

+-----+-----+

Al finalizar la ejecución del comando, vemos la lista de las vulnerabilidades que *Brakeman* revisa, así como un reporte de los problemas encontrados. En nuestro caso no encontró ningún problema.

*Brakeman* es sólo parte de la ayuda que podemos obtener al revisar la seguridad en nuestras aplicaciones, como se mencionó anteriormente, el análisis de nuestro código es sólo una parte de las auditorias de seguridad.

La otra parte proviene de las gemas de las que hace uso nuestra aplicación. Cuando se encuentra algún problema de seguridad en alguna de la gemas, generalmente se hace un [reporte de vulnerabilidades](#) o CVE.

El revisar estos reportes puede ser una tarea ardua, afortunadamente existe [bundler-audit](#) que nos ayuda a automatizar estas revisiones y alertarnos de las gemas que puedan representar algún riesgo para nuestra aplicación.

Para instalar *bundler-audit* sólo tenemos que agregar la gema correspondiente al archivo [Gemfile](#) en la sección de `:development, :test` y ejecutar el comando `bundle`.

```
#Gemfile
```

```
group :development, :test do
```

```
gem 'brakeman', require: false

gem 'bundler-audit', require: false

gem 'byebug'

gem 'capybara-webkit'

gem 'minitest-rails'

gem 'minitest-rails-capybara'

gem 'launchy'

gem 'reek', require: false

gem 'rubocop', require: false

end

$ bundle
```

Ahora podemos iniciar una auditoría sobre las gemas instaladas, solamente ejecutando el comando **bundle-audit**.

```
$ bundle-audit
```

```
Insecure Source URI found: git://github.com/turbolinks/turbolinks-rails.git
```

Vulnerabilities found!

En el caso de nuestra aplicación únicamente indica como inseguro el uso de la gema *Turbolinks* desde un repositorio de *Github*, no porque *Turbolinks* tenga algún problema reportado de vulnerabilidad, pero porque incluir gemas directamente de *Github* puede representar un problema.

Incluimos *Turbolinks* de este modo, ya que al momento de escribir esta sección del libro, no hay una gema oficial liberada con el nuevo código de *Turbolinks*.

Algo importante que debemos de realizar con *bundle-audit* cada cierto tiempo, es actualizar la base de datos de vulnerabilidades, esto lo podemos realizar con el siguiente comando:

```
$ bundle-audit update
```

```
Updating ruby-advisory-db ...
```

```
remote: Counting objects: 36, done.
```

```
remote: Compressing objects: 100% (12/12), done.
```

```
remote: Total 36 (delta 13), reused 9 (delta 9), pack-reused 13
```

```
Unpacking objects: 100% (36/36), done.
```

```
From https://github.com/rubysec/ruby-advisory-db
```

```
* branch          master      -> FETCH_HEAD
```

```
cca8b77..96ce851  master      -> origin/master
```

```
Updating cca8b77..96ce851
```

## Fast-forward

```
gems/RedCloth/{OSVDB-115941.yml => CVE-2012-6684.yml} | 7 ++++++-
```

```
gems/administrate/CVE-2016-3098.yml | 14 ++++++-----
```

```
gems/safemode/CVE-2016-3693.yml | 13 ++++++-----
```

```
gems/spina/CVE-2015-4619.yml | 16 ++++++-----
```

```
4 files changed, 49 insertions(+), 1 deletion(-)
```

```
rename gems/RedCloth/{OSVDB-115941.yml => CVE-2012-6684.yml} (71%)
```

```
create mode 100644 gems/administrate/CVE-2016-3098.yml
```

```
create mode 100644 gems/safemode/CVE-2016-3693.yml
```

```
create mode 100644 gems/spina/CVE-2015-4619.yml
```

## Updated ruby-advisory-db

```
ruby-advisory-db: 267 advisories
```

Esta es la forma en como mantenemos actualizada su base de datos y puede alertarnos de posibles problemas.

Con [bundle-audit](#) finalizamos la sección de escribiendo código seguro y es el momento para realizar un [commit](#) de los cambios realizados.

```
$ git add .
```

```
$ git commit -m "Add brakeman and bundle-audit to check code vulnerabilities."
```

```
$ git push origin tools-setup
```

## Integrando las herramientas en el flujo de trabajo

Ya hemos visto como nos podemos apoyar en diversas herramientas para el desarrollo de nuestras aplicaciones, ahora vamos a ver cómo podemos integrarlas en nuestro flujo de trabajo que nosotros y/o nuestro equipo las utilicen y de esa forma colaborar en hacer aplicaciones mantenibles y seguras.

La forma más simple de integración puede ser través de *Rake*. Cómo ya vimos en uno de los capítulos anteriores, *Rake* es una herramienta de automatización de tareas, en donde las tareas se escriben en *Ruby*.

Como ejemplos de *Rake* tenemos `rake test` que se encarga de ejecutar nuestras pruebas automáticas o `rake db:migrate` que se encarga de aplicar migraciones a la base de datos.

Para la integración de las herramientas de calidad y seguridad vamos a escribir una tarea de *Rake* que nos permita 3 cosas:

- Correr todas herramientas juntas con un sólo comando
- Correr sólo las herramientas de calidad
- Correr sólo las herramientas de seguridad

Para crear nuestra tarea a Ruby on Rails, tenemos el directorio `lib/tasks` en donde cualquier archivo que se encuentre ahí con la extensión `rake` va a ser reconocido.

Nuestro archivo lo vamos a crear como `lib/tasks/quality.rake`.

```
#lib/tasks/quality.rake
```

```
# frozen_string_literal: true
```

```
begin
```

```
require "brakeman"

require "bundler/audit/task"

require "reek/rake/task"

require "rubocop/rake_task"

Bundler::Audit::Task.new

namespace :brakeman do

  desc "Check your code with Brakeman"

  task :check do

    result = Brakeman.run app_path: ".", print_report: true

    unless result.filtered_warnings.empty?

      exit Brakeman::Warnings_Found_Exit_Code

    end

  end

end
```

```
    end
```

```
end
```

```
RuboCop::RakeTask.new do |task|
```

```
  task.fail_on_error = false
```

```
end
```

```
Reek::Rake::Task.new do |task|
```

```
  task.fail_on_error = false
```

```
end
```

```
namespace :quality do
```

```
  desc "Check your code quality and security"
```

```
  task check: ["rubocop", "reek", "brakeman:check", "bundle:audit"]
```

```

namespace :check do

  desc "Check your code quality"

  task quality: %w(rubocop reek)

  desc "Check your code security"

  task security: ["brakeman:check", "bundle:audit"]

end

rescue StandardError => e

  puts "Quality check failed: #{e.message}"

end

```

A partir de la línea que contiene `namespace :quality do` es donde declaramos las 3 tareas de *Rake* anteriormente mencionadas, por lo que si ejecutamos `rake -T` para que nos muestre las tareas disponibles, las 3 tareas aparecen ahí entre todas las demás:

```
$ rake -T  
...  
  
rake quality:check          # Check your code quality and security  
  
rake quality:check:quality  # Check your code quality  
  
rake quality:check:security # Check your code security  
  
...
```

`rake quality:check` corre todas las herramientas, incluidas calidad y seguridad a un mismo tiempo.

`rake quality:check:quality` corre todas las herramientas de calidad, es decir *Rubocop* y *Reek*.

Y finalmente `rake quality:check:security` corre todas las herramientas de seguridad, *Brakeman* y *Bundler Audit*.

Los integrantes de nuestro equipo de desarrollo y nosotros mismos ya podemos ejecutar las tareas en cualquier momento y tomar acciones sobre el código que estamos escribiendo, pero ¿qué pasa si alguno de los miembros del equipo hace `commit` sin revisar las recomendaciones de las herramientas?

Gracias a que ya tenemos un servidor de integración como *TravisCI*, podemos automatizarlo, no sólo para que ejecute las pruebas, pero también ejecute las herramientas y en caso de que alguna muestre alguna recomendación, la recomendación se va a escribir como un comentario en Github.



Para que esto suceda es necesario agregar la gema de *Pronto* a nuestro proyecto, así como los *runners* de las herramientas que necesitamos que se ejecuten en *TravisCI*. Los *runners* son gemas de *Pronto* que le permite ejecutar *Brakeman*, por ejemplo.

Vamos a agregar *Pronto* y los *runners* para *Brakeman*, *Reek* y *Rubocop* a nuestro archivo *Gemfile* en la sección de `:development, :test`.

```
# Gemfile

...
group :development, :test do
  gem 'brakeman', require: false
  gem 'bundler-audit', require: false
  gem 'byebug'
  gem 'capybara-webkit'
  gem 'minitest-rails'
  gem 'minitest-rails-capybara'
```

```
gem 'launchy'

gem 'pronto'

gem 'pronto-brakeman', require: false

gem 'pronto-reek', require: false

gem 'pronto-rubocop', require: false

gem 'reek', require: false

gem 'rubocop', require: false

end
```

Ahora ejecutamos el comando `bundle`:

```
$ bundle
```

El siguiente paso va a ser indicarle a `TravisCI`, a través de su archivo de configuración `.travis.yml` que deseamos ejecutar las herramientas.

```
# .travis.yml
```

```
language: ruby
```

```
cache: bundler
```

```
rvm:
```

- 2.3.0

```
bundler_args: --without production
```

```
before_script:
```

```
- RAILS_ENV=test bundle exec rake db:setup
```

```
script:
```

```
- bundle exec rake test
```

```
- 'if [ "$TRAVIS_PULL_REQUEST" == false ]; then echo "TRAVIS_PULL_REQUEST is
unset, skipping Pronto"; else PULL_REQUEST_ID=${TRAVIS_PULL_REQUEST} bundle exec
pronto run -f github_pr; fi'
```

Para que desde *TravisCI* se puedan enviar comentarios a *Github*, es necesario en *TravisCI* configurar una variable **GITHUB\_ACCESS\_TOKEN**, el valor de esa variable lo vamos a generar desde *GitHub*, pero es necesario navegar a la página de [Personal Access Tokens](#) y ahí presionar el botón *Generate New Token*.

New personal access token

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to authenticate to the API over Basic Authentication.

**Token description**

**Select scopes**

Scopes define the access for personal tokens. [Read more about OAuth scopes.](#)

|   |   |
|---|---|
| <input checked="" type="checkbox"/> <b>repo</b>     | Full control of private repositories              |
| <input checked="" type="checkbox"/> repo:status     | Access commit status                              |
| <input checked="" type="checkbox"/> repo_deployment | Access deployment status                          |
| <input checked="" type="checkbox"/> public_repo     | Access public repositories                        |
| <input type="checkbox"/> admin:org                  | Full control of orgs and teams                    |
| <input type="checkbox"/> write:org                  | Read and write org and team membership            |
| <input type="checkbox"/> read:org                   | Read org and team membership                      |
| <input type="checkbox"/> admin:public_key           | Full control of user public keys                  |
| <input type="checkbox"/> write:public_key           | Write user public keys                            |
| <input type="checkbox"/> read:public_key            | Read user public keys                             |
| <input type="checkbox"/> admin:repo_hook            | Full control of repository hooks                  |
| <input type="checkbox"/> write:repo_hook            | Write repository hooks                            |
| <input type="checkbox"/> read:repo_hook             | Read repository hooks                             |
| <input type="checkbox"/> admin:org_hook             | Full control of organization hooks                |
| <input type="checkbox"/> gist                       | Create gists                                      |
| <input type="checkbox"/> notifications              | Access notifications                              |
| <input type="checkbox"/> user                       | Update all user data                              |
| <input type="checkbox"/> user:email                 | Access user email addresses (read-only)           |
| <input type="checkbox"/> user:follow                | Follow and unfollow users                         |
| <input type="checkbox"/> delete_repo                | Delete repositories                               |
| <input type="checkbox"/> admin:gpg_key              | Full control of user gpg keys (Developer Preview) |
| <input type="checkbox"/> write:gpg_key              | Write user gpg keys                               |
| <input type="checkbox"/> read:gpg_key               | Read user gpg keys                                |

**Generate token**

Le asignamos un nombre y activamos el *check* en donde dice **repo**, luego presionamos el botón **Generate Token**. Copiamos el valor y lo guardamos, es importante notar que **Github** ya no nos volverá a mostrar ese *token* si lo perdemos o nos olvidamos, es necesario eliminarlo y crear uno nuevo.

Ahora vamos a nuestro proyecto en **TravisCI** y nos dirigimos al área de configuración en donde definimos la variable **GITHUB\_ACCESS\_TOKEN** con el valor del token que obtuvimos de **Github**.



Después de terminar la configuración vamos a realizar un *commit* de los cambios en el proyecto, esos cambios son los que van a activar que **TravisCI** ejecute las herramientas contra nuestro código.

```
$ git add .
```

```
$ git commit -m "Add a rake task to run quality tools, setup Travis CI to run our tools via Pronto"
```

```
$ git push origin tools-setup
```

Como prueba generé un [commit](#) con código que genera una recomendación por parte de [Reek](#) y aquí está la imagen que muestra como [TravisCI](#) insertó un comentario en [Github](#).

The screenshot shows a GitHub pull request diff for a file named `test/models/user_test.rb`. The diff highlights several lines of code with color-coded annotations:

| Line | Annotation | Code   |
|------|------------|--|
| 33   | Red        | <code>    33  ... @@ -33,11 +34,11 @@ class UserTest &lt; ActiveSupport::TestCase</code> |
| 34   | Red        | <code>    34      assert_empty user.errors[:password]</code>                             |
| 35   | Red        | <code>    35    end</code>   |
| 36   | Red        | <code>    36    - test 'Invalid user with short password' do</code>                      |
| 37   | Red        | <code>    37      - user = User.new.tap do  u </code>                                    |
| 38   | Red        | <code>    38        u.email = 'test@mail.com'</code>                                     |
| 39   | Red        | <code>    39        u.password = 'pa\$\$'</code>   |
| 40   | Red        | <code>    40        u.password_confirmation = 'pa\$\$'</code>                            |
| 37   | Green      | <code>    37    + test "Invalid user with short password" do</code>                      |
| 38   | Green      | <code>    38    +     user = User.new.tap do  u </code>                                  |

Below the diff, a note from the user `mariochavez` states: "Has the variable name 'm' (UncommunicativeVariableName)".

## Conclusiones

Como vimos en el capítulo, el desarrollar software es más allá de implementar la funcionalidad requerida, también hay que asegurarnos que nuestro código tenga un cierto nivel de calidad, calidad en base a que tan fácil o difícil es entenderlo y modificarlo posteriormente; pero además que nuestro código sea seguro y que no represente un riesgo.

Las herramientas que se mencionaron en este capítulo a final de cuentas son una ayuda más, no tienen la palabra final, esa nos corresponde a nosotros, pero poder decir es necesario comprender el contexto y tomar decisiones sobre seguir o ignorar las recomendaciones.

# Capítulo 6 - Actualizando nuestra aplicación a Ruby on Rails 5.

En los meses anteriores he recibido preguntas si el libro se actualizaría a la versión más reciente de Ruby on Rails, la versión 5.

Después de contemplarlo y dado que la versión 5 ha estado disponible ya por varios meses, he decidido hacer la actualización.

Pero en lugar de reescribir los capítulos previos, tomaré este capítulo para describir el proceso de actualizar una aplicación existente, en este caso la aplicación *OnePx*, buscando que este ejercicio tenga los siguientes objetivos:

- Continuar con el resto del libro en la versión 5 de Ruby on Rails.
- Conocer cuál es el proceso de actualización de una aplicación.

## Ruby on Rails 5

El pasado 30 de Junio del 2016 se anunció la nueva versión de Ruby on Rails<sup>1</sup>, la versión 5.

Esta versión reemplaza a la versión 4.2, pero no es sólo un cambio en el número de versión, Rails 5 ha sido retrabajada en 3 diferentes aspectos importantes:

- Eliminación de código *legacy*
- Mejoras en el rendimiento general de Rails.
- Nueva funcionalidad que nos permite escribir otro tipo de aplicaciones disponibles a través del Internet.

---

<sup>1</sup> <http://weblog.rubyonrails.org/2016/6/30/Rails-5-0-final/>

De igual forma Rails 5 únicamente ofrece soporte para versiones más recientes del lenguaje Ruby. Rails ahora requiere como versión mínima de Ruby la versión 2.2.2 aunque la versión recomendada para nuestras aplicaciones es la versión 2.3.1.

## Ruby 2.3.1

La decisión del equipo de desarrollo de Rails 5 de soportar únicamente el lenguaje Ruby a partir de la versión 2.2.2 se debe que versiones previas de Ruby ya no cuentan con soporte ni mantenimiento, y tentativamente pueden convertirse en un problema de seguridad en nuestras aplicaciones.

Otro motivo es que a partir de la versión 2.2.2 se incluye un nuevo *Garbage Collector* - Recolección de basura - que promete mejorar el rendimiento de nuestras aplicaciones primeramente al tener la posibilidad de liberar memoria de los **Symbol** o símbolos en Ruby, además de separar el espacio de memoria entre objetos nuevos y objetos viejos, haciendo que el barrido de memoria de Ruby para liberar objetos que ya no son necesarios sea más eficiente.

Y aunque la versión 2.2.2 es la mínima que soporta Rails 5, lo ideal es hacer que nuestras aplicaciones funcionen con la versión 2.3.1, ya que esta versión agrega serie de mejoras en el rendimiento de nuestra aplicaciones y nueva sintaxis que hace la vida de nosotros los desarrolladores más sencilla.

La versión 2.3.0 fue liberada el 25 de diciembre del 2015<sup>2</sup> y la versión 2.3.1 fue liberada el 26 de abril del 2016<sup>3</sup>, siendo esta versión denominada de *bug fixes*, es decir no tiene nueva funcionalidad, únicamente incluye corrección de errores.

El capítulo 1 del libro ha sido actualizado con información de cómo instalar la versión 2.3.1 en nuestros ambientes locales de desarrollo, es importante contar con dicha versión o mejor, ya que durante el resto del libro se hará uso en algunos casos de la nueva sintaxis de Ruby 2.3.0.

Veamos algunos ejemplos de los cambios en Ruby 2.3.1.

---

<sup>2</sup> <https://www.ruby-lang.org/en/news/2015/12/25/ruby-2-3-0-released/>

<sup>3</sup> <https://www.ruby-lang.org/en/news/2016/04/26/ruby-2-3-1-released/>

## Did you mean?

En cuantas ocasiones no nos hemos encontrado en el caso en que al escribir un bloque de código nos equivocamos con algún nombre de variable o nombre de método y de Ruby únicamente recibimos un `NameError` o `NoMethodError` sin más ayuda.

Ahora recibimos más ayuda de Ruby e inclusive nos da una sugerencia de cuál es el método o nombre de variable que quisimos utilizar.

```
class Account

  def initialize

    @full_name = 'John Doe'

  end

  def name

    full_name

  end

end

> account = Account.new
```

```
NameError: uninitialized constant Acount
```

```
Did you mean? Account
```

```
> account = Account.new
```

```
> account.nams
```

```
NoMethodError: undefined method `nams' for #<Account:0x007fb2291db068>
```

```
Did you mean? name
```

```
> account.name
```

```
NameError: undefined local variable or method `full_name' for #<Account:0x007fb2292b00d8 @full_name="John Doe">
```

```
Did you mean? @full_name
```

Mucho más útil el mensaje de error.

## Operador de navegación segura

En Ruby podemos tener una variable definida y con valor de asignación `nil`, si esperamos que esa variable contenga la referencia a un objeto, es posible que más adelante en nuestro código deseemos llamar un método de ese objeto, pero si lo hacemos sin ningún tipo de protección podemos generar el error `NoMethodError`.

```
> @user = nil

> @user.profile # this will raise NoMethodError: undefined method `profile' for
nil:NilClass
```

Una forma de prevenir el error es verificando si la variable tiene asignado el valor `nil`.

```
> if !@user.nil?

>   @user.profile

> end
```

O en Rails existe el método `Object#try` que en caso que nuestro objeto sea `nil` regresa `nil` como valor.

```
> @user.try(:profile) # regresa el valor nil
```

Con el nuevo operador de navegación segura `&`, podemos hacer la misma verificación que hacemos con `if` o con `Object#try` en Rails, pero inclusive podemos encadenar llamadas.

```
> @user&.profile&.name # regresa el valor nil
```

## Pragma de congelación de cadenas de texto

Ruby tiene soporte para congelar cadenas de texto, es decir hacerlas no-mutables o no-modificables, esto a través de `Object#freeze`.

```
> text = "Hello".freeze

> text << " World"
```

```
RuntimeError: can't modify frozen String
```

Pero existe la posibilidad de que Ruby 3.0 por omisión congele todas las cadenas de texto. Es por eso que en la versión Ruby 2.3.0 se introdujo un *pragma* que en lugar de tener que indicar de manera explícita que cadenas de texto queremos que se congelen con `Object#freeze`, podemos hacer esto de manera global y así preparar desde hoy nuestros programas para esta posible nueva funcionalidad en Ruby.

Para utilizar el *pragma* solamente tenemos que agregar el siguiente comentario al inicio de cada uno de nuestros archivos de Ruby `# frozen_string_literal: true`.

```
# frozen_string_literal: true
```

```
> text = "Hello"
```

```
> text << "World"
```

```
RuntimeError: can't modify frozen String
```

O bien ejecutar nuestro programa de Ruby con la bandera `--enable-frozen-string-literal-debug`, si no queremos agregar el *pragma* a cada uno de nuestros archivos.

```
$ ruby --enable-frozen-string-literal-debug string_freeze.rb
```

La ventaja de congelar nuestras cadenas de texto es que al hacer esas cadenas inmutables podemos evitar que alguna parte de nuestros programas por accidente cambie el valor que esperamos y que provoque efectos secundarios como errores en nuestro código.

## Novedades en Ruby on Rails 5

Vamos a hacer un pequeño recuento de los cambios de Rails 5 con respecto a la versión anterior.

## Nueva sintaxis de Ruby on Rails

Uno de los cambios de Rails es la versión mínima de Ruby que es compatible, como ya mencionamos anteriormente.

Con este cambio, fue posible limpiar el código de Rails para mantener compatibilidad con otras versiones de Ruby, pero también abrió la posibilidad de usar características del lenguaje que sólo estaban presentes por ejemplo los *Keyword arguments*.

```
# Sin keyword arguments
```

```
def init_user_with(name, email)  
  ... # hacemos la inicialización  
end
```

```
> init_user_with("Pedro Ruíz", "pedro@mail.com")
```

```
# Con keyword arguments
```

```
def init_user_with(name:, email:)  
  ... # hacemos la inicialización  
end
```

```
> init_user_with name: "Pedro Ruiz", email: "pedro@mail.com"
```

```
# O podemos cambiar el orden de los parámetros
```

```
> init_user_with email: "pedro@mail.com", name: "Pedro Ruiz"
```

Cómo vemos en el primer ejemplo sin *Keyword arguments* pasamos los parámetros sin indicar cuál parámetro es cuál ya que tenemos que pasarlos en el orden correcto.

En el segundo ejemplo con *Keyword arguments* podemos pasar los parámetros en cualquier orden, debido a que nombramos explícitamente que valor corresponde a cuál parámetro.

APIs de Rails están empezando lentamente a tomar. *Keyword arguments* no es el único cambio y tampoco va a ser el último, poco a poco vamos aver como el API de Rails adopta más cosas de la nueva sintaxis de Ruby.

## Limpieza de código viejo

Con cada nueva versión siempre hay anuncios de funcionalidad que eventualmente va a ser eliminada de futuras versiones de Rails, es decir cada nueva versión de Ruby on Rails no sólo trae nueva funcionalidad, pero también elimina funcionalidad que ya no es muy popular o que fue reemplazada por algo más nuevo.

El eliminar este código viejo ayuda a minimizar el código que hay que mantener, pero también a *empujar* a los desarrolladores a hacer uso de las nueva y mejor funcionalidad.

## Mejoras en el rendimiento

Debido a la decisión del equipo de Rails 5 de utilizar y soportar versiones modernas de Ruby, por omisión heredan las ventajas de menor uso de memoria y rendimiento en las aplicaciones, pero el equipo estuvo ocupado limpiando código en diferentes áreas para tratar de que el código fuera lo más eficiente posible en el uso de recursos.

Hay casos en donde los cambios en áreas específicas de Rails 5 hay incrementos de velocidad hasta en un 30%.

## ActionCable

[ActionCable](#) es una nueva funcionalidad en Rails 5 que permite que nuestras aplicaciones se comuniquen mediante [WebSockets](#).

[WebSockets](#) permite que las aplicaciones se comuniquen en tiempo real, enviando eventos y mensajes a través canales persistentes entre el navegador y el servidor.

Rails proporciona todo lo necesario del lado del servidor y librerías de Javascript para el cliente que permiten desarrollar aplicaciones de una manera sencilla.

Debido a esta nueva funcionalidad Rails finalmente desechó el uso del servidor web [Webrick](#) en favor del servidor [Puma](#), el cuál permite la comunicación vía [WebSockets](#).

## El modo API

Hoy en día no todas las aplicaciones Web requiere de una interfase gráfica, hay ocasiones que únicamente requerimos que nuestra aplicación exponga una interfase API para ser consumida por aplicaciones móviles, por una aplicación de javascript o por otra aplicación.

Antes de la versión 5 de Rails existía una gema del mismo equipo de desarrollo de Rail que configuraba nuestra aplicación para funcionar en modo API, es decir deshabilitaba toda la funcionalidad de Rails para el manejo de sesiones, [rendereo](#) vistas y demás herramientas del marco de trabajo que no son necesarias para trabajar en modo API.

El objetivo de reducir la configuración es evitar que el servidor de Rails consuma memoria con funcionalidad que no se está utilizando.

Con la versión 5 de Rails ya no es necesario instalar una gema adicional para activar el modo de API, ahora al momento de crear nuestra aplicación simplemente tenemos que agregar la opción [--api](#) para que nuestra aplicación funcione en modo API.

```
rails new mi_nueva_app --api
```

El cambio más visible que vamos a encontrar en nuestra nueva aplicación es que el controlador `ApplicationController` ya no hereda de `ActionController::Base` si no que hereda `ActionController::API`.

## Soporte para consultas con OR en ActiveRecord

Con la nueva versión, `ActiveRecord` ya tiene soporte para poder realizar consultas a la base de datos usando `OR`. La inclusión de la cláusula `OR` ha sido tema de discusión durante varias versiones de Rails, anteriormente no se había aceptado por las implicaciones de `rendimiento` al ejecutar consultas en la base de datos. Es muy sencillo escribir una consulta con `OR` y que afecte de manera negativa a la base de datos, por lo que hay que utilizarse con precaución.

```
User.where(email: 'user1@mail.com').or(User.where(email: 'user2@mail.com'))
```

## Actualizando nuestra aplicación a Rails 5

Ya que conocemos cuáles son las principales características de Ruby 2.3 y Ruby on Rails 5, vamos a proceder a actualizar nuestra aplicación de ejemplo, pero vamos a tener en cuenta que lo aquí explicado puede aplicarse a otras aplicaciones de Rails.

Lo ideal antes de actualizar una aplicación, es asegurarnos que nuestro conjunto de pruebas automáticas se ejecuten y terminen sin problemas. Esto nos da la tranquilidad de que si hay algo que cambió en la nueva versión de Rails únicamente las pruebas que fallen una vez actualizada nuestra aplicación es donde tendríamos que enfocar nuestros esfuerzos para entender y corregir lo que haya cambiado en su comportamiento.

Si bien es posible actualizar nuestras aplicaciones aún si no cuentan con pruebas o con pruebas que fallen, no es lo ideal.

### *Qué pasa si quiero actualizar una aplicación de Rails en una versión anterior a Ruby on Rails 4.2?*

Además de tener nuestro conjunto de pruebas en estado verde, es decir que todas las pruebas pasen, será necesario revisar los cambios entre las diferentes versiones de Ruby on Rails para conocer todos los posibles puntos de falla al momento de actualizar.

En estos casos la actualización va a ser más compleja de lo que se mostrará en este capítulo.

Idealmente es necesario tratar de mantener nuestras aplicaciones actualizadas, no sólo por la nueva funcionalidad, pero para asegurarnos que obtenemos parches de seguridad y mejoras en el rendimiento de las aplicaciones.

## *Actualizando nuestra aplicación*

Lo primero que vamos a hacer es crear una nueva rama en *Git* para trabajar nuestros cambios de manera aislada.

```
$ git branch rails-upgrade
```

```
$ git checkout rails-upgrade
```

Nos aseguramos que nuestro conjunto de pruebas está verde.

```
$ rake test
```

```
Run options: --seed 47328
```

```
# Running:
```

```
.....
```

```
Finished in 0.415098s, 26.4998 runs/s, 62.6358 assertions/s.
```

```
11 runs, 26 assertions, 0 failures, 0 errors, 0 skips
```

Perfecto! Estamos listos para iniciar la actualización.

Vamos a abrir el archivo `Gemfile` y vamos actualizar la versión de la gema de `Rails` de la versión `4.2.6` a la versión `5.0.1` que es la más actual a este momento. Hay que notar que algunas otras gemas tiene versiones explícitas, las cuales vamos a necesitar modificar en un paso más adelante. Después del cambio nuestro archivo `Gemfile` se ve como sigue.

```
source 'https://rubygems.org'
```

```
gem 'bcrypt'
```

```
gem 'coffee-rails', '~> 4.1.0'
```

```
gem 'jquery-rails'
```

```
gem 'turbolinks', github: 'turbolinks/turbolinks-rails'
```

```
gem 'pg'
```

```
gem 'pretty_formatter'
```

```
gem 'puma'
```

```
gem 'rails', '5.0.1'

gem 'sass-rails', '~> 5.0'

gem 'uglifier', '>= 1.3.0'

group :development do

  gem 'pry-rails', require: false

  gem 'spring'

  gem 'web-console', '~> 2.0'

end

group :development, :test do

  gem 'brakeman', require: false

  gem 'bundler-audit', require: false

  gem 'byebug'
```

```
gem 'capybara-webkit'

gem 'minitest-rails'

gem 'minitest-rails-capybara'

gem 'launchy'

gem 'pronto'

gem 'pronto-brakeman', require: false

gem 'pronto-reek', require: false

gem 'pronto-rubocop', require: false

gem 'reek', require: false

gem 'rubocop', require: false

end
```

Guardamos nuestro archivo y ejecutamos el comando para actualizar nuestras gemas.

```
$ bundle update rails
```

```
Fetching gem metadata from https://rubygems.org/.....
```

```
Fetching version metadata from https://rubygems.org/..
```

```
Fetching dependency metadata from https://rubygems.org/.
```

```
Resolving dependencies...
```

```
Bundler could not find compatible versions for gem "railties":
```

```
In Gemfile:
```

```
minitest-rails-capybara was resolved to 2.1.1, which depends on
```

```
minitest-rails (~> 2.1) was resolved to 2.2.0, which depends on
```

```
railties (~> 4.1)
```

```
rails (= 5.0.1) was resolved to 5.0.1, which depends on
```

```
railties (= 5.0.1)
```

Encontramos el primer error. En este caso nos indica que la gema `minitest-rails-capybara` tiene una dependencia a `minitest-rails`, la cuál tiene dependencia a `railties`, una gema de Rails, en la versión `4.1`. Este error es un común cuando actualizamos versiones de Rails en nuestras aplicaciones. Generalmente otras gemas que usamos en nuestros proyectos también tienen que ser actualizadas para que sean compatibles con la nueva versión de Rails.

En nuestro caso esto no representa gran problema, ya que para este momento todas las gemas de terceros deben de estar actualizadas.

Para las gemas que nos impiden actualizar Rails vamos a observar que el archivo `Gemfile` no especifica una versión particular de las mismas, así que para indicarle a `Bundler` que deseamos nuevas versiones de esas gemas ejecutamos el comando para actualizar con el nombre de las mismas.

```
$ bundle update rails minitest-rails minitest-rails-capybara
```

```
Fetching gem metadata from https://rubygems.org/.....
```

```
Fetching version metadata from https://rubygems.org/..
```

```
Fetching dependency metadata from https://rubygems.org/.
```

```
Resolving dependencies...
```

```
Bundler could not find compatible versions for gem "railties":
```

```
In Gemfile:
```

```
minitest-rails was resolved to 3.0.0, which depends on
```

```
    railties (~> 5.0)
```

```
rails (= 5.0.1) was resolved to 5.0.1, which depends on
```

```
    railties (= 5.0.1)
```

```
sass-rails (~> 5.0) was resolved to 5.0.4, which depends on
```

```
railties (< 5.0, >= 4.0.0)
```

Por el resultado del comando parece que en lugar de resolver un problema creamos 2 más. Vamos ahora a enfocarnos en la gema **sass-rails**. En nuestro archivo **Gemfile** vemos que tiene especificada una versión **~> 5.0** y esa versión entra en conflicto con la nueva versión de Rails.

Para conocer que versión es la que tenemos que especificar, podemos navegar a [RubyGems](#) buscar la gema para ver el historial de versiones. Ahí vemos que la versión **5.0.6** nos permite trabajar con la gema **railties** que tenga una versión **< 6 y >= 4.0.0**. Vamos a cambiar la versión de **sass-rails** a **5.0.6** y reintentar el comando de actualización.

```
$ bundle update rails minitest-rails minitest-rails-capybara sass-rails
```

Vemos que al final el comando fue un éxito, sin embargo este es sólo el primer paso.

Con cada cambio de versión en Rails se agregan o se eliminan opciones de configuración que encontramos en los diferentes archivos localizados en el folder **config**. Para actualizarlos vamos a ejecutar el siguiente comando de Rails.

```
$ rake app:update
```

Al momento de ejecutarse nos indicará los archivos de configuración que va a actualizar, el único problema con esto es que no puede mezclar cambios realizados en los archivos anteriores, es decir, los puede sobre escribir o los puede ignorar.

El primer archivo que nos indica es el archivo de **config/boot.rb**, este archivo se encarga de cargar las dependencias de **Bundler** en nuestra aplicación. Si presionamos la letra **d** nos va a mostrar cuál es la diferencia entre los dos archivos.

```
-ENV['BUNDLE_GEMFILE'] ||= File.expand_path '../../../../../Gemfile', __FILE__  
  
+ENV['BUNDLE_GEMFILE'] ||= File.expand_path '../Gemfile', __dir__  
  
require 'bundler/setup' # Set up gems listed in the Gemfile.
```

En este caso `-` significa que es la línea original y `+` nos indica que es la nueva línea. Para `config/boot.rb` no hay problema con sobre escribirlo con la nueva versión, por lo tanto presionamos `Y`.

El siguiente archivo que trata de actualizar es `config/routes.rb`, este archivo en ningún caso deseamos que se actualice, por lo que presionamos `n`. Como ejercicio podemos presionar `d` para conocer las diferencias.

Después trata de actualizar el archivo `config/application.rb`, este archivo tiene cambios que realizamos anteriormente, presionando la letra `d` podemos ver que entre otras líneas trata de modificar la configuración de internacionalización de Rails, la cual tenemos en español.

```
config.i18n.default_locale = :es
```

Vamos a optar por indicarle que sí queremos reemplazar el archivo, pero una vez modificado, con nuestro editor agregamos la línea de configuración nuevamente.

Ahora es el turno para el archivo `config/environment.rb`, del cual al presionar la letra `d` vemos que no continúa ninguna configuración personalizada, por lo que podemos proceder a presionar la `Y` para que se actualice.

`config/secrets.yml` contiene información secreta de nuestra aplicación, por ejemplo llaves de cifrado, llaves a servicios de terceros, etc. En nuestro caso no hemos agregado nada nuevo a `config/secrets.yml` pero en la mayoría de los casos deseamos mantener el archivo original, así que presionamos `n`.

Ahora sigue el turno de los archivo `config/environments/development.rb`, `config/environments/test.rb` y `config/environments/production.rb`. Dependiendo de las necesidades de nuestra aplicación quizás hayamos modificado alguna configuración para los tres posibles ambientes de ejecución de Rails. En el caso de estos archivos lo mejor es presionar la letra `d`, copiar las líneas de configuración que hayamos introducido, actualizarlos presionando la letra `Y` y acto seguido con el editor poner de regreso nuestra configuración que haya sido reemplazada.

El archivo `config/initializers/cookies_serializer.rb` tiene cambios, que para el caso de nuestra aplicación no tienen ningún impacto. Este archivo indica cómo es que las `cookies` van a ser guardadas en el navegador. Hay 3 opciones: `:marshal`, `:json` e `:hybrid`.

Si nuestra aplicación original la creamos con Rails 4.1 o una versión más nueva, no hay ningún problema, la opción puede `:json`, si la creamos con una versión anterior a 4.1, entonces por omisión Rails guarda las `cookies` en formato `:marshal`.

El modificar el formato a `:json` implica que automática invalidaremos todas las `cookies` de nuestros clientes, lo cual puede ser un problema. Por lo tanto la opción que tenemos en este caso es

usar el modo `:hybrid`, el cuál va a migrar las cookies de `:marshal` a `:json` que es el formato por omisión en Rails 5.

Pero como en nuestro caso la aplicación fue creada con la versión 4.2 de Rails, podemos simplemente indicar que sí queremos reemplazar el archivo con `Y`. Si ustedes están migrando desde una versión anterior a 4.1, pueden también indicar que sí se reemplace, pero posteriormente modificarlo para que tenga la opción `:marshal`.

Toca el turno a `config/initializers/wrap_parameters.rb`, presionamos `d` para revisarlos los cambios y podemos observar que son de formato en el archivo, así como la eliminación de un `if`. Procedemos a presionar `Y` para reemplazarlo.

Para los archivos `bin/rails`, `bin/rake` y `bin/setup` podemos indicar `Y` para que sean actualizados.

Al finalizar la actualización encontramos que el proceso actualizó los siguientes archivos:

`bin/rails`

`bin/rake`

`bin/setup`

`config/application.rb`

`config/boot.rb`

`config/environment.rb`

`config/environments/development.rb`

`config/environments/production.rb`

`config/environments/test.rb`

```
config/initializers/cookies_serializer.rb
```

```
config/initializers/wrap_parameters.rb
```

Y se crearon los siguientes archivos:

```
bin/update
```

```
config/cable.yml
```

```
config/initializers/application_controller_renderer.rb
```

```
config/initializers/new_framework_defaults.rb
```

```
config/puma.rb
```

```
config/spring.rb
```

Este es un buen momento para nuevamente ver que nuestro conjunto de pruebas aún sigue en verde, es decir ninguna falla. Hasta antes de la actualización siempre ejecutábamos nuestras pruebas con `rake test`, pero ahora es Rails 5 es posible hacer lo mismo ejecutando `rails test`.

Este es uno de los cambios, y que no se mencionó al inicio del capítulo, de Rails 5.

```
$ rails test
```

```
...
```

```
DEPRECATION WARNING: Using positional arguments in functional tests has been deprecated,
```

in favor of keyword arguments, and will be removed in Rails 5.1.

Deprecated style:

```
get :show, { id: 1 }, nil, { notice: "This is a flash message" }
```

New keyword style:

```
get :show, params: { id: 1 }, flash: { notice: "This is a flash message" },
```

```
session: nil # Can safely be omitted.
```

```
(called from block in <class:RegistrationControllerTest> at /Users/marioch/Development/onepx/test/controllers/registration_controller_test.rb:12)
```

.

Finished in 0.441981s, 24.8879 runs/s, 61.0886 assertions/s.

Nuestra prueba concluyen exitosamente, pero ahora tenemos unos mensajes de [DEPRECATION](#) [WARNING](#), nos dice que tenemos que utilizar en nuestras pruebas par nueva sintaxis que usa [keyword arguments](#), mencionados al inicio del capítulo. Por el momento vamos a ignorar los mensajes y vamos a proceder al siguiente paso de la actualización.

En el capítulo anterior incluimos a nuestra aplicación herramientas que nos permiten revisar si tenemos alguna vulnerabilidad de seguridad en las gemas de nuestra aplicación o en nuestro código. Con el siguiente comando podemos ejecutar *Brakeman* y *Ruby Advisory*.

```
$ rails quality:check:security
```

```
...
```

```
Insecure Source URI found: git://github.com/turbolinks/turbolinks-rails.git
```

```
Vulnerabilities found!
```

Vemos que el único reporte es sobre *Turbolinks* ya que estamos usando la versión de *Github* en lugar de la gema. Hay que actualizar nuestro archivo **Gemfile** para que use la gema. Pero también vamos actualizar la versión de la gema **coffee-rails** para que utilice la más reciente:

[4.2.1.](#)

De manera que el nuevo **Gemfile** queda así.

```
source 'https://rubygems.org'
```

```
gem 'bcrypt'
```

```
gem 'coffee-rails', '~> 4.2.1'
```

```
gem 'jquery-rails'
```

```
gem 'turbolinks'
```

```
gem 'pg'
```

```
gem 'pretty_formatter'

gem 'puma'

gem 'rails', '5.0.1'

gem 'sass-rails', '~> 5.0.6'

gem 'uglifier', '>= 1.3.0'

group :development do

  gem 'pry-rails', require: false

  gem 'spring'

  gem 'web-console', '~> 2.0'

end

group :development, :test do

  gem 'brakeman', require: false
```

```
gem 'bundler-audit', require: false

gem 'byebug'

gem 'capybara-webkit'

gem 'minitest-rails'

gem 'minitest-rails-capybara'

gem 'launchy'

gem 'pronto'

gem 'pronto-brakeman', require: false

gem 'pronto-reek', require: false

gem 'pronto-rubocop', require: false

gem 'reek', require: false

gem 'rubocop', require: false

end
```

Ahora ejecutamos bundler en modo *update* para que actualice las gemas de las que cambiamos versión explícita y de paso actualice todas las gemas de nuestra aplicación a su versión más reciente.

```
$ bundle update
```

Durante la actualización vemos que fueron varias gemas las actualizadas, lo sabemos porque a la izquierda del nombre de la gema aparece la palabra *Installing*.

Una vez más ejecutamos nuestras pruebas para ver en qué estado se encuentran.

```
$ rails test
```

Seguimos en verde, ninguna prueba falla, pero aún tenemos el *warning*. Vamos a corregirlo. El *warning* nos da la solución, indica que el problema está en el archivo `test/controllers/ registration_controller_test.rb` y para cada prueba reportada nos da el número de línea. También nos indica que hay que usar las llaves `params:` y `flash:` según corresponda para los argumentos de la llamada y que podemos eliminar el `nil` que pasamos como argumento, si es que pasamos alguno.

Después del cambio, el archivo `test/controllers/ registration_controller_test.rb` queda como sigue.

```
# frozen_string_literal: true

require "test_helper"

class RegistrationControllerTest < ActionController::TestCase

  test "GET /new" do
    get :new
  end
end
```

```
    assert_response :success

end

test "POST /create without params" do

  post :create, params: { user: { email: "" } }

  assert_response :success

end

test "POST /create with valid params" do

  post :create, params: { user: { email: "newuser@mail.com",
                                password: "pa$$word",
                                password_confirmation: "pa$$word" } }


```

```
    assert_redirected_to root_path

end
```

```
end
```

Ejecutamos nuestra pruebas nuevamente y ahora ya no tenemos ningún *warning*.

```
$ rails test
```

```
Run options: --seed 29619
```

```
# Running:
```

```
.....
```

```
Finished in 0.302437s, 36.3712 runs/s, 89.2748 assertions/s.
```

```
11 runs, 27 assertions, 0 failures, 0 errors, 0 skips
```

Oficialmente nuestra aplicación ha sido actualizada a Rails 5!. Solamente nos resta hacer el *commit* de los cambios y listo.

```
$ git add .  
  
$ git commit -am "Código del capítulo 6"  
  
$ git push --set-upstream origin rails-upgrade
```

## Conclusiones

El proceso para actualizar una aplicación de Rails a la versión más actual puede variar en complejidad, en el caso de la aplicación que estamos construyendo en el libro fue muy sencillo, pero en una aplicación que ya está en producción puede llevar un poco más de tiempo, aunque la metodología es la misma: Verificar que las pruebas estén verdes, ir haciendo los cambios poco a poco y volver a ejecutar las pruebas, repetir hasta que completar.

Es importante incluir en nuestro proceso de desarrollo tiempo para que una vez al mes, por lo menos, actualicemos las gemas de nuestra aplicación, así como versiones intermedias de Rails.

El mantener actualizada nuestra aplicación, en la mayoría de los casos, implica solamente actualizar el numero de versión y listo. Son las excepciones donde tenemos que hacer ajustes a nuestro código después de actualizar una gema o una versión intermedia de Rails; pero afortunadamente, si tenemos pruebas, podemos detectar esos lugares en donde nuestro código falla después de la actualización y corregirlo.

El actualizar las versiones de Rails y las gemas no sólo nos asegura que tenemos las versiones más recientes, también nos asegura que actualizaciones de seguridad y de rendimiento han sido aplicadas.

# Capítulo 7 - Trabajando con recursos Web en Rails

Hasta este momento hemos trabajado en escribir funcionalidad para nuestra aplicación, nos permite crear una cuenta de usuario, pero no hemos realizado trabajo alguno con la presentación de nuestra aplicación, fuera del HTML mínimo necesario.

En este capítulo vamos a trabajar en hacer que nuestra aplicación tenga presentación visual interesante para nuestros usuarios, para eso vamos a conocer el [Asset Pipeline](#) de Rails, el cual el la herramienta en Ruby on Rails nos permite trabajar con hojas de estilo, archivos de Javascript, fuentes tipográficas e imágenes y otros recursos.

El capítulo no se basará en enseñar a programar hojas de estilo o javascript, el objetivo será entender cómo trabajar con ellos de forma correcta en Rails.

## Asset Pipeline

Una aplicación web está compuesta por código al que llamamos [Backend](#) y que se ejecuta como procesos del sistema operativo. Este código contiene la lógica no visible para el usuario y hace que nuestra aplicación funcione.

Pero una aplicación web también se compone de una parte visual que sirve de interfase para que el usuario puede hacer uso de nuestra aplicación, este código lo conocemos como [Frontend](#).

El [Frontend](#) generalmente está compuesto de una serie de archivos de hojas de estilo, Javascript, imágenes, y fuentes tipográficas. Dependiendo de como estructuremos esta parte de nuestra aplicación, puede ser por lo menos un archivo de cada tipo o bien pueden ser varios archivos de cada tipo que trabajen en conjunto.

El último caso es el más común, multiples archivos de cada tipo, que contienen diferentes responsabilidades en nuestro [Frontend](#). Adicionalmente podemos requerir de marcos de trabajo de hojas de estilo como: [Bootstrap](#), [PureCSS](#) o [Susy](#). En el caso de Javascript también puede surgir esa necesidad con herramientas como: [jQuery](#), [React](#) o [AngularJS](#).

Ruby on Rails a través del [Asset Pipeline](#) nos permite organizar esos archivos, además de realizar post-procesamiento automático de los mismos, por ejemplo en el caso de las hojas de estilo, las podemos escribir en sintaxis [SASS](#) y Rails automáticamente generará los archivos CSS correspon-

dientes; lo mismo sucede con Javascript, el cuál podemos escribir con sintaxis de [ES6](#) y obtener archivos de Javascript que los navegadores entiendan.

Para el proceso de desarrollo tiene mucho sentido tener múltiples archivos de hojas de estilo y Javascript, pero una vez que ponemos nuestra aplicación en producción es necesario optimizar el acceso a esos archivos, ya que sería poco eficiente para un navegador pedir al servidor web 5 o 10 o 20 archivos, para este caso lo que el [Asset Pipeline](#) hace, es primeramente concatenar esos archivos, es decir ordena los archivos de acuerdo a como se requieran y luego los une para generar un sólo archivo de hojas de estilo o un sólo archivo de Javascript, según sea el caso.

Una vez que los une los comprime eliminando comentarios y espacios innecesarios para el navegador, finalmente a los archivos, en el nombre, les agrega una huella digital, la cual permite a los navegadores, entre otras cosas, mantener una copia de esos archivos, para que en caso de volvamos a visitar el mismo sitio web no descargue los archivos de nuevo, a menos de que hayan cambiado, haciendo así que el sitio web responda más rápido.

La huella digital se ve como se muestra en el siguiente ejemplo, donde el nombre original del archivo es [global.css](#).

`global-908e25f4bf641868d8683022a5b62f54.css`

Para las imágenes y fuentes tipográficas, el [Asset Pipeline](#) únicamente agrega la huella digital al nombre.

Ahora que tenemos una idea general del manejo de recursos web con el [Asset Pipeline](#), vamos a ponerla en práctica en nuestra aplicación.

## Trabajando con recursos web

Hasta este momento en nuestra aplicación la única área que hemos completado es la parte del registro de usuarios, pero si iniciamos nuestro servidor web con el comando `rails s` y luego apuntamos el navegador a `http://localhost:3000/registro` vemos que la aplicación no se ve nada agradable a la vista.

## Crear una cuenta

Correo electrónico

Contraseña

Confirmar contraseña

Vamos a corregir esto durante este capítulo, aunque la explicación del mismo se enfocará en como trabajar con Rails con los diferentes archivos de recursos web, sin embargo no se enfocará a explicar el uso y sintaxis de CSS y Javascript.

## *Agregando un poco de estilo*

Para el estilo de nuestra aplicación vamos a hacer uso de [PureCSS](#) como base y a partir de él vamos a ir agregando los estilos adicionales en archivos de [SASS](#).

Para instalar [PureCSS](#) lo vamos a descargar desde la url <https://unpkg.com/purecss@0.6.2/build/pure-min.css>, pero antes hay que decidir donde vamos a colocar el archivo, de forma que el [Asset Pipeline](#) pueda hacer uso de él.

Todas las aplicaciones de Rails cuentan una carpeta [app/assets](#), la cuál es el lugar indicado para colocar los recursos web que nuestra aplicación va a necesitar. Dentro de la carpeta vamos a encontrar que existen otras 3 carpetas llamadas: [images](#), [javascripts](#) y [stylesheets](#). En cada una de esas carpetas colocamos imágenes, archivos de Javascript y hojas de estilo respectivamente.

Pero de igual forma en todas las aplicaciones de Rails encontraremos la carpeta [vendor/assets](#), la cuál tiene las carpetas [javascripts](#) y [stylesheets](#); entonces cuál es la carpeta indicada para nuestros recursos?

No hay una regla estricta en Rails en cuál de las 2 carpetas utilizar, pero generalmente la idea es que la carpeta [app/assets](#) es utilizada para poner ahí los recursos web que son [exclusivos](#) de la aplicación que estamos desarrollando. [vendor/assets](#) es utilizada para colocar ahí recursos de terceros que vamos a incluir y necesitar en nuestra aplicación.

En el caso de [PureCSS](#) lo vamos a colocar en la carpeta de [vendor/assets/stylesheets](#), la forma en como lo vamos a hacer es la siguiente desde la terminal, hay que asegurarse que estamos dentro de la carpeta indicada.

```
vendor/assets/stylesheets $ curl -0 https://unpkg.com/purecss@0.6.2/build/pure-min.css
```

Una vez que se descargó el archivo, es necesario hacer referencia de él para que el *Asset Pipeline* lo pueda utilizar. En una aplicación de Rails la manera en como hacemos referencia a archivos de hojas de estilos es mediante el archivo `assets/stylesheets/application.css`.

Al abrir ese archivo en el editor vemos lo que parece ser un comentario que describe el uso del mismo.

```
/*
 * This is a manifest file that'll be compiled into application.css, which will
 * include all the files

 * listed below.

 *

 * Any CSS and SCSS file within this directory, lib/assets/stylesheets, vendor/
 * assets/stylesheets,
 *
 * or any plugin's vendor/assets/stylesheets directory can be referenced here us-
 * ing a relative path.

 *

 * You're free to add application-wide styles to this file and they'll appear at
 * the bottom of the

 *
 * compiled file so the styles you add here take precedence over styles defined
 * in any styles
```

```
* defined in the other CSS/SCSS files in this directory. It is generally better
to create a new

* file per style scope.

*

*= require_tree .

*= require_self

*/
```

Pero las 2 últimas líneas que son directivas para el *Asset Pipeline*. `require_tree` indica que queremos incluir todos los archivos que se encuentran en la misma carpeta donde está el archivo. `require_self` indica que queremos incluir como referencia al archivo mismo.

Si lo vemos detenidamente, `application.css` es un archivo de manifiesto donde declaramos que archivos de hojas de estilo necesitamos en nuestra aplicación.

Vamos a eliminar la línea `*= require_tree .` y la línea `*= require_self` la modificamos para se lea `*= require pure-min`. Con este cambio indicamos en nuestro manifiesto que solamente requerimos el archivo que acabamos de descargar. Como vemos no fue necesario indicar toda la ruta al archivo ni incluir la extensión.

Guardamos los cambios de `application.css` y recargamos la página del navegador que apunta a nuestra aplicación.

Con este simple cambio podemos observar que la tipografía utilizada por omisión ha sido cambiada, es el efecto de *PureCSS*.

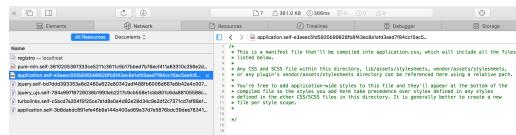
## Crear una cuenta

Correo electrónico

Contraseña

Confirmar contraseña

Además si abrimos las herramientas de desarrollo en nuestro navegador, abrimos la pestaña de Red y recargamos la página, vamos a ver que 2 archivos CSS fueron cargados: pure-min.self y application.self. Podemos ver como los 2 archivos tienen su huella digital en el nombre, pero no están comprimidos ni concatenados como un sólo archivo, contrario a lo que dije anteriormente.



El motivo es que estamos ejecutando nuestra aplicación en modo de *Development* y en ese caso Rails evita la compresión y concatenación para que nos sea más sencillo trabajara con esos archivos.

Ya que confirmamos que *PureCSS* si es utilizado por el *Asset Pipeline*, vamos a agregar un poco de estilo a nuestro formulario. Para referencias a las clases de CSS que se van a utilizar, por favor visitar el sitio de [PureCSS](#).

En el editor abrimos el archivo `app/views/registration/new.html.erb` y cambiamos:

```
<%= form_for @user, url: registration_path do |form| %>
```

Por

```
<%= form_for @user, url: registration_path, html: { class: "pure-form pure-form-stacked" } do |form| %>
```

Después cambiamos:

```
<%= form.submit %>
```

Por:

```
<%= form.submit class: "pure-button pure-button-primary" %>
```

Finalmente es necesario que el contenido del formulario se encuentre dentro de una etiqueta de HTML *fieldset*, para que archivo final quede de la siguiente forma:

```
#app/views/registration/new.html.erb
```

```
<h3><%= t('.title') %></h3>

<%= form_for @user, url: registration_path, html: { class: "pure-form pure-form-stacked" } do |form| %>

<fieldset>

<%-- if @user.errors.any? %>

<div class='form-error-message'><%= t('form.error.message') %></div>

<% end %>

<div class='field <%= set_error_class(form.object, :email) %>'>

<%= form.label :email %>

<%= form.text_field :email %>

</div>
```

```

<div class='field <%= set_error_class(form.object, :password) %>'>

  <%= form.label :password %>

  <%= form.password_field :password %>

</div>

<div class='field <%= set_error_class(form.object, :password_confirmation) %>'>

  <%= form.label :password_confirmation %>

  <%= form.password_field :password_confirmation %>

</div>

<div class='action'>

  <%= form.submit class: "pure-button pure-button-primary" %>

</div>

</fieldset>

<% end %>

```

Guardamos los cambios del archivo y refrescamos la página del navegador y vemos que la forma ya se ve muy diferente.

## Crear una cuenta

Correo electrónico

Contraseña

Confirmar contraseña

**Crear cuenta**

Lo único que cambiamos fue la adición del parámetro `class:` en la declaración de la forma y del botón, los valores utilizados vienen de la documentación de [PureCSS](#).

Nuestro formulario ya se ve más presentable pero definitivamente requiere de un poco más de trabajo. Para realizar los ajustes necesarios vamos a crear una

nueva carpeta en `app/assets/stylesheets`, el nombre de la carpeta va a ser `modules`.

De manera que la estructura de la capeta `app/assets` se vea de la siguiente forma:

|  
|   └ images

|  
|   └ javascripts

|   |   └ application.js

|   └ stylesheets

|   └ application.css

## └ modules

Ahora vamos a crear 2 archivos, `_form.scss` y `base.scss`. El primero dentro de la carpeta `modules` y el segundo en `stylesheets`. Estos nuevos archivos tienen una extensión peculiar `scss`, lo que le indica al `Asset Pipeline` que esos archivos de hojas de estilo van a contener sintaxis de `SASS`, por lo que va a ser necesario de Rails hacer un post-proceso para convertirlos a sintaxis de CSS antes de mandarlos al navegador.

En la primera línea de `base.scss` agregamos la instrucción para hacer referencia a el archivo `_form.scss`.

```
# app/assets/stylesheets/base.scss
```

```
@import "modules/form";
```

Luego editamos el manifiesto `application.css` para agregar un segundo `require` al archivo `base.scss`.

```
# app/assets/stylesheets/application.css
```

```
...
```

```
*= require pure-min
```

```
*= require base
```

```
*/
```

El archivo `base.scss` va a contener CSS general para nuestra aplicación, pero también va importar archivos parciales como `modules/_form.scss` que para este caso en particular contendrá únicamente CSS que afecte a los formularios.

Mantener este tipo de estructura y organización en los archivos de hojas de estilos hace que su manejo sea más simple de administrar.

Para los ajustes del formulario hay que copiar el siguiente código a `_form.scss`.

```
# app/assets/stylesheets/modules/_form.scss

.pure-form-stacked {

    box-sizing: border-box;

    color: $foreground-color;

    margin: 5em auto;

    padding: 40px;

    width: 400px;

}

h1 {

    color: $header-color;

    font-weight: normal;

    margin-bottom: 1.5em;

    text-align: center;

}
```

```
.field {  
  margin-bottom: 1em;  
}  
  
label {  
  font-weight: lighter;  
}  
  
input[type=text],  
input[type=password] {  
  border: 1px solid $border-color;  
  border-radius: 2px;  
  box-shadow: none;  
}
```

```
width: 320px;

}

.action {

margin-top: 2em;

}

input[type=submit] {

border-radius: 2px;

height: 40px;

width: 320px;

}

}
```

```
.form-error-message {  
  color: $error-color;  
  
  text-align: center;  
  
  margin-bottom: 1.5em;  
}  
  
.error-field {  
  
  label {  
  
    color: $error-color;  
  
  }  
  
  input {  
  
    border-color: $error-color !important;  
  }  
}
```

```
}
```

El archivo `base.scss` también se modificó para declarar variables de los colores a usar en la hoja de estilos, la definición de variables es una de las funcionalidades que provee la sintaxis de [SASS](#).

```
# app/assets/stylesheets/base.scss

$border-color: rgba(185, 193, 199, 0.5);

$foreground-color: #71767a;

$header-color: #525558;

$error-color: #ff0000;

@import "modules/form";
```

Finalmente es necesario hacer algunos ajustes al archivo `app/views/registration/new.html.erb`, el cuál es la plantilla de HTML para el formulario. Los cambios consistieron en mover el encabezado `h3` dentro de la función de la forma y a los `text_field` y `password_field` se les colocó un máximo de caracteres que pueden aceptar.

```
# app/views/registration/new.html.erb

<%= form_for @user, url: registration_path, html: { class: "pure-form pure-form-stacked" } do |form| %>

<h1><%= t('.title') %></h1>
```

```
<fieldset>

<%- if @user.errors.any? %>

  <div class='form-error-message'><%= t('form.error.message') %></div>

<% end %>

<div class='field <%= set_error_class(form.object, :email) %>'>

  <%= form.label :email %>

  <%= form.text_field :email, maxlength: 45 %>

</div>

<div class='field <%= set_error_class(form.object, :password) %>'>

  <%= form.label :password %>

  <%= form.password_field :password, maxlength: 20 %>

</div>

<div class='field <%= set_error_class(form.object, :password_confirmation) %>'>

  <%= form.label :password_confirmation %>
```

```
<%= form.password_field :password_confirmation, maxlength: 20 %>

</div>

<div class='action'>

<%= form.submit class: "pure-button pure-button-primary" %>

</div>

</fieldset>

<% end %>
```

De igual manera los mensajes del encabezado y el botón del formulario se modificaron.

```
# config/locales/es.views.yml
```

```
es:
```

```
registration:
```

```
new:
```

```
  title: 'Únete a onepx'
```

```
helpers:
```

**submit:**

**user:**

**create: 'Unirse'**

**form:**

**error:**

**message: 'Por favor corrija los siguientes errores'**

Una vez que tenemos todos los cambios, podemos refrescar la pantalla del navegador y vamos a ver cómo cambio el formulario.

## Únete a onepx

Correo electrónico

Contraseña

Confirmar contraseña

**Unirse**

Inclusive si presionamos el botón de [Unirse](#) vamos a ver que el formulario nos muestra los campos que tienen algún error.

# Únete a onepx

Por favor corrija los siguientes errores

Correo electrónico

Contraseña

Confirmar contraseña

Unirse

Nuestro formulario se ve muy bien, por lo menos muy diferente a como estaba al inicio de este capítulo. Pero vamos a realizar unos ajustes más.

En el capítulo 4 hablamos de *layouts*, los cuales son plantillas maestras dentro de Rails. Para nuestra aplicación ya contamos con un *layout* en `app/views/layouts/application.html.erb`, pero para la sección de registro necesitamos un *layout* diferente.

Lo primero de vamos a hacer es crear un nuevo archivo `app/views/layouts/account.html.erb`, el contenido va a ser muy parecido a `app/views/layouts/application.html.erb`, pero vamos a hacer algunos ajustes.

```
# app/views/layouts/account.html.erb
```

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
<title>onepx</title>
```

```

    <%= stylesheet_link_tag      'account', media: 'all', 'data-turbolinks-track'
=> true %>

    <%= javascript_include_tag 'account', 'data-turbolinks-track' => true %>

    <%= csrf_meta_tags %>

</head>

<body>

    <%= yield %>

</body>

</html>

```

Los métodos `stylesheet_link_tag` y `javascript_include_tag` se utilizan en las plantillas para hacer referencia a los archivos manifiesto de hojas de estilo y de Javascript respectivamente. Si revisamos el `layout application.html.erb` veremos que los nombres de los manifiestos también se llaman `application`, tal y como en este `layout account` se llaman `account`, pero los nombres son totalmente una coincidencia, ya que en realidad para el `layout account` pudimos hacer referencia a los manifiestos `application` pero debido a que vamos a incluir cosas diferentes en cada manifiesto, es por eso que cada `layout` tiene el suyo propio.

Ahora vamos a crear los manifiestos para `account`. Primeramente el manifiesto para archivos de Javascript.

```
# app/assets/javascripts/account.js

//= require jquery

//= require jquery_ujs

//= require turbolinks
```

La sintaxis es un poco diferente a la sintaxis de el manifiesto de hojas de estilo, pero el significado es el mismo, `//= require` es para referenciar a un archivo, `//= require_tree .` hace referencia a los archivos dentro de una carpeta y finalmente `//= require_self` para hacer referencia al mismo archivo.

El manifiesto `account` para hojas de estilo se ve como sigue.

```
# app/assets/stylesheets/account.css

/*
 *= require pure-min

 *= require base

*/
```

Ya que tenemos los archivos necesarios para el `layout account`, el siguiente paso es como hacemos que Rails lo utilice? En este caso, deseamos que el controlador `RegistrationController` haga uso del nuevo `layout`, en lugar del `layout application` que es utilizado por omisión por todos los controladores.

Para que el controlador `RegistrationController` use el nuevo layout agregamos la siguiente línea al inicio del código del controlador `layout "account"`.

```
# app/controllers/registration_controller.rb

class RegistrationController < ApplicationController

layout "account"

...

end
```

Si en este momento vamos a nuestro navegador y refrescamos la pantalla, vamos a ver que Rails nos arroja un error.

`Asset was not declared to be precompiled in production.`

`Add `Rails.application.config.assets.precompile += %w( account.css )` to `config/initializers/assets.rb` and restart your server`

El error indica que Rails no reconoce el manifiesto `account.css` y nos da sugerencias de donde podemos registrarla, siendo en lugar indicado el archivo de inicialización `config/initializers/assets.rb`. Vamos a modificar ese archivo para que quede como sigue.

```
# config/initializers/assets.rb
```

```
# Be sure to restart your server when you modify this file.
```

```
# Version of your assets, change this if you want to expire all your assets.
```

```
Rails.application.config.assets.version = '1.0'
```

```

# Add additional assets to the asset load path

# Rails.application.config.assets.paths << Emoji.images_path

# Precompile additional assets.

# application.js, application.css, and all non-JS/CSS in app/assets folder are
already added.

Rails.application.config.assets.precompile += %w( application.js application.css
account.js account.css )

```

El cambio se llevó a cabo en la última línea donde se indicó cuales son los manifiestos que nuestra aplicación va a utilizar. Ahora reiniciamos el servidor de Rails, refreshamos la página y vemos que el error desapareció, aunque visualmente el formulario no cambió nada, pero si tenemos ya una separación entre los diferentes layouts.

En la carpeta `app/assets/stylesheets` vamos a crear una carpeta llamada `layouts`, dentro de esta carpeta vamos a crear un archivo de hojas de estilo llamada `_account.scss`, vamos a utilizar este archivo para incluir ahí el CSS que es específico para el `layout account`.

```

# app/assets/stylesheets/layouts/_account.scss

body {

background-color: $background-color;

}

```

```
form {  
  
background-color: #fff;  
  
border: 1px solid $border-color;  
  
}
```

Después creamos otro archivo llamado `account_base.scss` que nos va a servir como la base para el `layout` de `account`.

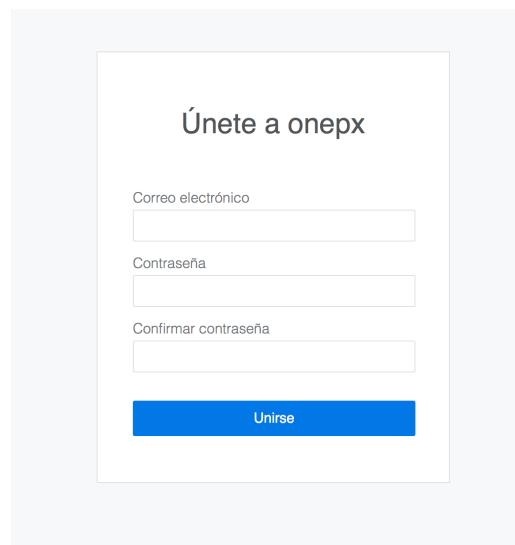
```
# app/assets/stylesheets/account_base.scss  
  
$background-color: #f7f8fa;  
  
$border-color: rgba(185, 193, 199, 0.5);  
  
@import "layouts/account";
```

Y finalmente requerimos en el archivo `account.css` a `account_base.scss`.

```
# app/assets/stylesheets/account.css  
  
/*  
  
*= require pure-min
```

```
*= require base  
  
*= require account_base  
  
*/
```

Recargamos la página en el navegador y vemos el cambio en el *layout* para *account*.



Ahora vamos a agregar el logo de nuestra aplicación encima de el formulario, con una liga que nos permita ir al inicio de la aplicación. Para esto vamos a modificar `app/views/layouts/account.html.erb` y justo entre el `body` y `<%= yield %>` agregamos lo siguiente:

```
# app/views/layouts/account.html.erb
```

```
...
```

```
<body>
```

```
<header>
```

```

<%= link_to root_path do %>

  <%= image_tag "onepx-small.png" %>

<% end %>

</header>

<%= yield %>

...

```

Con `link_to` agregamos una liga que nos llevara a `root_path`, el cuál contiene la `URL` que definimos como `root` en el archivo `config/routes.rb`. `image_tag` nos permite mostrar una imagen en las plantillas de HTML, es importante hacerlo de esta manera ya que hay que recordar que el `Asset Pipeline` va a agregarle una huella digital al nombre de la imagen y sí sólo utilizamos una etiqueta `img` de HTML, entonces el navegador va a indicar que la imagen no existe.

La imagen se copió a `app/assets/images` y se llama `onepx-small.png`.

Para que la imagen se muestre correctamente, agregamos un poco más de CSS al archivo `_account.scss`.

```

# app/assets/stylesheets/layouts/_account.scss

body {

background-color: $background-color;

```

```
}

header {

    text-align: center;

    margin-top: 2em;

}

form {

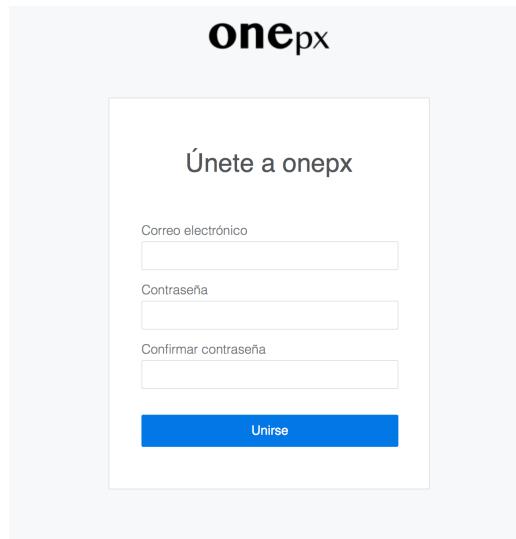
    background-color: #fff;

    border: 1px solid $border-color;

    margin-top: 3em !important;

}
```

Refrescamos una vez más la pantalla de el navegador y finalmente terminamos con las hojas de estilo para el *layout account*.



Al examinar el código HTML con las herramientas de desarrollo del navegador, vemos que `image_tag` generó la etiqueta HTML con el nombre de la imagen incluyendo la huella digital que agrega `Asset Pipeline`.

Ya con un estilo para el formulario de registro, vamos a ajustar el estilo de la página principal, la cual actualmente utiliza el *layout application*. Si hacemos click en el logo en la página de registro vemos que nos lleva a la página mencionada.

El método va a ser el mismo, se agregarán un algunos archivos `.scss` a la carpeta de `modules` y se registraran en un nuevo archivo llamado `application_base`.

Primeramente vamos a modificar el `layout application.html.erb` para agregar una barra de navegación.

```
# app/views/layouts/application.html.erb
```

```
<!DOCTYPE html>
```

<html>

...

```
<body>

<header class="navbar">

  <%= image_tag "onepx-tiny.png", class: "logo" %>

</header>

<!-- Content -->

<div class="content">
  <h1>Welcome to OnePixel</h1>
  <p>A simple static site generator for OnePixel</p>
</div>

</body>

</html>
```

Después modificamos la plantilla `index.html.erb` del controlador `HomeController` para dar la estructura necesaria.

```
# app/views/home/index.html.erb

<div class="home">

  <section class="hero">

    <div class="background"></div>

    <div class="inner">
      <h1>Welcome to OnePixel</h1>
      <p>A simple static site generator for OnePixel</p>
    </div>
  </section>
</div>
```

```
<div class="content">

  <div class="center">

    <div class="text">

      <div class="wrap">

        <h1><%= t(".hero.title") %></h1>

        <p><%= t(".hero.subtitle") %></p>

        <%= link_to t(".hero.join"), signup_path, class: "pure-button pure-
button-primary" %>

      </div>

    </div>

  </div>

</div>

</div>

</section>
```

```
<section class="info">
```

```
</section>
```

```
</div>
```

Debido a que agregamos algunas traducciones en la plantilla, ya en capítulo 4 hablamos de traducciones.

```
# config/locales/es.views.yml
```

```
es:
```

```
...
```

```
home:
```

```
index:
```

```
hero:
```

```
    title: "Aprendiendo Ruby on Rails 5."
```

```
    subtitle: "onepx es un proyecto en Ruby on Rails 5, parte del libro  
Aprendiendo Ruby on Rails 5."
```

```
join: "Unirse"
```

Modificamos el manifiesto `application.css` para que haga uso de otro archivo llamado `application_base.scss` y que sea este último el haga referencia a los archivos adicionales que necesitamos.

```
# app/assets/stylesheets/application.css

/*
.....
*= require pure-min

*= require application_base

*/
```

El archivo `application_base.scss` queda como sigue.

```
# app/assets/stylesheets/application_base.scss

@import "base";

@import "modules/navbar";

@import "modules/home";
```

Hace referencia a el archivo `base.scss` original, el cual requiere a los archivos comunes para todos los `layouts`, pero requiere, en este caso, un par de archivos que son específicos para el `layout application`: `_navbar.scss` y `_home.scss`.

`_navbar.scss` contiene estilos de CSS para la barra de navegación que agregamos al archivo `layout application.html.erb`.

```
# app/assets/stylesheets/modules/_navbar.scss

.navbar {
```

```
background-color: $white-color;

border-bottom: 1px solid $border-color;

box-sizing: border-box;

height: 50px;

margin: 0;

overflow: hidden;

padding: 0 30px 0 40px;

position: fixed;

text-align: left;

top: 0;

width: 100%;

z-index: 1000;

.logo {
```

```
margin-top: 15px;
```

```
}
```

```
}
```

`_home.scss` nos ayudará a darle un estilo al HTML de la plantilla `index.html.erb` del controlador `HomeController`.

```
# app/assets/stylesheets/modules/_home.scss
```

```
.home {
```

```
  .hero {
```

```
    .background {
```

```
      background: image-url("home-background.jpg") right 75% no-repeat;
```

```
      background-size: cover;
```

```
      bottom: 0;
```

```
      left: 0;
```

```
      position: absolute;
```

```
      right: 0;
```

```
    top: 0;

}

.content {

height: -webkit-calc(100vh - 136px);

height: -moz-calc(100vh - 136px);

height: calc(100vh - 136px);

min-height: 320px;

padding-top: 0;

position: relative;

z-index: 10

}

.center {
```





```
    font-size: 1.3em;  
  
    letter-spacing: 1px;  
  
    width: 650px;  
  
}  
  
}
```

```
.info {  
  
background-color: $white-color;  
  
min-height: 200px;  
  
position: relative;  
  
}
```

```
.pure-button-primary {
```

```

border-radius: 2px;

font-size: 1.2em;

font-weight: lighter;

letter-spacing: 1px;

padding: .5em 1em;

}

}

```

Es este archivo es importante tomarnos un momento para observar la línea `background: image-url("home-background.jpg") right 75% no-repeat;`. `image-url` es un *helper* de `Asset Pipeline`, el cuál nos ayuda a hacer referencia a una imagen que se encuentra en `app/assets/images` de manera que el `Asset Pipeline` pueda ponerle la huella digital al archivo y después modificar `image-url` con la instrucción correcta de CSS. En este caso `Asset Pipeline` convierte `background: image-url("home-background.jpg") right 75% no-repeat;` a una instrucción válida de CSS `background: url(/assets/home-background-9e426017d726e25c5ac7064e22b2c258db6793e33b64f5edb2f21aa51c677c35.jpg) right 75% no-repeat;`.

Finalmente un pequeño ajuste más que va a hacer la organización de las hojas de estilo más conveniente. Se creó un nuevo archivo `color.scss` para en él definir todas las variables de `SASS` que hacen referencia a los colores utilizados en las hojas de estilo.

```

# app/assets/stylesheets/_colors.scss

$background-color: #f7f8fa;

$foreground-color: #71767a;

```

```
$border-color: rgba(185, 193, 199, 0.5);
```

```
$header-color: #525558;
```

```
$error-color: #ff0000;
```

```
$white-color: #ffffff;
```

Y se modificó el archivo `base.scss` para que haga uso de él.

```
# app/assets/stylesheets/base.scss
```

```
@import "colors";
```

```
@import "modules/form";
```

Una vez que tenemos todos los cambios listos, abrimos el navegador en la *URL* `http://localhost:3000` y vemos cómo nuestra página de inicio cambió.



Hasta este punto hemos logrado un progreso importante, simplemente trabajando con el *Asset Pipeline* y las hojas de estilo, principalmente usando las sintaxis *SASS*.

Vamos a realizar unos últimos ajustes, en donde vamos a incluir el uso de Javascript, pero con la sintaxis de *ES6*.

En nuestro navegador vamos a abrir la dirección `http://localhost:3000/registro` y llenar todos los campos, por simplicidad podemos usar:

- Correo electrónico: `usuario-1@mail.com`
- Contraseña: `mi-secreto`
- Confirmación de contraseña: `mi-secreto`

Podemos reutilizar los valores una y otra vez, simplemente con incrementar el número en la dirección de correo electrónico. Presionamos el botón de *Únirme* y vemos que somos llevados a la página de inicio, pero no pasa nada más.

Si recordamos, en el capítulo 4, en la acción `create` del controlador `RegistrationController`, una vez que el nuevo usuario es guardado, se hace una redirección a `root_path` y agregamos un mensaje `notice: t('.welcome', email: @user.email)`, este mensaje se guarda en un objeto especial de Rails llamado `flash` y que está disponible para el la acción y el controlador al cuál fuimos redireccionados.

Examinando el *layout application*, vemos que contiene el código necesario para mostrar el mensaje que esté guardado en el objeto `flash` de Rails.

```
<%- if flash.notice.present? %>

<div class='flash-notice'>

<%= flash.notice %>

</div>

<%- end %>
```

Sin embargo al realizar la operación, de registrar un usuario, no se muestra. Esto se debe a la falta de estilo en CSS para el mensaje. Primeramente vamos a ajustar el HTML del *layout* para que el mensaje se muestre dentro de una etiqueta **p**.

```
<%- if flash.notice.present? %>

<div class='flash-notice'>

<p><%= flash.notice %></p>

</div>

<%- end %>
```

Vamos a agregar los estilos de CSS para los mensajes en **flash**

```
# app/assets/stylesheets/modules/_flash.scss

@mixin flash {

  box-sizing: border-box;

  color: $white-color;

  font-weight: lighter;

  height: 35px;

  overflow-y: hidden;
```

```
padding: 8px 20px;

position: absolute;

transition-property: all;

transition-duration: .5s;

transition-timing-function: cubic-bezier(0, 1, 0.5, 1);

top: 50px;

width: 100%;

z-index: 2000;

p {

margin: 0 auto;

max-width: 960px;

text-align: center;

}
```

```

}

.flash-notice {

background-color: $notice-color;

@include flash;

}

.flash-close {

height: 0px;

padding: 0px;

}

```

Algo a notar en este archivo de hoja de estilo, es el uso de la sintaxis **SASS** con las palabras claves **mixin** y **include**. Ambas trabajan en conjunto, con **mixin** se puede declarar una sección de CSS que va a ser reutilizada en otras lugares del archivo o bien en otros archivos de hojas de estilo. **include** es la instrucción con la que indicamos que deseamos reutilizar el CSS definido por **mixin**.

Ya con el archivo **\_flash.scss** guardado hacemos referencia de él en **base.scss**.

```
# app/assets/stylesheets/base.scss
```

```
@import "colors";  
  
@import "modules/flash";  
  
@import "modules/form";
```

Volvemos al navegador y registramos un nuevo usuario en <http://localhost:3000/registro> y ahora vemos que el mensaje se despliega correctamente.



El único problema es que no desaparece después de unos segundos, el mensaje se queda ahí hasta que refrescamos la página o vamos a otra dirección.

Para hacer que el mensaje desaparezca después de 5 segundos vamos a hacer uso de Javascript, pero antes vamos a configurar Rails para que nos permita hacer uso de la nueva sintaxis denominada [ES6](#).

Para activar [ES6](#) en el [Asset Pipeline](#) necesitamos agregar la gema [\*\*sprockets-es6\*\*](#) a nuestro manifiesto [\*\*Gemfile\*\*](#).

[\*\*sprockets\*\*](#) es la gema de Ruby que realmente realiza todo el trabajo detrás del [\*Asset Pipeline\*](#).

```
# Gemfile
```

```
...
```

```
gem 'sass-rails', '~> 5.0.6'
```

```
gem 'sprockets-es6'

gem 'uglifier', '>= 1.3.0'

...
```

Como siempre que modificamos este archivo, ejecutamos el comando `bundle` para instalar las gemas necesarias.

Ahora vamos a crea un archivo llamado en `flash.es6` dentro de la carpeta `app/assets/javascripts` con el siguiente contenido.

```
# app/assets/javascripts/flash.es6

class Flash {

  constructor() {

    this.flash = document.querySelectorAll('.flash-notice')

    if (this.flash.length > 0) {

      this.timeout = window.setTimeout(this.hideFlash.bind(this), 5000)

    }

  }

}
```

```

hideFlash() {

    let elements = Array.prototype.slice.call(this.flash)

    elements.forEach(
        function(element) {

            element.classList.add('flash-close')

        }
    )

    window.clearTimeout(this.timeout)

}

}

```

La clase `Flash` de Javascript busca todos los elementos de HTML que tengas una clase de CSS llamada `.flash-notice`, los guarda temporalmente e iniciativa un contador para que a los 5000 milisegundos (5 segundos) a esos elementos les agregue una clase adicional de CSS `.flash-close`, que se encarga de desaparecer el mensaje.

Finalmente lo único que resta es hacer que el código de Javascript de la clase `Flash` se ejecute cada que se cargue una página con el *layout application*. Abrimos el archivo `application.js` y modificamos su contenido de la siguiente forma.

```

# app/assets/javascripts/application.js

//


//= require jquery

//= require jquery_ujs

//= require turbolinks

//= require flash

//= require_self

document.addEventListener('turbolinks:load', function() {

  var flash = new Flash()

})


```

Cómo podemos ver, requerimos el archivo `flash.es6` - aunque no se utilice la extensión -, hacemos que el archivo `application.js` se requiera a sí mismo con `require_self` y después nos vinculamos con el evento `turbolinks:load` para ejecutar el código de el archivo `flash.es6`.

[Turbolinks](#) es una gema de Ruby on Rails que optimiza el cargado de páginas en las aplicaciones. Es por eso que en lugar de escuchar por el evento de Javascript `DOMContentLoaded` para saber cuando una página ya fue cargada, estamos utilizando el evento `turbolinks:load` de [Turbolinks](#).

Ya con estos últimos cambios, registramos un nuevo usuario, con el registro exitoso, Rails nos lleva a la página de inicio, vemos el mensaje de bienvenida, esperamos 5 segundos y el mensaje desaparece automáticamente.

Ahora que completamos los estilos de CSS y un poco de Javascript para la actual funcionalidad de nuestra aplicación, vamos a comprobar lo que se mencionó en un principio sobre el *Asset Pipeline* acerca de que une los múltiples archivos de recursos web que componen nuestra aplicación en base los manifiestos que tenemos para Javascript y las hojas de estilo.

La manera de comprobarlo de una manera simple es abriendo el archivo `config/development.rb` y al final del archivo, antes del último `end` agregamos `config.assets.debug = false`, guardamos el archivo, reiniciamos el servidor de Rails y recargamos la página de inicio.

Aparentemente no hay ningún cambio, pero si abrimos las herramientas de desarrollo del navegador que estamos usando, vamos a notar en la pestaña de red que el navegador sólo cargó un archivo `application-xxxx.css` y otro `application-xxxx.js`, donde `xxxx` es la huella digital de cada archivo. Aquí podemos ver el trabajo del *Asset Pipeline* en acción.



| Name  | Domain    | Type       | Method | Scheme |
|---|-----------|------------|--------|--------|
| localhost   | localhost | Document   | GET    | HTTP   |
| application-289c0fdbcb358475a62d6940324409fbcc920b...     | localhost | Stylesheet | GET    | HTTP   |
| application-cc75a89e55d01a608df5374a7dd095ae0298d0b975... | localhost | Script     | GET    | HTTP   |
| onepx-tiny-c1c42738f795c6c4387d559f828e989bc1d421b9a4...  | localhost | Image      | GET    | HTTP   |
| home-background-9e426017d726e25c5ac7064e22b2c258db6...    | localhost | Image      | GET    | HTTP   |

Una vez que comprobamos el trabajo del *Asset Pipeline* debemos eliminar la línea de código que agregamos anteriormente. En modo de producción esa línea existe en el archivo `config/production.rb` para asegurarse que en ese modo, Rails envía al usuario los archivos optimizados.

Con nuestra funcionalidad completa sólo queda ejecutar las pruebas automáticas para verificar que la funcionalidad de nuestra aplicación no se modificó de manera que cause algún problema. Para hacer esto ejecutamos el comando `rake test`.

```
$ rake test
```

```
Run options: --seed 23959
```

```
# Running:
```

```
E
```

```
Error:
```

```
UserSignupTest#test_dont_fill_in_form_at_signup,_click_button_and_see_form_errors:
```

```
Capybara::ElementNotFound: Unable to find button "Crear cuenta"
```

```
    test/features/user_signup_test.rb:31:in `block (2 levels) in <class:UserSignupTest>'
```

```
    test/features/user_signup_test.rb:30:in `block in <class:UserSignupTest>'
```

```
bin/rails test test/features/user_signup_test.rb:27
```

E

Error:

```
UserSignupTest#test_fill_in_form_at_/signup_and_see_welcome_message:
```

```
Capybara::ElementNotFound: Unable to find button "Crear cuenta"
```

```
    test/features/user_signup_test.rb:21:in `block (2 levels) in <class:User-
SignupTest>'
```

```
test/features/user_signup_test.rb:16:in `block in <class:UserSignupTest>'
```

```
bin/rails test test/features/user_signup_test.rb:13
```

F

**Failure:**

```
UserSignupTest#test_navigate_to_/signup_and_see_a_registration_form [/Users/mari-  
och/Development/onepx/test/features/user_signup_t$st.rb:9]:
```

```
Expected to find text "Crear una cuenta" in "Únete a onepx Correo electrónico  
Contraseña Confirmar contraseña".
```

```
bin/rails test test/features/user_signup_test.rb:6
```

```
.....
```

```
Finished in 1.131054s, 9.7254 runs/s, 19.4509 assertions/s.
```

```
11 runs, 22 assertions, 1 failures, 2 errors, 0 skips
```

Desafortunadamente vemos que hay 1 falla y 2 errores. Analizando los mensajes vemos que tienen que ver con el cambio en los textos del botón y los mensajes del formulario de registro, algo sencillo que corregir en nuestro archivo de pruebas `test/features/user_signup_test.rb`.

Buscamos el texto *Crear una cuenta* y lo reemplazamos con *Únete a onepx Correo electrónico* *Contraseña Confirmar contraseña*, después buscamos el texto *Crear cuenta* y lo reemplazamos con *Unirse*, guardamos los cambios y ejecutamos las pruebas automáticas nuevamente.

```
$ rake test
```

```
Run options: --seed 63664
```

```
# Running:
```

```
.....
```

```
Finished in 0.522501s, 21.0526 runs/s, 51.6745 assertions/s.
```

```
11 runs, 27 assertions, 0 failures, 0 errors, 0 skips
```

En esta ocasión no hay ningún error.

Preparamos el *commit* con nuestros cambios al repositorio de *Git* y con eso finalizamos este capítulo.

```
$ git add .
```

```
git commit -m "Código del capítulo 7"
```

## Conclusiones

En este capítulo conocimos la herramienta de Ruby on Rails para trabajar con archivos que son considerados como parte del *Frontend* de una aplicación web.

Aprendimos que el *Asset Pipeline* puede trabajar con diferentes sintaxis tanto para las hojas de estilo como para los archivos de Javascript. Además aprendimos que el *Asset Pipeline* les agrega una huella digital a los archivos ayudando a los navegadores a mantener una copia de los mismos y sólo actualizarlos cuando hay una versión nueva, reduciendo el tráfico a través de la red y dando la sensación de que las páginas se cargan más rápido después de la primera vez.

De igual forma vimos que para optimizar la transmisión de los archivos de hojas de estilo y Javascript, el *Asset Pipeline* se encarga de unirlos en uno o más archivos, dependiendo de cuantos manifiestos tenga nuestra aplicación para minimizar el número de archivos que se transmiten a través de la red.

Esta funcionalidad de unir los archivos que forma parte de un manifiesto es importante ya que nos permite trabajar con archivos más pequeños y que podemos reutilizar en otros manifiestos de la aplicación, lo que hace que nuestro trabajo sea un poco más sencillo al momento de trabajar con recursos web.

# Capítulo 8 - Envío de correos electrónicos y procesos asíncronos.

Una aplicación Web moderna, además de permitir a sus usuarios realizar acciones a través del navegador es indispensable que tenga la posibilidad de comunicarse a mediante correos electrónicos.

En una aplicación Web generalmente vamos a encontrar 2 tipos de correos electrónicos a enviar: Información general y correos transacciones.

En el primer tipo, hablamos de correos que van dirigidos a todos los usuarios de la aplicación, comúnmente son correos para comunicar cosas como una nueva funcionalidad, tiempos de mantenimiento de la aplicación, etc. Este tipo de correo se manejan a través de listas de distribución y normalmente son correos creados por el equipo de mercadotecnia y operados en una aplicación de terceros como [Mailchimp](#).

El Segundo tipo de correo, los transnacionales, son correo programados dentro de la aplicación Web para reaccionar a una acción de los usuarios, por ejemplo, cuando un usuario se registra en la aplicación es importante enviarle un correo de bienvenida comentándole lo emocionados que estamos de que decidiera a utilizar nuestro servicio.

Como ya mencioné, este tipo de correos, están programados dentro de la aplicación y son enviados a través de un servicio de correo electrónico mediante el protocolo [SMTP](#). Debido a que el servicio de [SMTP](#) generalmente es proporcionado por un proveedor ajeno a nuestra aplicación, es importante tomar en cuenta que la comunicación para enviar el correo puede tomar tiempo o puede ocurrir un error en el proceso, por lo que es buena idea para este tipo de integraciones crear procesos para enviarlos a una cola de procesamiento y que posteriormente sean atendidos.

En este capítulo vamos a trabajar con el segundo tipo de correos electrónicos, afortunadamente Ruby on Rails tiene funcionalidad para el desarrollo de correos electrónicos transaccionales y el deferir operaciones mediante procesos o procesos asíncronos.

## Trabajando con ActionMailer

[ActionMailer](#) es la herramienta dentro de Rails que nos permite el envío y recepción de correos electrónicos desde nuestra aplicación. Como vamos a ver durante el capítulo, [ActionMailer](#) nos

presenta [Mailers](#) y plantillas, los cuales son muy parecidos a los controladores y plantillas para mostrar [HTML](#) en el navegador.

Para nuestra aplicación vamos a crear un [Mailer](#) para enviar un correo de bienvenida cuando un usuario se registra a nuestra aplicación.

## *Creando nuestro primer Mailer*

Vamos a crear nuestro primer [Mailer](#), el cuál se llamará [WelcomeMailer](#), para esto vamos a hacer uso de los generadores de Rails, por lo tanto desde la terminal vamos a ejecutar el siguiente comando:

```
$ rails g mailer welcome

create  app/mailers/welcome_mailer.rb

create  app/mailers/application_mailer.rb

invoke  erb

create    app/views/welcome_mailer

create    app/views/layouts/mailер.text.erb

create    app/views/layouts/mailер.html.erb

invoke  minitest

create    test/mailers/welcome_mailer_test.rb

create    test/mailers/previews/welcome_mailer_preview.rb
```

Vemos que el generador crea varios archivos para nosotros. El primer archivo que vamos a revisar es `app/mailers/application_mailer.rb`, el cual contiene la clase `ApplicationMailer`, este archivo es la clase base de donde heredan todos nuestros `Mailers`, es como el  `ApplicationController` de nuestros controladores, aquí podemos definir configuración general para los `Mailers` o crear métodos que de igual manera van a estar disponibles para los `Mailers`.

En `ApplicationMailer` la configuración `default from:` define quién es el originario del correo electrónico, es decir va a indicar quien envió el correo al destinatario. Vamos a cambiar esta opción para que sea una dirección de nuestro dominio ficticio `onepx.com`, pero en lugar de escribir la dirección de correo electrónico en el archivo, vamos a indicarle que debe de leer esa información desde la configuración de Rails.

```
# app/mailers/application_mailer.rb

class ApplicationMailer < ActionMailer::Base

  default from: Rails.application.config.base_email

  layout "mailer"

end
```

Para definir un nuevo valor de configuración en Rails, `base_email` en este caso, vamos a crear el archivo `config/initializers/mailers.rb` en donde primeramente se verificará que la opción de configuración no está ya definida y así proceder a definirla con un valor por omisión.

```
# config/initializers/mailers.rb

unless Rails.application.config.respond_to?(:base_email)

  Rails.application.config.base_email = "no-reply@onepx.com"

end
```

Hacerlo de esta forma nos permite poner una dirección de correo base, y si para el ambiente de producción necesitamos que sea un valor diferente, podemos simplemente cambiar el valor dentro del archivo `config/environments/production.rb` y nuestro `ApplicationMailer` no necesita de cambiar.

Continuando con `ApplicationMailer` vemos que define un `layout` general para los correos electrónicos, este funciona exactamente igual que en los controladores, en donde se utiliza una plantilla base, en nuestro caso nos sirve para que todos los correo electrónicos se vean igual, el `layout` que va a utilizar es `mailer` y que fue creado por el generador.

Pero si examinamos a detalle los archivo generados, vamos a encontrar que el generador creo 2 archivos `layout`, uno con extensión `html.erb` y otro `text.erb`. Esto se debe a que Rails envía correos electrónico en los 2 formatos, `HTML` y texto, siendo el último para los clientes de correo que no pueden desplegar `HTML`.

Ahora vamos a revisar el archivo `app/mailers/welcome_mailer.rb` el cuál es el `Mailer` que estamos creando. Al abrirlo vemos que sólo cuenta con la definición de la clase y nada más.

```
# app/mailers/welcome_mailer.rb

class WelcomeMailer < ApplicationMailer

end
```

En este punto estamos listos para iniciar la implementación de nuestro `Mailer`. Como primer paso escribiremos una prueba para asegurarnos que nuestra implementación funciona como esperamos.

Abrimos el archivo `test/mailers/welcome_mailer_test.rb` donde vamos a escribir nuestra prueba.

```
# test/mailers/welcome_mailer_test.rb

require "test_helper"
```

```
class WelcomeMailerTest < ActionMailer::TestCase

  test "welcome email" do

    user = users(:user)

    email = WelcomeMailer.welcome_email(user)

    assert_emails 1 do

      email.deliver_now

    end

    assert_equal [user.email], email.to

    assert_equal [Rails.application.config.base_email], email.from

    assert_equal [I18n.t("welcome_mailer.welcome_email.subject")], email.subject

    assert_match(/Hola/, email.body.parts.first.to_s)

    assert_match(/Hola/, email.body.parts.last.to_s)

  end
end
```

```
end
```

```
end
```

Lo que buscamos verificar con nuestra prueba es que:

1. Podemos enviar un correo electrónico
2. El correo va dirigido a una cuenta de correo electrónico
3. El correo proviene de la cuenta de correo electrónico que configuramos
4. El motivo del email es el correcto
5. El cuerpo del correo electrónico, por lo menos contiene cierta información, para esto usamos una expresión regular. La prueba verifica el contenido de los 2 tipos posible de email.

Si ejecutamos la prueba en este momento vamos a tener un error:

```
$ bin/rails test test/mailers/welcome_mailer_test.rb
```

```
Run options: --seed 15917
```

```
# Running:
```

```
E
```

```
Error:
```

```
WelcomeMailerTest#test_welcome_email:
```

```
NoMethodError: undefined method `welcome_email' for WelcomeMailer:Class
```

```
test/mailers/welcome_mailer_test.rb:6:in `block in <class:WelcomeMailerTest>'
```

```
bin/rails test test/mailers/welcome_mailer_test.rb:4
```

```
Finished in 0.328182s, 3.0471 runs/s, 0.0000 assertions/s.
```

```
1 runs, 0 assertions, 0 failures, 1 errors, 0 skips
```

El problema es que nuestro *Mailer* aún no implementa el método `welcome_email`, vamos a implementarlo.

```
# app/mailers/welcome_mailer.rb

class WelcomeMailer < ApplicationMailer

  def welcome_email(user)
    ...
  end
end
```

Ejecutamos nuestra prueba de nuevo y ahora vemos que falla indicando que no email fue enviado.

```
$ bin/rails test test/mailers/welcome_mailer_test.rb
```

```
...
```

```
emails expected, but 0 were sent.
```

```
Expected: 1
```

Actual: 0

...

Vamos a implementar la funcionalidad de envío de correo.

```
# app/mailers/welcome_mailer.rb

class WelcomeMailer < ApplicationMailer

  def welcome_email(user)

    @user = user

    mail(to: @user.email, subject: I18n.t("welcome_mailer.welcome_email.subject"))

  end

end
```

Lo primer que hacemos en nuestro método es guardar la referencia del usuario en una variable de clase, después realizamos el envío llamando al método `mail` donde indicamos el destinatario y el motivo del correo electrónico.

Ejecutamos una vez más nuestra prueba y esperamos el resultado:

```
$ bin/rails test test/mailers/welcome_mailer_test.rb
```

...

```
WelcomeMailerTest#test_welcome_email:
```

```
ActionView::MissingTemplate: Missing template welcome_mailer/welcome_email with  
"mailer". Searched in:
```

```
* "welcome_mailer"
```

```
...
```

Ahora la prueba nos indica que no fue posible encontrar la plantilla para nuestro correo. Vamos a crear nuestra plantilla, pero en lugar de crear 1 vamos a crear 2, una con extensión de *html* y otra con la extensión *text*. Como se mencionó anteriormente, la necesidad de las dos plantillas es debido a que el correo electrónico que genera Rails es *multipart*, es decir incluye los 2 formatos y el cliente de correo decide cuál formato desplegar.

El contenido de los archivos es muy similar, utilizaremos al funcionalidad de internacionalización para crear el contenido para ambas plantillas..

```
# app/views/welcome_mailer/welcome_email.html.erb
```

```
<p><%= t(".greetings") %></p>
```

```
<p><%= t(".first_paragraph") %></p>
```

```
<p><%= t(".second_paragraph") %></p>
```

```
<p><%= t(".bye") %></p>
```

```
# app/views/welcome_mailer/welcome_email.text.erb
```

```
<%= t(".greetings") %>
```

```
<%= t(".first_paragraph") %>

<%= t(".second_paragraph") %>

<%= t(".bye") %>
```

El contenido de internacionalización para nuestro email lo agregaremos en un archivo específico para los [Mailers](#), `config/locales/es.mailers.yml`. Como ya hemos visto, los archivos de internacionalización iniciar con el lenguaje, en nuestro caso `es`. Después agregamos el contenido de nuestro email, hay que recordar que cómo cada llave tiene un `.` al inicio, Rails va a inferir las llaves previas definiéndonos así la estructura del archivo de internacionalización.

```
# config/locales/es.mailers.yml

es:

welcome_mailer:

welcome_email:

greetings: Hola,

first_paragraph: Gracias por registrarte e unirte a esta gran comunidad de
fotografía.
```

```
second_paragraph: Estamos ansiosos por ver la primera fotografía que compartirás con nosotros.
```

```
bye: El equipo de onepx.
```

Ejecutamos nuevamente las pruebas del email y ahora tenemos un resultado diferente.

```
bin/rails test test/mailers/welcome_mailer_test.rb
```

```
Finished in 0.318901s, 3.1358 runs/s, 25.0862 assertions/s.
```

```
1 runs, 8 assertions, 0 failures, 0 errors, 0 skips
```

Hemos completado nuestro primer Mailer en Rails.

## *Diseño básico de correo electrónico.*

Con nuestro primer Mailer completo, ahora vamos a darle un poco de diseño básico, para esto vamos a hacer uso la funcionalidad de per-visualización de Rails, pero antes hay algunas cosas que necesitamos saber.

Como ya se mencionó anteriormente, cuando utilizamos el generador se crearon 2 plantillas base para todos los correos electrónicos, una plantilla por cada formato de correo. En el caso de correos de texto el diseño es relativamente básico, para los correos con formato de *html* tenemos la posibilidad de utilizar *css* para crear correos más atractivos. Aunque podemos utilizar *html* y *css*, existen algunas limitaciones.

Dentro de las limitaciones de correos de *html* primeramente debemos de saber que el *css* debe ser incluido dentro del cuerpo del correo, ciertas cosas como *css* flotante puede no funcionar correctamente y la estructura del *html* es preferente realizarlo con tablas.

Ahora que estamos enterados de estas limitaciones, vamos a ver nuestro correo en el pre-visualizador, para tal efecto ejecutamos el servidor y nos dirigimos a */rails/mailers*, en la lista vamos a encontrar nuestro Mailer. Esta funcionalidad sólo está disponible cuando el servidor se ejecuta en modo de *development*.

Al hacer *click* en el nombre de nuestro correo electrónico podemos visualizarlo sin la necesidad de enviarlo, inclusive tenemos la opción de pre-visualizarlo en el formato que deseemos.



El primer archivo que vamos a modificar en la plantilla para los correos en formato *html*, le vamos a realizar los siguientes cambios.

```
# app/views/layouts/mailer.html.erb

<!DOCTYPE html>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<style>
```

```
body {  
  
    font-family: 'Arial', sans-serif;  
  
    font-size: 16px;  
  
}  
  
a {  
  
    color: #337ab7;  
  
    text-decoration: none;  
  
},  
  
a:hover {  
  
    color: #4face1;  
  
}  
  
table {  
  
    margin: 0 auto;  
  
    width: 960px;
```

```
        }
```

```
    th, .footer {
```

```
        text-align: center;
```

```
    }
```

```
    th {
```

```
        padding: 16px;
```

```
    }
```

```
    .footer {
```

```
        padding-top: 16px;
```

```
    }
```

```
</style>
```

```
</head>
```

```
<body>
```

```
<table>

<thead>

<tr>

<th>

<%= link_to root_url do %>

<%= image_tag 'onepx-tiny.png' %>

<% end %>

</th>

</tr>

</thead>

<tbody>

<tr>

<td>

<%= yield %>


```

```

</td>

</tr>

<tr>

<td class='footer'>

<%= link_to root_url do %>

<%= t('.footer', year: Date.today.year) %>

<% end %>

</td>

</tr>

</tbody>

</table>

</body>

</html>

```

Y para el archivo de plantilla de texto hacemos estos cambios.

```
# app/views/layouts/mailер.text.erb
```

```
<%= yield %>
```

```
<%= t('.footer', year: Date.today.year) %>. Visite <%= root_url %> para más información.
```

La diferencia de los cambios entre ambos es notable, obviamente el `html` es más extenso debido al `css` y `html`, pero en esencia los 2 muestran la misma información.

Si observamos los cambios veremos que usamos el `helper image_tag`, el cuál ya utilizamos anteriormente, y `root_url` del cuál habíamos utilizado la variación `root_path`. La diferencia entre ambos es que los `helpers` con la terminación `_path` generan una dirección relativa de nuestra aplicación y los que terminan con `_url` generan una dirección absoluta, es decir, incluyen el dominio de nuestra aplicación como parte de la ruta. Esto es importante porque cuando el correo electrónico

sea recibido en un cliente, el cliente debe tener la dirección completa para poderla abrir en el navegador.

Para que los `helpers image_tag` y `root_url` funcionen correctamente es necesario agregar las siguientes 2 líneas a los archivos que configuran los ambientes en nuestra aplicación. La primera es para que `image_tag` genere una dirección absoluta para las imágenes y la segunda para que `root_url` haga lo mismo.

Por ejemplo para `development` agregamos al final del archivo `config/environments/development.rb`, el valor `http://localhost:3600` debe de apuntar a nuestro servidor de desarrollo.

```
config.action_mailer.asset_host = 'http://localhost:3600'
```

```
config.action_mailer.default_url_options = { host: 'localhost:3600' }
```

En el caso del ambiente de `test`, modificamos el archivo `config/environments/test.rb` y el valor para ambas configuraciones lo fijamos como `http://example.com`, realmente no importa mucho, pero si es importante que exista, de lo contrario a ejecutar las pruebas tendremos el siguiente error:

```
Missing host to link to! Please provide the :host parameter, set default_url_options[:host], or set :only_path to true
```

Así como hicimos el cambio para `test` y `development` es necesario hacerlo para `production` en el archivo `config/environments/production.rb`, pero en esta caso el valor debe de ser el dominio de nuestra aplicación, por ejemplo <https://onepx.co>.

Dado que las plantillas de ambos tipos de correo utilizan traducciones, es necesario agregarlas al archivo `config/locales/es.mailers.yml`

```
# config/locales/es.mailers.yml
```

```
es:
```

```
layouts:
```

```
mailer:
```

```
footer: Derechos Reservados onepx %{year}.
```

```
welcome_mailer:
```

```
welcome_email:
```

```
subject: Bienvenido a onepx
```

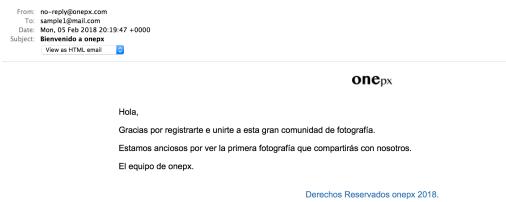
```
greetings: Hola,
```

**first\_paragraph:** Gracias por registrarte e unirte a esta gran comunidad de fotografía.

**second\_paragraph:** Estamos ansiosos por ver la primera fotografía que compartirás con nosotros.

**bye:** El equipo de onepx.

Después de realizar los cambios refrescamos el navegador y veremos los cambios en nuestro correo.



Una vez más ejecutamos las pruebas automáticas para verificar que pasen correctamente.

**bin/rails test**

.....

**Finished in 0.795405s, 15.0867 runs/s, 44.0027 assertions/s.**

**12 runs, 35 assertions, 0 failures, 0 errors, 0 skips**

## *Enviando correos electrónicos desde Rails*

Con nuestro correo electrónico listo vamos a modificar nuestro `RegistrationsController` para que se envíe el correo electrónico al momento que se completa el registro pero antes vamos a modificar la prueba del controlador para asegurarnos en la prueba de que esperamos el envío del correo al completar el registro.

Modificamos la prueba como sigue

```
# test/controllers/registration_controller_test.rb

# frozen_string_literal: true

require "test_helper"

class RegistrationControllerTest < ActionController::TestCase

  test "GET /new" do

    get :new

    assert_response :success

  end

end
```

```
test "POST /create without params" do

  assert_no_difference "ActionMailer::Base.deliveries.size" do

    post :create, params: { user: { email: "" } }

  end

  assert_response :success

end

test "POST /create with valid params" do

  assert_difference "ActionMailer::Base.deliveries.size", 1 do

    post :create, params: { user: { email: "newuser@mail.com",
                                   password: "pa$$word",
                                   password_confirmation: "pa$$word" } }

  end

end
```

```
    end

    assert_redirected_to root_path

end
```

end

El cambio consiste en poner la llamada a `post` a dentro de un bloque que en el primer caso verifique que no exista incremento y en el segundo caso que `ActionMailer::Base.deliveries.size` se incrementó por 1 su valor.

En el ambiente de `test` Rails no envía correos electrónicos, en lugar de eso los atrapa dentro de una arreglo en `ActionMailer::Base.deliveries` de manera que podemos comprobar si, en este caso, el controlador trató de enviar algún correo. Una vez realizados los cambios ejecutamos nuestras pruebas y vemos que fallan debido a que no se realizó el envío de email.

```
$ rake test
```

```
Run options: --seed 57811
```

```
# Running:
```

```
.....F
```

Failure:

```
RegistrationControllerTest#test_POST_/create_without_params [/Users/marioch/De-  
velopment/onepx/test/controllers/registration_controller_test.rb:12]:
```

```
"ActionMailer::Base.deliveries.size" didn't change by 1.
```

Expected: 1

Actual: 0

```
bin/rails test test/controllers/registration_controller_test.rb:11
```

```
Finished in 1.051135s, 11.4162 runs/s, 34.2487 assertions/s.
```

```
12 runs, 36 assertions, 1 failures, 0 errors, 0 skips
```

Lo cuál es esperado ya que no hemos realizado modificación alguna al controlador, ahora vamos a hacer el cambio en el método `create` del `RegistrationsController`.

```
# app/controllers/registration_controller.rb
```

...

```
def create

  @user = User.new secure_params

  if @user.save

    WelcomeMailer.welcome_email(@user).deliver_now

    return redirect_to root_path,
      notice: t(".welcome", email: @user.email)

  end

  render :new

end

...
```

El cambio consiste en realizar la llamada para que se envíe el correo electrónico `WelcomeMailer.welcome_email(@user).deliver_now`. Nuevamente ejecutamos nuestra pruebas y en esta ocasión finalizan correctamente.

```
$ rake test
```

```
Run options: --seed 13033
```

```
# Running:
```

```
.....
```

```
Finished in 0.936223s, 12.8175 runs/s, 39.5205 assertions/s.
```

```
12 runs, 37 assertions, 0 failures, 0 errors, 0 skips
```

## *Configurando Rails para enviar correos electrónicos*

Para los ambientes de *production* y *development* va a ser necesario utilizar un servicio de *SMTP* para el envío de correos electrónicos, existen varios servicios como [Sendgrid](#) o [Sparkpost](#) en donde es posible crear cuentas gratuitas con ciertas limitantes en comparación a las cuentas de paga. Generalmente para proceso de desarrollo de aplicaciones el uso de esas cuentas gratuitas es más que suficiente.

Existe una opción más para el ambiente de *development* en donde podemos usar un *SMTP* local llamado [Mailcatcher](#). Mailcatcher se instala de manera simple y cumple con la finalidad de permitirnos enviar y revisar correos electrónicos cuando estamos en el proceso de desarrollo.

Para instalar Mailcatcher lo hacemos desde la terminal.

```
$ gem install mailcatcher
```

Para iniciar Mailcatcher simplemente ejecutamos el siguiente comando.

```
$ mailcatcher
```

Starting MailCatcher

```
==> smtp://127.0.0.1:1025
```

```
==> http://127.0.0.1:1080
```

```
*** MailCatcher runs as a daemon by default. Go to the web interface to quit.
```

Si abrimos el navegador en la dirección <http://127.0.0.1:1080> vamos a ver el cliente de correo de Mailcatcher.



Para terminar la ejecución de Mailcatcher, si es que así lo deseamos, presionamos el botón *Quit* en la esquina superior derecha.

Ahora que tenemos Mailcatcher funcionando, vamos a configurar Rails para que haga uso de él. Abrimos el archivo `config/environments/development.rb` y al final del mismo agregamos la configuración de *SMTP* para el entorno de *development*.

```
#config/environments/development.rb
```

```
...
config.action_mailer.delivery_method = :smtp

config.action_mailer.smtp_settings = {

  address: 'localhost',

  port: 1025

}

...

```

Después de agregar la configuración, reiniciamos el servidor de Rails o lo iniciamos si es que no se está ejecutando y procedemos a crear un nuevo registro. Debido a que no estamos usando un servicio de *SMTP* real podemos poner cualquier correo electrónico aunque no exista de cualquier forma Mailcatcher lo va a recibir.

Una vez creado el registro hay que abrir Mailcatcher en el navegador usando la dirección <http://127.0.0.1:1080> y ahí vamos a encontrar el correo que se acaba de enviar por Rails.



Desde Mailcatcher vamos a poder ver el contenido del correo para el formato de [HTML](#), texto y el contenido de cómo fue enviado originalmente antes de ser interpretado por un cliente de correo electrónico.

En caso de haber seguido la ruta de no usar Mailcatcher y haber creado una cuenta con alguno de los servicios de [SMTP](#) mencionados anteriormente, entonces la configuración en el archivo `config/environments/development.rb` va a depender de lo que indique el servicio, pero generalmente se ve así.

```
config.action_mailer.delivery_method = :smtp

config.action_mailer.smtp_settings = {

  address: 'smtp.miservicio.com',

  domain: 'example.com',

  username: 'miusuario',

  password: 'micontraseña',

  port: 587

}
```

Algo a tener en cuenta, es que el correo electrónico que utilicemos para registrarnos debe de ser un correo que existe lo cual se convierte en un inconveniente cuando estamos desarrollando. Una opción es utilizar una funcionalidad de Rails llamada [interceptors](#) la cuál nos permite interceptar el correo electrónico enviado a un servicio real de [SMTP](#) y cambiarle la dirección de destino.

Vamos definiendo un interceptor para tener una idea más clara de cómo funciona. Hay que crear un archivo `lib/development_interceptor.rb` y agregarle el siguiente contenido.

```
# lib/development_interceptor.rb

class DevelopmentInterceptor

  def self.delivering_email(message)

    message.to = ["development@onepx.co"]

  end

end
```

Con el interceptor en el método `delivering_email` tenemos la referencia al mensaje de correo que se va a enviar y lo que haces en cambiar la dirección de destino, de esta manera podemos probar en desarrollo registrar cuentas de correo que no existen y aún así recibimos el correo de confirmación en una cuenta que nosotros designemos.

Para activar el interceptor hay que crear un inicializador en Rails que se encargue de cargarlo únicamente si estamos en el modo de `development`.

```
# config/initializers/development_interceptor.rb

if Rails.env.development?

  require Rails.application.root.join("lib/development_interceptor")
```

```
ActionMailer::Base.register_interceptor(DevelopmentInterceptor)
```

```
end
```

Para confirmar que funciona simplemente hay que ir a la página de registro y crear una cuenta con alguna dirección que exista o no y al recibirlo vamos a ver que llega a la dirección de correo que indicamos en el interceptor.

## Trabajando con ActiveJob

Hasta este momento nuestra aplicación de Ruby on Rails puede enviar correos electrónicos, pero lo hace de manera instantánea. Normalmente el envío de correos se hace a través de una cola de trabajos.

El motivo de hacerlo así es que si por algún motivo el servicio de envío de correos no está funcionando o está funcionando lentamente no se afecta al usuario recibiendo un mensaje de error o haciéndolo esperar hasta que el servicio responda, el usuario puede completar su acción y posteriormente el correo le será enviado.

### *Enviando correos electrónico de manera asíncrona*

Rails soporta varias librerías para [ActiveJob](#) que es el marco de trabajo dentro de Rails para las colas de trabajos. El más popular de todas las librerías es [Sidekiq](#) debido a que está diseñado para manejar colas de trabajo a grandes escalas, pero también es el que requiere de un poco más de complejidad para instalarse y configurarse.

Para nuestro proyecto vamos a utilizar [delayed job](#) como alternativa, es un poco más modesto de Sidekiq, pero cumple con su función y es más simple de instalar y configurar. De cualquier forma, los conceptos de como hacer de Rails trabaje con colas de trabajos son los mismos sin importar la librería.

Para instalar delayed job lo primero que tenemos que hacer es agregar la gema al archivo de [Gemfile](#).

```
# Gemfile
```

...

```
gem 'coffee-rails', '~> 4.2.1'
```

```
gem 'delayed_job_active_record'
```

```
gem 'jquery-rails'
```

...

Como siempre que modificamos el archivo hay que ejecutar el comando `bundle`.

```
$ bundle
```

...

```
Bundle complete! 27 Gemfile dependencies, 105 gems now installed.
```

```
Use `bundle info [gemname]` to see where a bundled gem is installed.
```

Dado que delayed job necesita de una tabla en la base de datos, vamos a ejecutar un generador que va a crear una migración y posteriormente vamos a realizar la migración.

```
$ bin/rails generate delayed_job:active_record
```

```
create  bin/delayed_job
```

```
chmod  bin/delayed_job
```

```
create  db/migrate/20180223002706_create_delayed_jobs.rb
```

```

$ bin/rails db:migrate

== 20180223002706 CreateDelayedJobs: migrating =====
-- create_table(:delayed_jobs, {:force=>true})
                               ^

-- > 0.0130s

-- add_index(:delayed_jobs, [:priority, :run_at], {:name=>"delayed_jobs_priority"})
                               ^

-- > 0.0040s

== 20180223002706 CreateDelayedJobs: migrated (0.0173s) =====

```

Solo falta indicarle a `ActiveJob` que queremos que utilice delayed job como el manejador de colas de trabajo. Esto lo hacemos en el archivo `config/application.rb`.

```

# config/application.rb

...
config.i18n.default_locale = :es

config.active_job.queue_adapter = :delayed_job

...

```

La línea que lo activa es `config.active_job.queue_adapter = :delayed_job`.

Para probar que delayed job quedó configurado correctamente en nuestra aplicación hay que cambiar la llamada del método `deliver_now` a `deliver_later` en el controlador `RegistrationsController`.

```
# app/controllers/registrations_controller.rb  
  
...  
  
  WelcomeMailer.welcome_email(@user).deliver_later  
  
...
```

Reiniciamos el servidor de Rails y registramos una nueva cuenta en esta ocasión veremos que el correo no es enviado, sin embargo el registro del usuario se completó. Lo que sucede es que `ActiveJob` a través de delayed job atrapó el envío de correo y lo tiene esperando en una cola de trabajos.

Para ejecutar todos los trabajos que se encuentren en la cola, en el caso de delayed job, ejecutamos el siguiente comando en la terminal.

```
$ bin/rails jobs:workoff  
  
[Worker(host:Marios-MacBook-Pro.local pid:72269)] Starting job worker  
  
[Worker(host:Marios-MacBook-Pro.local pid:72269)] Job ActiveJob::Queue-  
Adapters::DelayedJobAdapter::JobWrapper (id=1) (queue=mailers) RUNNING  
  
[Worker(host:Marios-MacBook-Pro.local pid:72269)] Job ActiveJob::Queue-  
Adapters::DelayedJobAdapter::JobWrapper (id=1) (queue=mailers) COMPLETED after  
0.6256  
  
[Worker(host:Marios-MacBook-Pro.local pid:72269)] 1 jobs processed at 1.4143 j/s,  
0 failed
```

```
[Worker(host:Marios-MacBook-Pro.local pid:72269)] No more jobs available. Exiting
```

El comando le indica a delayed job que deseamos ejecutar todos los trabajos que están pendientes en la cola de trabajos, una vez que finaliza la ejecución el proceso finaliza. En este momento si revisamos nuestros correos electrónicos vemos que se envió el que estábamos esperando.

Otra forma de ejecutar los trabajos en la cola de trabajo es ejecutando el comando `bin/rails jobs:work` en cuál va a mantener un proceso en memoria para ir ejecutando los trabajos en cuanto lleguen a la cola de trabajo.

Ya que Rails está listo para mandar correos a través de una cola de trabajos volvemos a ejecutar nuestras pruebas y vemos que estas fallan indicando que el correo electrónico que se esperaba se enviara no ha sido enviado.

```
$ bin/rails test
```

```
Run options: --seed 5898
```

```
# Running:
```

```
F
```

```
Failure:
```

```
RegistrationControllerTest#test_POST_/create_with_valid_params [/Users/marioch/Development/onepx/test/controllers/registration_controller_test.rb:21]:
```

```
"ActionMailer::Base.deliveries.size" didn't change by 1.
```

```
Expected: 1
```

```
Actual: 0
```

```
bin/rails test test/controllers/registration_controller_test.rb:20
```

```
.....
```

```
Finished in 0.871262s, 13.7731 runs/s, 40.1716 assertions/s.
```

```
12 runs, 35 assertions, 1 failures, 0 errors, 0 skips
```

Esto se debe a que el correo fue enviado a la cola de trabajos, lo cuál no es práctico para las pruebas. Para indicarle a delayed job que no queremos que los correos se vayan a través de la cola de trabajos en el ambiente de `test` vamos a crear un inicializador con el siguiente contenido.

```
# config/initializers/delayed_job.rb
```

```
Delayed::Worker.delay_jobs = !Rails.env.test?

Delayed::Worker.raise_signal_exceptions = :term

Delayed::Worker.logger = Logger.new(File.join(Rails.root, "log", "de-
layed_job.log"))
```

Aquí la parte importante es `Delayed::Worker.delay_jobs = !Rails.env.test?` donde indicamos que en el caso del ambiente de `test` queremos que los correos no se vayan a través de la cola de trabajos. Una vez realizado el cambio nuestras pruebas se ejecutan correctamente.

## Conclusiones

En este capítulo conocimos de `ActionMailer` que es la herramienta de Rails que nos permite el envío de correos transaccionales desde la aplicación. Vimos que con Rails es posible mandar 2 formatos de correos al mismo tiempo uno en `HTML` y otro en texto.

También vimos que podemos tener un `layout` que se aplique a todos los correos electrónicos para mantener uniformidad. Aprendimos que Rails tiene una herramienta para previsualizar los correos al momento que los estamos desarrollando y diseñando.

Vimos como realizar pruebas automáticas en nuestra lógica para asegurarnos del envío de correos, además de que integraremos `ActiveJob` en conjunto con delayed job para el envío de correos a través de colas de trabajo. Más adelante vamos a ver como utilizar `ActiveJob` para otro tipo de operaciones que toman más tiempo y/o más recursos.