Universidad de Alcalá

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

TRABAJO DE FIN DE GRADO

SOCIAL NETWORK MADE WITH RUBY on RAILS

Israel Sepúlveda Castillejo

Supervisado por:
Dr. Antonio Moratilla Ocaña

Dedicado a
mi familia

Agradecimientos

¡Muchas gracias a todos!

Resumen

Se trata de una bonita historia.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Érase una vez...

1.1. sección1

Bla bla bla

1.1.1. subsección1

Ble ble ble

1.1.1.1. subsubsección1

Bli bli bli

párrafo1 Blo blo blo

Capítulo 2

Ruby

2.1. Introducción

Ruby es un lenguaje de programación diseñado y desarrollado a mitad de los años 90 por *Yukihiro "Matz" Matsumoto* en Japón. De acuerdo con su creador, *Ruby* está influenciado por *Perl*, *Smalltalk*, *Eiffel*, *Ada* and *Lisp*. Soporta varios paradigmas de la programación, incluyendo funcional, orientación a objetos e imperativo. Además proporciona tipos dinámicos y tratamiento de memoria.

2.2. Historia

2.2.1. Concepto Inicial

Ruby fue concebido el 24 de Febrero de 1993. En un *post* de *ruby-talk* en 1999, *Matsumoto* describió sus ideas iniciales sobre el lenguaje:

> "Estaba hablando con un compañero de trabajo acerca de la posibilidad de un lenguaje orientado a objetos y a *scripting*. Yo sabía *Perl* (*Perl4*, no *Perl5*), pero no me gustó el lenguaje realmente porque "tenía un olor a programación de juguete (Aún la tiene)". Los lenguajes orientados a objetos parecían muy prometedores. Yo sabía *Python* por aquel entonces pero no me gustaba porque pensé que no era un verdadero lenguaje de programación orientado a objetos — Las características de orientación a objetos parecían un *add-on* o un *plugin* al lenguaje. Como maníaco de los lenguajes de programación y fan de la orientación a objetos por 15 años, realmente quise un lenguaje genuino orientado a objetos, fácil de usar en forma de lenguaje *script*. Busqué uno que satisficiera los requisitos pero no lo encontré. Así que decidí hacerlo yo." —Yukihiro "Matz" Matsumoto [1]

Matsumoto describe el diseño de *Ruby* tan simple como el de núcleo *Lisp*, con un sistema de orientación a objetos parecido a *Smalltalk*, bloques inspirados por funciones de orden superior[1] y una utilidad práctica parecida a la de *Perl*.

2.2.2. El nombre: *Ruby*

El nombre *Ruby* surgió durante una sesión de chat online entre *Matsumoto* y *Keiju Ishitsuka*, el 24 de Febrero de 1993, antes de haberse escrito ni una sola línea de código. Dos nombres surgieron inicialmente: *Coral* y *Ruby*. *Matsumoto* finalmente escoge *Ruby* en un email enviado a *Ishitsuka* posteriormente[2].

2.2.3. Primera publicación

El primer lanzamiento público de *Ruby 0.95* fue anunciado grupos de noticias locales de Japón el 21 de Diciembre de 1995. En consecuencia, tres versiones de *Ruby* más fueron lanzadas en dos días.

En este estado de desarrollo ya estaba presentes muchas de las funcionalidades y características familiares en las futuras versiones de *Ruby* como la Orientación a Objetos, clases y herencia de clases, mixins, iteradores, clausuras, manejo de excepciones y colector de basura.

2.2.4. Primeras versiones

Después del lanzamiento de *Ruby 0.95*, le siguieron varias versiones estables los próximos años.

1. *Ruby 1.0*: 25 de Diciembre de 1996
2. *Ruby 1.2*: Diciembre de 1998
3. *Ruby 1.4*: Agosto de 1999
4. *Ruby 1.6*: Septiembre de 2000
5. En 1997, el primer artículo sobre *Ruby* fue publicado en la Web. [2]

[1] Paradigma de la programación funcional. Una función puede tomar una o varias funciones como entradas y devolver la salida en una función.

[2] Una de las razones de la decisión final fue que la piedra rubí era la "piedra de nacimiento" de uno de sus compañeros de trabajo.

6. En 1998, *Ruby Application Archive* fue lanzado por *Matsumoto* con una simple homepage en inglés. [3]

7. En 1999, surge el primer email sobre el lenguaje en habla inglesa, *ruby-talk*, lo que eleva el interés por el lenguaje fuera de Japón. En el mismo año *Matsumoto* e *Ishitsuka* publican el primer libro sobre *Ruby*, *"The Object-oriented Scripting Language Ruby"* que se publicó en Japón.

8. Allá en el 2000, *Ruby* era ya más popular que *Python* en Japón. En Septiembre del año 2000, fue impreso el primer libro de *Ruby* en inglés.

9. En el año 2002, la lista de correo inglesa de *ruby-talk* ya tenías más tráfico de mensajes que la lista de correo japonesa demostrando que *Ruby* había incrementado su popularidad internacionalmente.

2.2.5. Ruby 1.8

Fue inicialmente lanzado en Agosto de 2003, fue estable durante mucho tiempo y finalmente fue retirado en Junio de 2013. A pesar de estar *deprecated*, todavía existe código basado en el. *Ruby* 1.8 es solo parcialmente compatible con *Ruby 1.9*.

Ruby 1.8 ha sido sujeto de varias estandarizaciones de la industria. Las especificaciones del lenguaje para *Ruby* fueron desarrolladas por *Open Standards Promotion Center* de *Information-Technology Promotion Agency*[4]. Fue aceptado como estándar internacional ISO/IEC 30170 en 2012. Alrededor de 2005, surge gran interés por *Ruby* gracias a *Ruby on Rails*, un *web application framework* escrito en *Ruby*, del que se hablará más a fondo posteriormente.

2.2.6. Ruby 1.9

Ruby 1.9 fue lanzado en Diciembre de 2007. Se introdujeron nuevas funcionalidades y mejoras sobre la versión 1.8 como el manejo de variables locales dentro de los bloques y sintaxis de lenguaje lambda. *Ruby 1.9* está obsoleto desde Febrero de 2015 por lo que lo que se recomienda encarecidamente a los usuarios, actualizar a una versión superior.

[3] Ruby Application Archive (RAA). RAA fue cerrado en 2003, véase el enlace `https://www.ruby-lang.org/en/news/2013/08/08/rip-raa/`. La documentación se ha trasladado a `https://rubygems.org/` y a `https://www.ruby-toolbox.com/`.

[4] Agencia gubernamental japonesa.

2.2.7. Ruby 2.0

Introdujo cambios significativos y pretendió ser completamente con *Ruby 1.9.3*. El lanzamiento oficial de *Ruby 2.0* el 24 de Febrero de 2013 solo contenía cinco incompatibilidades conocidas con la versión anterior.

2.2.8. Ruby 2.1

Ruby 2.1.0 fue lanzado el día de Navidad de 2013 y contenía corrección de bugs, actualización de librerías e incremento de rendimientos.

2.2.9. Ruby 2.2

Fue lanzado el día de Navidad de 2014. Su cambio más notorio es en el manejo y tratamiento de la memoria. Además incluye arreglos de errores, actualización de librerías y eliminación de APIs obsoletas.

2.2.10. Ruby 2.3

Lanzado el 25 de Diciembre de 2015, *Ruby 2.3* contiene bastantes mejoras, de las cuales las más notables son:

1. La habilidad de marcar todos literales de tipo string como *frozen* por defecto, mejorando ampliamente el rendimiento de las operaciones del tipo *string*.

2. Comparación de *hash* directa comprobando la clave y el valor en vez de únicamente las claves.

3. Una nueva forma de navegación segura para el operador `&.`, que facilita el tratamiento de valores nulos.

4. La gema `did_you_mean` is integrada por defecto.

5. La adicción de `Hash#dig` y `Array#dig` ayuda a la extracción de valores profundamente anidados.

2.3. Filosofía de Ruby

Matsumoto asiente que *Ruby* está diseñado para la productividad y la diversión. Propone que casi todos los lenguajes de programación, muchos ingenieros y científicos de computación están muy centrados en la máquina. Piensa que se debe facilitar el trabajo al programador y no al ordenador.

Aunque *Matsumoto* no lo hiciera a propósito, *Ruby* está asociado con un diseño POLA (Principle Of Least Astonishment), es decir, pretende evitar confusión a los usuarios más avanzados. [3]

2.4. Ruby.*new*

Matsumoto y *Ishitsuka*, en su libro *"The Object-oriented Scripting Language Ruby"*, comienzan explicando *Ruby* con un breve de resumen sobre tipos básicos, asignaciones, declaraciones y otros detalles, para posteriormente hacer un análisis profundo utilizando un enfoque top-down. En el presente documento se procederá a realizar un análisis que detalle las funcionalidades y particularidades de *Ruby* más importantes. Se respetará la estructura del libro de *Matsumoto* e *Ishitsuka*.

2.4.1. Ruby *es* un lenguaje Orientado a Objetos

Ruby es genuinamente un lenguaje Orientado a Objetos. Todo lo que se manipula en *Ruby* es un objeto, y los resultados de sus manipulaciones son objetos también.

En *Ruby*, los objetos son creados llamando a su constructor que es un método especial asociado a la clase del objeto. El constructor estándar es *new*.

```
song1 = Song.new("Billie Jean")
song2 = Song.new("Never gonna give you up")
# and so on
```

Éstas instancias provienen de la misma clase `Song` pero tienen características únicas. Primero, cada objeto tiene un identificador único de objeto (*object id*). Segundo, se pueden definir `instance variables`, variables con valores que son únicas a cada objeto. Estas variables instanciadas mantienen el estado del objeto. En el caso de la clase `Song`, por ejemplo, probablemente tendrán una variable instanciada que contenga el título de la canción.

Dentro de cada clase, se pueden definir métodos. Cada método es un bloque que realiza una funcionalidad que podrá ser llamada desde dentro de la clase (dependiendo de las restricciones de acceso) o desde fuera. Estos métodos tiene acceso a las variables instanciadas y por consiguiente, al estado del objeto.

Los métodos son invocados enviando un mensaje al objeto. El mensaje contiene el nombre del método y los parámetros que el método pueda necesitar.

```
"gin joint".length       >>      9
"Rick".index("c")        >>      2
-1942.abs                >>      1942
sam.play(aSong)          >>      "Never gonn..."
```

Al contrario que en otros lenguajes de programación, en *Ruby*, los métodos tienen métodos útiles y frecuentes de forma inherente; mientras que en otros lenguajes se debe llamar a una función diferente.

```
number = Math.abs(number); //Java code
```

```
number = number.abs #Ruby code
```

```
strlen("mecanica celeste"); //C code
```

```
"mecanica celeste".length #Ruby code
```

2.4.2. Arrays and Hashes

Los `arrays` y `hashes` de *Ruby* son colecciones indexadas. Ambos tipos de datos almacenan objetos que accesibles mediante una clave. La clave de los `arrays` son tipos enteros, mientras que los `hashes` soportan cualquier objeto como clave. Ambos crecen cuando necesitan almacenar nuevos objetos. Los `arrays` son más eficientes pero los `hashes` proporcionan mucho más flexibilidad.

Se pueden crear e inicializar un nuevo `array` usando cualquier `array literal` — conjunto de elementos en corchetes. Dado un objeto `array` , se puede acceder a los elementos individuales almacenados en el mismo proporcionando un índice entre corchetes.

```
a = [ 1, 'cat', 3.14 ]   # array tres elementos

# acceso al primer elemento
a[0]     >>      1

# asignacion de valor al tercer elemento
a[2] = nil

# mostrar array
a        >>      [1, "cat", nil]
```

Se pueden crear `arrays` usando cualquier `array literal`, como se mencionó previamente, o usando constructor de la clase `array`, `Array.new`.

```
empty1 = []
empty2 = Array.new
```

Los `hashes` de *Ruby* son similares a los `arrrays`. Un `hash literal` usa llaves en vez de corchetes. El literal debe proporcionar dos objetos por cada entrada: uno para la clave y otro para el valor.

```
instSection = {
  'cello'     => 'string',
  'clarinet'  => 'woodwind',
  'drum'      => 'percussion',
  'oboe'      => 'woodwind',
  'trumpet'   => 'brass',
  'violin'    => 'string'
}
```

Para recuperar el valor de un `hash` se procede con la misma notación que los `arrays`.

```
instSection['oboe']      >>      "woodwind"
instSection['cello']     >>      "string"
instSection['bassoon']   >>      nil
```

Como se muestra en el ejemplo anterior, un `hash`, por defecto, devuelve `nil` cuando es indexado con una clave que no está contenida en el `hash`. Para cambiar este comportamiento por defecto, se debe pasar un argumento al constructor del `hash`.

```
histogram = Hash.new(0)
histogram['key1']        >>      0
histogram['key1'] = histogram['key1'] + 1
histogram['key1']        >>      1
```

2.4.3. Estructuras de control

Ruby posee las estructuras de control típicas de los lenguajes de programación como instrucciones `if` y bucles `while`. *Ruby* no usa llaves en los cuerpos de las instrucciones, como C o Java; sin embargo utiliza `end` para finalizar un cuerpo.

```
if count > 10
  puts "Try again"
elsif tries == 3
  puts "You lose"
else
```

```
  puts "Enter a number"
end
```

De manera similar, las instrucciones `while` terminan con `end`.

```
while weight < 100 and numPallets <= 30
  pallet = nextPallet()
  weight += pallet.weight
  numPallets += 1
end
```

Ruby posee *statement modifiers* que son unos atajos que se pueden utilizar cuando el cuerpo de un `if` o `while` son solamente una única expresión. Simplemente se escribe la expresión seguido de `if` o `while` y la condición.

```
if radiation > 3000
  puts "Danger, Will Robinson"
end
```

Esto es lo mismo que lo anterior, reescrito usando un *statement modifier*.

```
puts "Danger, Will Robinson" if radiation > 3000
```

Análogamente, un bucle `while` usando y sin usar *statement modifiers*.

```
while square < 1000
   square = square*square
end
```

```
square = square*square  while square < 1000
```

2.4.4. Expresiones regulares

Casi todos los tipos de datos de *Ruby* son familiares para los programadores. La mayoría de lenguajes poseen cadenas de caracteres, flotantes, vectores... Sin embargo, hasta la aparición de *Ruby*, el soporte para expresiones regulares sólo estaba soportado en los llamados lenguajes de *script*, como Perl o Python. Las expresiones regulares, aunque resulten crípticas y en ocasiones complicadas, son una poderosa herramienta para trabajar con textos.

Una expresión regular es una forma de especificar un patrón de caracteres que debe corresponderse en una cadena de caracteres. En *Ruby*, típicamente se crea una expresión regular escribiendo un patrón entre dos barras de slash `/`.

```
/Perl|Python/
```

El carácter `|` se usa para separar las dos cosas que deben ser correspondidas. Los paréntesis se usan de la misma manera que expresiones aritméticas.

```
/P(erl|ython)/
```

Los caracteres `+` y `*` se usan para denotar la posibilidad de la presencia de una o muchas apariciones del mismo carácter, en el caso de `+`; y ninguna o muchas en el caso de `*`.

```
/ab+c/ %acepta abc, abbc, abbbc, ab..bc
/ab*c/ %acepta ac, abc, abbc, ab..bc
```

Se puede formar patrones por grupos de caracteres. Algunos de ellos son:

\s Se corresponde con espacios, tabulaciones, salto de linea, ...

\d Se corresponde con cualquier dígito.

\w Se corresponde con cualquier carácter que pueda aparecer en una palabra normal.

. El punto es el comodín. Acepta cualquier carácter.

```
# La hora con formato como 12:34:56
/\d\d:\d\d:\d\d/

# Perl, cero o mas caracteres, despues Python
/Perl.*Python/

# Perl, uno o mas espacios, luego Python
/Perl\s+Python/

# Ruby, un espacio, seguido de Perl o Python
/Ruby (Perl|Python)/
```

Una vez creado el patrón de la expresión regular, se utiliza el operador `=~` para intentar corresponder un `string` contra la expresión regular. Si se encuentra el patrón en la cadena, se devuelve su posición de inicio, si no, se devuelve `nil`. Se suelen emplear las expresiones regulares como condiciones en instrucciones `if` o `while`.

```
if line =~ /Perl|Python/
  puts "Scripting language mentioned: #{line}"
end
```

Otra forma útil de uso es reemplazar la parte de la cadena que casa con el patrón de la expresión regular por otro texto.

```
# Reemplaza el primer 'Perl' con 'Ruby'
line.sub(/Perl/, 'Ruby')

# Reemplaza todos 'Python' con 'Ruby'
line.gsub(/Python/, 'Ruby')
```

2.4.5. Bloques y Iteradores

Los bloques en *Ruby* son secciones de código que se pueden asociar de forma similar a un método a invocar. Se pueden utilizar bloques de código para implementar *callbacks*, para mover bloques de código de forma más flexible que las punteros a funciones de C, y para implementar iteradores.

Los bloques se denotan escribiendo código entre llaves o entre `do`...`end`.

```
{ puts "Hello" }        # bloque entre llaves

do                      #
  club.enroll(person)   # bloque do-end
  person.socialize      #
end                     #
```

Una vez creado un bloque, puede ser asociado con una llamada a un método. Ese método puede invocar el bloque una o varias veces con la instrucción `yield`.

```
#En este ejemplo se ejecuta dos veces el bloque
def callBlock
  yield
  yield
end

callBlock { puts "In the block" }

#Console output:

In the block
In the block
```

Los bloques de *Ruby* se pueden utilizar para implementar iteradores. Iteradores son métodos que retornan elementos sucesivos desde algún tipo de colección como un `array`.

```
# creacion del array
```

```
a = %w( ant bee cat dog elk )
 # iteracion sobre el array
a.each { |animal| puts animal }

#Console output:

ant
bee
cat
dog
elk
```

Siguiendo esta técnica, es fácil entender como iterar un array.

```
[ 'cat', 'dog', 'horse' ].each do |animal|
  print animal, " -- "
end

#Console output:

cat -- dog -- horse --
```

Muchas estructuras de bucles de otros lenguajes como C o Java están implementadas como simples llamadas a métodos en *Ruby*.

```
5.times {  print "*" }
3.upto(6) {|i|  print i }
('a'..'e').each {|char| print char }

#Console output:

*****3456abcde
```

2.5. Clases, Objetos y Variables

Para comentar esta sección y siguiendo con el esquema de Matsumoto, vamos a suponer que se pretende desarrollar una aplicación que simule el funcionamiento de un tocadiscos con el objetivo de entender con una aproximación más directa el modo de funcionamiento de Ruby.

Comenzamos creando una clase[5] `Song`[6] que contiene un único método,

[5]El concepto de *clase* se define como plantillas extensibles para crear objetos, proveyendo una serie de variables locales y un cuerpo para escribir sus métodos. W.B. Bruce [4]

[6]Los nombres de las clases deben empezar con letra mayúscula.

`initialize`[7], que instanciará un objeto y asignará valores a una serie de variables instancidas[8].

```
class Song
  def initialize(name, artist, duration)
    @name     = name
    @artist   = artist
    @duration = duration
  end
end
```

Veamos como instanciar un objeto a partir de la clase creada anteriormente.

```
aSong = Song.new("Never gonna give you up", "Rick
   Astley", 260)

aSong.inspect   >>
"#<Song:0x401b4924 @duration=260, @artist=\"Rick
   Astley\", @name=\"Never gonna give you up\">"
```

Para escribir el contenido del objeto, utilizaremos el método de *Ruby*, `.to_s` que renderiza cualquier objeto a una cadena de caracteres. Todos los objetos de *Ruby* presentan una serie de métodos incluidos por defecto. [9]

```
aSong = Song.new("Never gonna give you up", "Rick
   Astley", 260)
aSong.to_s      >>      "#<Song:0x401b499c>"
```

No resulta útil puesto que lo único que hace es mostrar el ID del objeto. Para resolver esto, vamos a anular "*override*" el comportamiento del método `.to_s`. En *Ruby*, las clases nunca están cerradas; siempre se pueden añadir métodos a una clase existente. Esto se aplica a las clases que el programador escribe como a las clases estándar que vienen por defecto en *Ruby*. Para lograrlo, se abre o edita la definición de la clase de una clase existente, y se añaden los nuevos contenidos que especifica el programador a lo que hubiera previamente en la definición.

```
class Song
```

[7]Los nombres de los métodos deben empezar con letra minúscula.

[8]Una variable instanciada se declara con un nombre empezando por letra minúscula precedido por un símbolo `@`.

[9]Clase `Object` de la documentación oficial de Ruby 2.3.0. Veasé la lista de métodos incluidos por defecto en la clase `Object`. `http://ruby-doc.org/core-2.3.0/Object.html`

```
  def to_s
    "Song: #{@name}--#{@artist} (#{@duration})"
  end
end
aSong = Song.new("Never gonna give you up", "Rick
   Astley", 260)
aSong.to_s       >>       "Song: Never gonna give you
   up--Rick Astley (260)"
```

Previamente se comentó como todos los objetos de *Ruby* soportan el método `.to_s`, la respuesta tiene que ver con herencia, por lo que será explicado en la siguiente sección.

2.5.1. Herencia y mensajes

Herencia en Orientación a Objetos es cuando una clase o un objeto está basado en otro objeto (herencia de prototipos) o en una clase (herencia basada en clases), usando la misma implementación (heredando de objeto o clase) o especificando la implementación para mantener el mismo comportamiento (realizando un interfaz; heredando comportamiento). Es un mecanismo de reutilización de código, que además, posibilita extensiones independientes del software original vía clases publicas e interfaces. Las relaciones de los objetos o clases a través de herencia propician una jerarquía. Herencia fue inventado para el lenguaje de programación Simula en 1967. [5]

En el caso de *Ruby*, la herencia permite crear una clase que es un refinamiento o especialización de otra clase. Siguiendo con el ejemplo de Matsumoto, se pretende dotar al tocadiscos de una funcionalidad de karaoke, por tanto, herencia de la clase `Song`; será una de las posibles aproximaciones para solventar el problema.

```
class KaraokeSong < Song
  def initialize(name, artist, duration, lyrics)
    super(name, artist, duration)
    @lyrics = lyrics
  end
end

aSong = KaraokeSong.new("My Way", "Sinatra", 225, "
   And now, the...")

aSong.to_s       >>       "Song: My Way--Sinatra (225)
   "
```

La salida por pantalla del método `.to_s` no es lo esperado puesto, que si una clase que hereda de otra, y que hace una llamada a un método de la clase padre, obviamente ésta llamada se realizará con el comportamiento de la clase padre. Para solucionar el problema, se debe redefinir el método para que tenga el comportamiento deseado en la clase hija.

```
class KaraokeSong
  # ...
  def to_s
    "KS: #{@name}--#{@artist} (#{@duration}) [#{
       @lyrics}]"
  end

  ### O mucho mejor, y para evitar acoplamiento:
  def to_s
    super + " [#{@lyrics}]"
  end
end

aSong = KaraokeSong.new("My Way", "Sinatra", 225, "
   And now, the...")

aSong.to_s       >>       "KS: My Way--Sinatra (225) [
   And now, the...]"
```

2.5.1.1. Herencia múltiple

Algunos lenguajes de Orientación a Objetos (C++ notablemente) soportan herencia múltiple, en la que una clase hereda de más de un padre inmediato. Pese a ser una herramienta muy poderosa, ésta técnica tiene riesgos ya que la jerarquía puede convertirse ambigua. Otros lenguajes, como Java, proporcionan herencia simple, lo que facilita y clarifica la implementación. Como contra, los objetos del mundo real tienen atributos de múltiples fuentes.

Ruby ofrece una solución de compromiso, dando la simplicidad de la herencia simple y la capacidad de herencia múltiple. *Ruby* solo puede tener un padre directo, lo que lo convierte en un lenguaje de herencia simple. Sin embargo, las clases de *Ruby* pueden incluir la funcionalidad de cualquier número de mixins[10], proporcionando una especie de herencia múltiple muy controlada pero sin sus riesgos.

[10]Un *mixin* es una definición parcial.

2.5.2. Atributos

El estado de los objetos de la clase `Song` que se crearon anteriormente tienen un estado interno que es privado a eso objeto; ésto es bueno porque significa que el objeto es responsable de mantener su propia consistencia. Sin embargo un objeto es que totalmente secreto es inútil puesto que puede ser creado pero no se puede interactuar con el mismo. Se deben definir métodos que permitan acceder y manipular el estado del objeto. Éstas facetas del objeto que son visibles desde el exterior, se denominan *atributos*.

```
class Song
  def name
    @name
  end
  def artist
    @artist
  end
  def duration
    @duration
  end
end

aSong = Song.new("Beat it", "Michael Jackson", 260)

aSong.artist    >>      "Michael Jackson"
aSong.name      >>      "Beat it"
aSong.duration  >>      260
```

Con el objeto de simplificar la sintaxis, *Ruby* proporciona un mecanismo que permite crear estos métodos que retornan el valor de las variables instanciadas.

```
class Song
  attr_reader :name, :artist, :duration
end

aSong = Song.new("Beat it", "Michael Jackson", 260)

aSong.artist    >>      "Michael Jackson"
aSong.name      >>      "Beat it"
aSong.duration  >>      260
```

2.5.2.1. Atributos modificables

Frecuentemente es necesario poder modificar y establecer los atributos desde fuera del objeto. En lenguajes como C++ o Java, esto se lleva a cabo con *setter functions*, mientras que en *Ruby*, los atributos de un objetos son accesibles asignando el valor de otra variable.

```
class Song
  def duration=(newDuration)
    @duration = newDuration
  end
end

aSong = Song.new("Beat it", "Michael Jackson", 260)

aSong.duration  >>      260
aSong.duration = 257   # modificacion del valor del
   atributo
aSong.duration  >>      257
```

De la misma manera, *Ruby* provee un mecanismo para crear estos métodos modificadores de atributos de forma más simple.

```
class Song
  attr_writer :duration
end

aSong = Song.new("Beat it", "Michael Jackson", 260)

aSong.duration  >>      260
aSong.duration = 257   # modificacion del valor del
   atributo
aSong.duration  >>      257
```

2.5.2.2. Atributos virtuales

Los variables calculadas o los atributos virtuales son métodos que internamente no se corresponden con la variable instanciada pero externamente parece un atributo normal.

```
class Song
  def durationInMinutes
    @duration/60.0
  end
```

```
  def durationInMinutes=(value)
    @duration = (value*60).to_i
  end
end

aSong = Song.new("Beat it", "Michael Jackson", 260)
aSong.durationInMinutes >>      4.333333333
aSong.durationInMinutes = 4.2
aSong.duration  >>      252
```

2.5.3. Variables de clase y métodos de clase

2.5.3.1. Variables de clase

Una variable de clase es una variable que es compartida por todos los objetos de la clase y además, accesible por todos los métodos que se describirán más adelante. Al contrario que las variables instanciadas, las variables de clase deben ser inicializadas antes de ser usadas.

En nuestro ejemplo de tocadiscos, se desea conocer cuantas veces una canción ha sido tocada pero también se quiere saber cuantas canciones han sido tocadas en total.

```
class Song
  @@plays = 0 #variable de clase
  def initialize(name, artist, duration)
    @name     = name
    @artist   = artist
    @duration = duration
    @plays    = 0
  end
  def play
    @plays += 1
    @@plays += 1
    "Reproducciones: #@plays . Total reproducciones:
       #@@plays ."
  end
end
```

```
s1 = Song.new("Song1", "Artist1", 234)  # test songs
  ..
s2 = Song.new("Song2", "Artist2", 345)
```

```
s1.play >>      "Reproduciones: 1. Total
   reproducciones: 1"
s2.play >>      "Reproduciones: 1. Total
   reproducciones: 2"
s1.play >>      "Reproduciones: 2. Total
   reproducciones: 3"
s1.play >>      "Reproduciones: 3. Total
   reproducciones: 4"
```

2.5.3.2. Métodos de clase

Las variables de clase son privadas a la clase y a sus instancias. Los métodos de clase proveen acceso a las variables de clase desde el exterior incluso sin estar ligado a un objeto en particular. Este es el caso de el método `new` que crea un nuevo objeto pero no está asociado el mismo a ningún objeto sino a la clase.[11] Los métodos de clase se distinguen de los métodos normales por su definición.

```
class Example
  def instMeth              # instance method
  end

  def Example.classMeth     # class method
  end
end
```

2.5.3.3. Singletons y otros constructores

En *Ruby* se puede modificar el comportamiento por defecto de como *Ruby* crea los objetos. Supongamos que desea un *log* para registrar el comportamiento de una cierta entidad. Obviamente la entidad tendrá único *log*, por tanto, se debe restringir la cantidad de objetos de tipo `Logger` que se pueden construir a uno solo.[12]

```
class Logger
  private_class_method :new
```

[11] Otros ejemplos claros de métodos de clase, los encontramos en librerias de *Ruby*, como por ejemplo, la clase `File`. La clase `File` provee numerosos métodos para manipular ficheros que no están abiertos y que por tanto no son un objeto de la clase `File`.

[12] La implementación de singletons necesita una serie de mecanismos adicionales para ser segura cuando multiples hilos están siendo ejecutados.

```
  #haciendolo privado, evitamos que se creen objetos
      utilizando el constructor por defecto.
  @@logger = nil
  def Logger.create
    @@logger = new unless @@logger
    @@logger
  end
end

Logger.create.id          >>      537766930
Logger.create.id          >>      537766930
```

2.5.4. Control de Acceso

Como sabemos, en *Ruby* sólo se puede cambiar el estado del objeto a través de sus métodos, por tanto, es importante considerar el nivel de exposición del objeto hacia el exterior. *Ruby* posee tres niveles de protección.

Métodos públicos Pueden ser llamados por cualquiera —luego no hay control de acceso. Los métodos son siempre públicos por defecto excepto `initialize` que es siempre privado.

Métodos protegidos Sólo pueden ser invocados por los objetos de la clase que definen o de sus subclases. El acceso queda restringido dentro de la familia del objeto.

Métodos privados No pueden ser llamados externamente. Puesto que no se puede especificar cuando un objeto cuando se usan, los metodos privados solo pueden ser llamados en la definición de la clase y por sus descendientes directos dentro del mismo objeto.

Ruby difiere de otros lenguajes orientados a objetos en que el control de acceso es determinado dinamicamente, por tanto, se notifica un error de violación de acceso solamente cuando el código intenta ejecutar un método restringido.

2.5.4.1. Especificando el control de acceso

Para especificar el control de acceso, se utilizan las funciones[13] `public`, `protected`, `private`. De forma similar al comportamiento de C++, después

[13]En *Ruby* son oficialmente funciones pero pueden ser consideradas como palabras reservadas.

de la utilizar la palabra clave los métodos consiguientes adoptarán ese nivel de acceso.

```
class MyClass

      def method1     # 'public' por defecto
        #...
      end

  protected           # metodos consiguientes seran '
     protected'

      def method2     # 'protected'
        #...
      end

  private             # metodos consiguientes seran '
     private'

      def method3     # 'private'
        #...
      end

  public              # metodos consiguientes seran '
     public'

      def method4     # 'public'
        #...
      end
end
```

Alternativamente, se puede establecer el nivel de control de acceso de los métodos, listándolos como argumentos de las funciones de control de acceso.

```
class MyClass

  def method1
  end

  # ... and so on

  public    :method1, :method4
  protected :method2
```

```
  private    :method3
end
```

2.5.5. Variables

Una variable es un espacio de memoria asociado a un nombre símbolico, el identificador, que contiene información conocida o desconocida referida como valor. Las variables son usadas para controlar y seguir los objetos, puesto que cada variable mantiene una referencia a un objeto.[14]

```
person = "Tim"
person.id          >>       537771100
person.type        >>       String
person   >>        "Tim"
```

La asignación de alias a los objetos, potencialmente puede producir múltiples variables que referencian al mismo objeto lo que puede dar problemas en el código de forma similar a Java.

```
#Codigo inseguro de "aliasing"
person1 = "Tim"
person2 = person1
person1[0] = 'J'
person1 >>       "Jim"
person2 >>       "Jim"
```

```
#Codigo seguro de "aliasing"
# El metodod dup de la clase String, crea un nuevo
   objeto con contenido similar.
person1 = "Tim"
person2 = person1.dup
person1[0] = "J"
person1 >>       "Jim"
person2 >>       "Tim"
```

[14]Es importante comentar que en *Ruby*, una variable no es objeto. Una variable es simplemente una referencia a un objeto.

2.6. Containers, Bloques e Iteradores

2.6.1. Containers

Ruby proporciona dos mecanismos para almacenar objetos dentro de una estructura. El tipo `Array` y el tipo `Hashes`.

2.6.1.1. Arrays

La clase `Array` mantiene una colección de referencias a objetos. Cada referencia a objeto ocupa una oposición del array, identificado por un índice entero no negativo. Los arrays pueden ser creados mediante literales[15] o explicitamente creando un objeto del tipo `Array`.[16]

```
a = [ 3.14159, "pie", 99 ]
a.type   >>      Array
a.length         >>      3
a[0]     >>      3.14159
a[1]     >>      "pie"
a[2]     >>      99
a[3]     >>      nil
b = Array.new
b.type   >>      Array
b.length         >>      0
b[0] = "second"
b[1] = "array"
b        >>      ["second", "array"]
```

Indexando arrays con enteros negativos, devuelve el valor comenzando desde el final, simpre y cuando, el valor absoluto del entero negativo no sea mayor que la longitud del array; en cuyo caso, se devolverá `nil`.

```
a = [ 1, 3, 5, 7, 9 ]
a[-1]    >>      9
a[-2]    >>      7
a[-99]   >>      nil
```

Se puede indexar arrays utilizando un par de números, `[start, count]`. Devuelve un nuevo array de longitud `count` con los valores del array original desde la posición `start`.

```
a = [ 1, 3, 5, 7, 9 ]
```

[15] Un literal de un array es simplemente una lista de objetos dentro de corchetes.
[16] Documentación de la clase `Array`, `http://ruby-doc.org/core-2.3.0/Array.html`

```
a[1, 3] >>       [3, 5, 7]
a[3, 1] >>       [7]
a[-3, 2]         >>       [5, 7]
```

2.6.1.2. Hashes

Los hashes son similares a los arrays en que son colecciones de objetos indexados. Sin embargo, mientras la indexación en arrays es con enteros, los hashes pueden ser indexados con objetos de cualquier tipo. Cuando se almacena un valor en un hash, se están proveyendo dos objetos —la clave y el valor entre llaves (`clave => valor`).[17]

```
h = { 'dog' => 'canine', 'cat' => 'feline', 'donkey'
    => 'asinine' }
h.length         >>       3
h['dog']         >>       "canine"
h['cow'] = 'bovine'
h[12]    = 'dodecine'
h['cat'] = 99
h        >>       {"cow"=>"bovine", "cat"=>99, 12=>"
   dodecine", "donkey"=>"asinine", "dog"=>"canine"}
```

Comparado con los arrays, los hashes tienen una gran ventaja: pueden usar cualquier objeto como índice. No obstante, tienen una desventaja importante, sus elementos están desordenados, por lo tanto, no se pueden crear fácilmente estructuras como pilas o colas.

2.6.2. Bloques e iteradores

Un iterador de *Ruby* es simplemente un método que invoca un bloque de código. Aunque los bloques parecen asemejarse a los de C o los de Java, presentan una serie de diferencias. Primero, un bloque puede aparecer solamente adyacente a la llamada al método; el bloque está escrito empezando en la misma linea como el último parámetro del método. Segundo, el bloque de código no es ejecutado cuando es encontrado. En vez de eso, *Ruby* "recuerda" el contexto en el que el bloque aparece[18], y luego entra en el método. Dentro del método, el bloque puede ser invocado, casi como si fuera método por si mismo, utilizando la instrucción `yield`. Siempre que la instrucción `yield` es ejecutada, es invocado el código en el bloque.

[17] Documentación de la clase `Hash`, `http://ruby-doc.org/core-2.3.0/Hash.html`

[18] variables locales, objeto actual, ...

```ruby
def threeTimes
  yield
  yield
  yield
end
threeTimes { puts "Hello" }

#Output:
Hello
Hello
Hello
```

```ruby
def fibUpTo(max)
  i1, i2 = 1, 1         # Asignacion paralela
  while i1 <= max
    yield i1
    i1, i2 = i2, i1+i2
  end
end
fibUpTo(1000) { |f| print f, " " }

#Output:
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Un bloque también puede retornar un valor al método. El valor de la última expresión evaluada en el bloque es pasado de vuelta al método como valor de la instrucción `yield`. Veamos en el siguiente fragmento de código esta última particularidad estudiando el funcionamiento de del método `find` de la clase `Array`.

```ruby
class Array
  def find
    for i in 0...size
      value = self[i]
      return value if yield(value)
    end
    return nil
  end
end
[1, 3, 5, 7, 9].find {|v| v*v > 30 }    >>      7
```

Aparte de `find`, otros iteradores comunes a varios tipos de colecciones de *Ruby* son: `each` y `collect`.

```
[ 1, 3, 5 ].each { |i| puts i }

#Output:
1
3
5
```

```
["H", "A", "L"].collect { |x| x.succ }  >>      ["I"
   , "B", "M"]
```

2.6.2.1. Bloques para transacciones

Los bloques de *Ruby* pueden ser utilizados para definir un fragmento de código que debe ser ejecutado bajo ciertas condiciones de control de transacciones. Por ejemplo, después abrir un fichero y trabajar con su contenido, se debe asegurar de que el fichero es cerrado al terminar.

```
#Esta implementacion podria realizarse con codigo
   convencional y no comtempla el manejo de
   excepciones.
class File
  def File.openAndProcess(*args)
    f = File.open(*args)
    yield f
    f.close()
  end
end

File.openAndProcess("testfile", "r") do |aFile|
  print while aFile.gets
end
```

2.7. Tipos básicos

2.7.1. Números

Ruby soporta enteros y número flotantes[19]. Los enteros pueden ser de cualquier longitud[20]. Los enteros que están entre cierto rango (normalmente

[19]Documentación de la clase `Float` de *Ruby* 2.3.0, `http://ruby-doc.org/core-2.3.0/Float.html`

[20]Hasta el máximo determinado por la memoria libre del sistema.

-2^{30} a $2^{30}-1$ o -2^{62} a $2^{62}-1$) son mantenidos internamente en forma binaria, y son objetos de la clase `Fixnum`[21]. Los enteros fuera de ese rango son mantenidos en objetos de la clase `Bignum`[22][23]. Este proceso es transparente, y *Ruby* maneja automáticamente la conversión hacia adelante y hacia atrás.

```
num = 8
6.times do
  print num.type, " ", num, "\n"
  num *= num
end

#Output:
Fixnum 8
Fixnum 64
Fixnum 4096
Fixnum 16777216
Bignum 281474976710656
Bignum 79228162514264337593543950336
```

```
#Se pueden escribir enteros con un signo opcional o
   con un indicador de la base.
123456                    # Entero corto
123_456                   # Entero corto (Guion bajo
    es ignorado)
-543                      # Entero corto negativo
123_456_789_123_345_789   # Entero largo
0xaabb                    # Hexadecimal
0377                      # Octal
-0b101_010                # Binario (negado)
```

2.7.2. Strings

En *Ruby*, las cadenas de caracteres son simplemente secuencias de 8-bit bytes. Normalmente almacenan caracteres pero también pueden contener datos binarios. Las cadenas de caracteres son objetos de la clase `String`[24].

[21] Documentación de la clase `Fixnum` de *Ruby* 2.3.0, `http://ruby-doc.org/core-2.3.0/Fixnum.html`

[22] Documentación de la clase `Bignum` de *Ruby* 2.3.0, `http://ruby-doc.org/core-2.3.0/Bignum.html`

[23] La clase `Bignum` está implementada actualmente con un set de enteros cortos.

[24] Documentación de la clase `String` de *Ruby* 2.3.0, `http://ruby-doc.org/core-2.3.0/String.html`

```
#Caracteres de escape
'escape using "\\"'      >>      escape using "\"
'That\'s right' >>       That's right

#Literales de cadena entre comillas dobles
"Seconds/day: #{24*60*60}"       >>      Seconds/day:
    86400
"#{'Ho! '*3}Merry Christmas"     >>      Ho! Ho! Ho!
   Merry Christmas
"This is line #$."       >>      This is line 3

#Otras formas de crear literales
%q/general single-quoted string/         >>
   general single-quoted string
%Q!general double-quoted string!         >>
   general double-quoted string
%Q{Seconds/day: #{24*60*60}}     >>      Seconds/day:
    86400
```

2.7.3. Rangos

Ruby intenta ayudar a modelizar la realidad, por lo tanto, es natural que *Ruby* soporte rangos. De hecho, *Ruby* usa rangos para implementar tres características separadas: secuencias, condiciones, e intervalos.

2.7.3.1. Rangos como secuencias

El uso más natural de los rangos es para expresar secuencias. Las secuencias tiene un punto de inicio y punto de fin, y una manera de producir sucesivos valores en la secuencia. En *Ruby*, las secuencias son creadas usando los operadores de rango ".." y "...". La forma "dos-puntos" se utiliza para crear un rango inclusivo, mientras que la forma "tres-puntos" crea un rango que excluye el valor más alto especificado.

```
1..10
'a'..'z'
0...anArray.length
```

En *Ruby*, los rangos no son representados internamente como listas. La secuencia 1..1000000 es mantenida como un objeto `Range`[25] que contiene dos

[25]Documentación de la clase `Range` de *Ruby* 2.3.0, `http://ruby-doc.org/core-2.3.0/Range.html`

referencias a objetos de la clase `Fixnum`.

```
#Se puede convertir un objeto de la clase Range a
   una lista usando el metodo to_a.
(1..10).to_a     >>       [1, 2, 3, 4, 5, 6, 7, 8, 9,
   10]
('bar'..'bat').to_a      >>       ["bar", "bas", "bat"
   ]
```

La clase `Range` implementa métodos que permiten iterar sobre los rangos y sus contenidos.

```
digits = 0..9
digits.include?(5)       >>       true
digits.min       >>       0
digits.max       >>       9
digits.reject {|i| i < 5 }       >>       [5, 6, 7, 8,
    9]
digits.each do |digit|
  dial(digit)
end
```

Los rangos no se quedan limitados a tipos numéricos o cadenas de caracteres; en *Ruby*, como se puede esperar de un lenguaje orientado a objetos, se pueden crear rangos de objetos definidos por el programador. La única constante es que los objetos deben responder al método `succ`, retornando el siguiente objeto de la secuencia y los objetos deben ser comparables usando operadores de comparación (`<=>`).

```
#Implementacion de una clase que representa una fila
    de caracteres #.
class VU

  include Comparable

  attr :volume

  def initialize(volume)  # 0..9
    @volume = volume
  end

  def inspect
    '#' * @volume
  end
```

```
  # Support for ranges

  def <=>(other)
    self.volume <=> other.volume
  end

  def succ
    raise(IndexError, "Volume too big") if @volume
       >= 9
    VU.new(@volume.succ)
  end
end

#Output:
#Testeo creando un rango de objetos vu.

medium = VU.new(4)..VU.new(7)
medium.to_a     >>      [####, #####, ######,
   #######]
medium.include?(VU.new(3))      >>      false
```

2.7.3.2. Rangos como condiciones

De la misma manera que los rangos representan secuencias, los rangos también pueden ser utilizados como expresiones condicionales.

```
#Ejemplo de un rango que implementa una condicion.
#El codigo imprime un grupo de lineas de la entrada
   estandar, donde la primera linea de cada grupo
   que contiene la parabra "start" y en la que la
   ultima linea contenga la palabra "end".
while gets
  print if /start/../end/
end
```

2.7.3.3. Rangos como intervalos

El último uso de los rangos son los intervalos. Se pueden comprobar si cierto valor está dentro de un intervalo representado por un rango. Se realiza mediante el operador "===".

```
(1..10)     === 5          >>       true
(1..10)     === 15         >>       false
(1..10)     === 3.14159    >>       true
('a'..'j') === 'c'         >>       true
('a'..'j') === 'z'         >>       false
```

2.7.4. Expresiones regulares

Expresiones regulares son utilizadas para probar patrones contra cadenas de caracteres. Las expresiones regulares son objetos de la clase `Regexp`[26] y pueden ser creados mediante la llamada explicita al constructor o mediante el uso de la forma literal /pattern/ y %\pattern\.

```
a = Regexp.new('^\s*[a-z]')      >>       /^\s*[a-z]/
b = /^\s*[a-z]/ >>       /^\s*[a-z]/
c = %r{^\s*[a-z]}        >>       /^\s*[a-z]/
```

Una vez creado un objeto de expresión regular, se puede comprobar contra una cadena utilizando `Regexp#match(aString)` o con los operadores de casamiento[27].

```
a = "Fats Waller"
a =~ /a/         >>       1
a =~ /z/         >>       nil
a =~ "ll"        >>       7
```

La comprobación de la cadena de caracteres contra la expresión regular devuelve `nil` si no se produce coincidencia o si se produce coincidencia, la posición del primer carácter de donde la coincidencia se ha producido.

2.7.4.1. Patrones

Toda expresión regular contiene un patrón, que es usado para casar la expresión regular contra una cadena de caracteres. Dentro de un patrón, todos los caracteres excepto , y `?`, coinciden con ellos mismos. Si se pretende utilizar uno de esos caracteres especiales, deben de ir precedidos de una barra invertida \.

[26] Documentación de la clase `Regexp` de *Ruby* 2.3.0, `http://ruby-doc.org/core-2.3.0/Regexp.html`

[27] `=~`, para comprobación positiva; `!~`, para comprobación negativa.

2.7.4.2. Anchors

Por defecto, una expresión regular intentará encontrar la primera coincidencia del patrón sobre la cadena de caracteres. Los patrones `^` y `$` comprueban el principio y el final de linea, respectivamente. Estos son frecuentemente utilizados como *anchor* de un patrón.

\A coincide con el principio de una cadena.

\z y \Z coincide el final de una cadena si la cadena no termina con $\backslash n$, en cuyo caso, coincidirá hasta antes de $\backslash n$.

\b puede coincidir con inicio de palabra.

\B puede coincidir con una subcadena dentro de una palabra.

2.7.4.3. Clases de caracteres

Una clase de carácter es un grupo de caracteres entre corchetes que coincidirá con cualquier carácter incluido entre esos corchetes. Existen algunas abreviaciones de los caracteres de clase.

\d	[0-9]	Carácter de dígito
\D	[^0-9]	Carácter de no-dígito
\s	[\s\t\r\n\f]	Carácter de espacio
\S	[^\s\t\r\n\f]	Carácter de no-espacio
\w	[A-Za-z0-9_]	Carácter de palabra
\W	[^A-Za-z0-9_]	Carácter de no palabra

2.8. Métodos

Otros lenguajes de programación poseen funciones, procedimientos, métodos o rutinas, pero en *Ruby* sólo existe *el método*, un bloque de expresiones que retornan un valor.

2.8.1. Definiendo un método

Como se ha visto a lo largo del capitulo, un método se define utilizando la palabra clave `def`. Los nombres de los métodos deben empezar con una letra minúscula.[28] Los métodos que actúan como consulta son frecuentemente

[28]No se recibe un error inmediatamente si se usa una letra mayúscula, pero cuando *Ruby* detecta que se está llamando al método, primero supone que es una constante, y no una invocación a un método, y como resultado la llamada puede ser analizada gramaticalmente erronea.

llamados con el sufijo `?`. Los métodos que son "peligrosos", o que modifican los datos que reciben; pueden ser llamados con el sufijo `!`[29].

```
def myNewMethod(arg1, arg2, arg3)      # 3 argumentos
  # Code for the method would go here
end

def myOtherNewMethod                   # No
   argumentos
  # Code for the method would go here
end
```

Ruby permite especificar valores por defecto a los argumentos de los métodos —valores que se serán usados si cuando se invoca no se pasan explicitamente.

```
def coolDude(arg1="Miles", arg2="Coltrane", arg3="
   Roach")
  "#{arg1}, #{arg2}, #{arg3}."
end
coolDude          >>       "Miles, Coltrane, Roach."
coolDude("Bart")          >>       "Bart, Coltrane,
   Roach."
coolDude("Bart", "Elwood")       >>       "Bart,
   Elwood, Roach."
coolDude("Bart", "Elwood", "Linus")      >>       "
   Bart, Elwood, Linus."
```

2.8.2. Invocación del método

Para invocar un método se especifica un receptor, el nombre del método, y de forma opcional algunos parámetros y un bloque asociado. Si se omite el receptor, por defecto, *Ruby* presupone que `self` es el objeto actual.

```
connection.downloadMP3("jitterbug") { |p|
   showProgress(p) }

#Para metodos de clases o de modulos, el receptor es
    la clase o el nombre del modulo.
File.size("testfile")
```

[29]Por ejemplo, `String` proporciona `chop` y `chop!`. El primero retorna la cadena modificada y el segundo modifica la cadena directamente.

```
Math.sin(Math::PI/4)

#Se omite el receptor.
self.id >>       537794160
id      >>       537794160
self.type        >>      Object
type    >>       Object
```

Los parámetros opcionales van seguidos del nombre del método. Si no hay ambigüedad, se pueden omitir los paréntesis que rodean a la lista de argumentos cuando se llama al método.[30]

```
a = obj.hash     # Lo mismo
a = obj.hash()   # que esto.

obj.someMethod "Arg1", arg2, arg3    # Misma
   resultado
obj.someMethod("Arg1", arg2, arg3)  # que con
   parentesis.
```

2.9. Expresiones

Una expresión es una combinación de uno o más valores, constantes, funciones, variables operadores, y funciones que el lenguaje de programación interpreta, siguiendo unas reglas de precedencia y asociación particulares, y computa para producir otro valor.

En *Ruby*, se considera expresión cualquier cosa que razonablemente retorna una valor, de hecho, algunas de las instrucciones de C o Java son expresiones en Ruby.

```
#Por ejemplo, if o case retornan el valor de la
   ultima expresion ejecutada.
songType = if song.mp3Type == MP3::Jazz
             if song.written < Date.new(1935, 1, 1)
               Song::TradJazz
             else
               Song::Jazz
             end
           else
```

[30] Algunas documentaciones de *Ruby* se refieren a las llamadas a métodos con argumentos sin paréntesis como "comandos".

```
              Song::Other
            end

rating = case votesCast
         when 0...10    then Rating::SkipThisOne
         when 10...50   then Rating::CouldDoBetter
         else                Rating::Rave
         end
```

2.9.1. Operadores

Ruby tiene el set básico de operadores (+, -, *, /, ...). En *Ruby*, muchos operadores son realmente llamadas a métodos. Cuando se escribe `a*b+c`, se está realmente pidiendo que el objeto referenciado por`a` ejecute el método `*`, pasando el parámetro `b`. Después se pide al objeto el resultado del cálculo de ejecutar `+`, pasando `c` como parámetro. En *Ruby* todo es un objeto, hasta tal punto que se pueden redefinir los operadores.

```
#Redefinicion del operador suma.
class Fixnum
  alias oldPlus +
  def +(other)
    oldPlus(other).succ
  end
end

1 + 2   >>      4
a = 3
a += 4  >>      8
```

Más útil es el hecho de que las clases pueden participar en expresiones como si fueran construidas por defecto.

```
#En este ejemplo, redefinimos el operador de
   indexacion [] para que retorne una seccion de una
    cancion.

class Song
  def [](fromTime, toTime)
    result = Song.new(self.title + " [extract]",
                      self.artist,
                      toTime - fromTime)
```

```
    result.setStartTime(fromTime)
    result
  end
end

aSong[0, 0.15].play
```

2.9.2. Otras expresiones

Ademas de las expresiones obvias, las llamadas métodos y las expresiones de instrucción, *Ruby* tiene más cosas que pueden ser utilizadas como expresiones.

2.9.2.1. Comandos

Si se encapsula una cadena de caracteres en comillas invertidas o usando el prefijo `%x`, se ejecutará, por defecto, como comando sobre el sistema operativo en que se esté ejecutando. El valor de la expresión es la salida estándar del comando.

```
`date`   >>       "Sun Jun  9 00:08:26 CDT 2002\n"
`dir`.split[34] >>      "lib_singleton.tip"
%x{echo "Hello there"}  >>      "Hello there\n"
for i in 0..3
  status = `dbmanager status id=#{i}`
  # ...
end
```

2.9.3. Asignación

Una asignación establece una variable o atributo de la parte izquierda, el valor de la parte derecha; y devuelve ese valor como resultado de la expresión de asignación.

```
a = b = 1 + 2 + 3
a         >>       6
b         >>       6
a = (b = 1 + 2) + 3
a         >>       6
b         >>       3
File.open(name = gets.chomp)
```

Las dos formas básicas de asignación en *Ruby* son:

1. Asignación de la referencia de un objeto a una variable o constante.
2. Asignación de un atributo de un objeto o elemento de referencia en la parte derecha.

2.9.3.1. Asignación paralela

Las asignaciones en paralelo de *Ruby* son efectuadas de forma eficiente, los valores asignados no son afectados por la asignación en si misma. Los valores en la parte derecha son evaluados en el orden en el cual aparecen antes de que se realice la asignación a las variables o atributos de la derecha.

```
x = 0    >>       0
a, b, c    =    x, (x += 1), (x += 1)      >>       [0,
   1, 2]
```

Cuando una asignación tiene más de un valor de la parte izquierda, la expresión de asignación devuelve un vector de los valores de los valores de la derecha. Si una asignación tiene mas valores en la izquierda que en la derecha, los valores de la izquierda sobrantes son inicializados como `nil`. Por el contrario, si una asignación contiene más valores en la derecha que en la izquierda, los valores de la derecha sobrantes son ignorados.

2.9.3.2. Asignaciones anidadas

Las asignaciones paralelas de *Ruby* propician las asignaciones anidadas. La parte de la izquierda de la asignación puede contener una lista de términos entre paréntesis. *Ruby* extrae la parte correspondiente del valor de la derecha, y lo asigna a los términos entre paréntesis, antes de continuar con la asignación del nivel superior.

```
b, (c, d), e = 1,2,3,4  >>       b == 1, c == 2, d ==
    nil,        e == 3
b, (c, d), e = [1,2,3,4]         >>       b == 1, c ==
    2, d == nil,        e == 3
b, (c, d), e = 1,[2,3],4         >>       b == 1, c ==
    2, d == 3, e == 4
b, (c, d), e = 1,[2,3,4],5       >>       b == 1, c ==
    2, d == 3, e == 5
b, (c,*d), e = 1,[2,3,4],5       >>       b == 1, c ==
    2, d == [3, 4],     e == 5
```

2.9.4. Ejecución condicional

Ruby presenta diferentes mecanismos para la ejecución condicional de código; muchos de ellos resultan similares a otros lenguajes y otros tienen algunas peculiaridades.

2.9.4.1. Expresiones booleanas

Ruby tiene una simple definición de verdad. Cualquier valor que no es `nil` o la constante `false` es verdadero. El número cero no está interpretado como un valor falso, como tampoco una cadena de caracteres de longitud cero.

2.9.4.2. Operadores booleanos

Ruby soporta todos los operadores booleanos estándar (`and`, `or`, `not`), e introduce un operador nuevo `defined?`.

`and` y `&&` Ambos operadores evalúan como `true` solo si ambos operandos son verdaderos. Además solo evalúan el segundo operando so si el primero es cierto. La única diferencia entre ellos es la precedencia, `and` tiene menos prioridad que `&&`.

`or` y `||` De forma similar, evalúan a verdadero si uno de los dos operandos son ciertos. Solo se evalúa el segundo operando si el primero es falso. Análogamente al operador `and`, la única diferencia entre `or` y `&&`, es la precedencia.[31]

`not` y `!` Retornan el opuesto de su operando. Solo se diferencian es su precedencia.

`defined?` retorna `nil` si su argumento no está definido, si no, devuelve una descripción del argumento.

[31] `and` y `or` tienen la misma precedencia, mientras que `&&` tiene una precedencia más alta que `||`.

Operadores de comparación	
`==`	Comprueba la igualdad de los operandos.
`===`	Usado para comprobar la igualdad dentro de una clausula `when` de una instrucción `case`.
`<=>`	Operador de comparación general. Devuelve `-1`, `0`, o `+1`, dependiendo si lo recibido es menor, igual o mayor, respectivamente.
`<`, `<=`, `>=`, `>`	Operadores de comparación, menor, menor o igual, mayor o igual, mayor, respectivamente.
`=~`	Patrón de coincidencia con expresión regular.
`eql?`	Verdadero si el receptor y el argumento tienen ambos el mismo valor y tipo. `1 == 1.0` es verdadero pero `1.eql?(1.0)` es falso.
`equal?`	Verdadero si el receptor y el argumento tienen el mismo id de objeto.

2.9.4.3. Expresiones If y Unless

En *Ruby*, `if` es una expresión con sintaxis similar a la instrucción `if` de otros lenguajes.

```
if aSong.artist == "Gillespie" then
  handle = "Dizzy"
elsif aSong.artist == "Parker" then
  handle = "Bird"
else
  handle = "unknown"
end

#Si se agrupan las instrucciones if en diferentes
   lineas, se puede omitir el then
if aSong.artist == "Gillespie"
  handle = "Dizzy"
elsif aSong.artist == "Parker"
  handle = "Bird"
else
  handle = "unknown"
end

#Ruby soporta el estilo de C
cost = aSong.duration > 180 ? .35 : .25
```

Ruby también tiene una forma negada de la expresión `if`, `unless`.

```
unless aSong.duration > 180 then
  cost = .25
else
  cost = .35
end
```

2.9.5. Expresión Case

En *Ruby*, `case` compara la expresión después de la palabra clave `case` con cada una de las expresiones de comparación después de la palabra clave `when`.

```
case inputLine

  when "debug"
    dumpDebugInfo
    dumpSymbols

  when /p\s+(\w+)/
    dumpVariable($1)

  when "quit", "exit"
    exit

  else
    print "Illegal command: #{inputLine}"
end
```

Análogamente a la expresión `if`, case retorna el valor de la ultima expresión ejecutada, por tanto, se necesita la palabra clave `then` si la expresión esta en la misma linea que la condición.

```
kind = case year
         when 1850..1889 then "Blues"
         when 1890..1909 then "Ragtime"
         when 1910..1929 then "New Orleans Jazz"
         when 1930..1939 then "Swing"
         when 1940..1950 then "Bebop"
         else                 "Jazz"
       end
```

2.9.6. Bucles

Ruby tiene los bucles típicos de cualquier otro lenguaje de programación.

El bucle `while` ejecuta su cuerpo cero o más veces siempre y cuando la condición es verdadera.

```
while gets
  # ...
end
```

Existe también la forma negada que excluye el cuerpo si la condición se hace cierta.

```
until playList.duration > 60
  playList.add(songList.pop)
end
```

Cuando `while` y `unless` son usados como modificadores de una instrucción `begin\end`, el código del bloque siempre se ejecutará al menos una vez, independientemente de la expresión booleana.

```

print "Hola\n" while false
begin
  print "Hasta luego\n"
end while false

#Output:
Hasta luego
```

2.9.6.1. Iteradores

Ruby no tiene un bucle `for`, al menos no del tipo que se pudieran encontrar en C o Java. *Ruby* no necesita bucles sofisticados, puesto que todo se puede implementar a través de iteradores. *Ruby* usa métodos definidos en varias clases incorporadas al lenguaje para proporcionar una funcionalidad equivalente pero menos propensión a errores.

```
3.times do
  print "Ho! "
end

#Output:
Ho! Ho! Ho!
```

```ruby
0.upto(9) do |x|
  print x, " "
end

#Output:
0 1 2 3 4 5 6 7 8 9
```

```ruby
0.step(12, 3) {|x| print x, " " }


0 3 6 9 12
```

```ruby
[ 1, 1, 2, 3, 5 ].each {|val| print val, " " }

#Output:
1 1 2 3 5
```

```ruby
loop {
  # codigo ...
}

#Se ejecutara infinitamente
```

2.9.6.2. For ... In

Este bucle `for` es diferente a los de C o Java. Este bucle pretende simplemente mejorar la sintaxis del lenguaje si no se quiere utilizar `each`. La única diferencia entre el bucle `each` y `for` es el alcance de las variables locales definidas en el cuerpo.

```ruby
for aSong in songList
  aSong.play
end

#Ruby lo traduce en:

songList.each do |aSong|
  aSong.play
end
```

```ruby
for i in ['fee', 'fi', 'fo', 'fum']
  print i, " "
```

```
end
for i in 1..3
  print i, " "
end
for i in File.open("ordinal").find_all { |l| l =~ /
   d$/}
  print i.chomp, " "
end

fee fi fo fum 1 2 3 second third
```

2.9.6.3. Break, Redo, Next

El estructuras de control de bucles `break`, `redo` y `next` permiten alterar el flujo normal de un bucle o iterador.

`break` Termina inmediatamente el bucle, y resume a la próxima instrucción después del bloque.

`redo` Repite la iteración del bucle desde el principio, pero sin revaluar la condición.

`next` Salta al final del bucle para empezar la siguiente iteración.

```
while gets
  next if /^\s*#/   # saltar comentarios
  break if /^END/   # parar al final
                    # sustituir and probar de nuevo
  redo if gsub!(/`(.*?)`/) { eval($1) }
end
```

2.9.6.4. Retry

La expresión `redo` provoca que un bucle repita la iteración actual, pero a veces, se necesita que se repita el bucle desde el principio del todo. La instrucción `retry` hace justo eso.

```
for i in 1..100
  print "Now at #{i}. Restart? "
  retry if gets =~ /^y/i
end
```

```
#Ouput:
Now at 1. Restart? n
Now at 2. Restart? y
Now at 1. Restart? n
 . . .
```

Capítulo 3

Rails

3.1. Introducción

Ruby on Rails es un framework que hace facilita el desarrollo, la implementación, y el mantenimiento de aplicaciones web. Durante unos meses después de su lanzamiento incial, *Rails* pasó de ser un "juguete" desconocido a ser un fenómeno mundial; y lo que es mucho más importante, se ha convertido en el framework más elegido para la implementación de muchísimas de las llamadas aplicaciones web 2.0.

La reputación de *Ruby on Rails* es enorme debido a una serie de razones. Primero, un gran colectivo de desarrolladores estaban frustrados con tecnologías que ellos utilizaban para crear aplicaciones web. Sin importar si usaban *Java*, *PHP*, o *.NET*, el sentimiento e idea general que tenía de su trabajo es que era demasiado difícil.

La sencillez no lo es todo. Los programadores profesionales, necesitaban la sensación de que las aplicaciones que desarrollaban aguantarían el paso del tiempo, implementado diseños y tecnologías modernas con técnicas profesionales. Naturalmente, estos desarrolladores miraron *Rails* y descubrieron que era una herramienta muy potente para el desarrollo de aplicaciones web.

Un ejemplo de ello es, todas las aplicaciones de *Rails* son implementadas usando la arquitectura modelo-vista-controlador (MVC). Los desarrolladores de *Java* están habituados a frameworks como *Taperstry* o *Struts*, que están basados en MVC. Pero *Rails* eleva la arquitectura MVC mucho más allá, cuando se desarrolla una aplicación en *Rails*, se empieza con una aplicación que funciona desde el momento cero, y que todas las piezas del código interactuan las unas con las otras de una forma estándar.

Los desarrolladores se preocupan por la implantación del sistema también. Encontraron en *Rails* una forma de implantar aplicaciones en cualquier

número de servidores con un solo comando. [6]

La última cosa a comentar respecto a este apartado es que Sam Ruby, Dave Thomas, David Heinemeier Hansson y otros muchos desarrolladores piensan que hay algo más sobre *Ruby on Rails*, algo bastante difícil de explicar, la sensación de que se están haciendo las cosas bien y como es debido. Al igual que otros muchos, yo también creo lo mismo sobre *Ruby on Rails*, "It just feel right".

3.2. Historia

David Heinemeier Hansson extrajo *Ruby on Rails* de su trabajo en la herramienta de gestión de proyectos *Basecamp* en la empresa de aplicaciones web también llamada *Basecamp*. [7] *Hansson* hizo el primer estreno de *Rails* como *open source* en Julio de 2004, pero no permitió los derechos de compartición de *commits* al proyecto hasta Febrero de 2005.[1] En Agosto de 2006, el framework alcanzó un hito cuando la empresa *Apple Inc.* anunció que exportaría *Ruby on Rails* en OS X v10.5 "Leopard". [8]

Rails versión 2.3 fue lanzado el 15 de Marzo de 2009 con nuevos desarrollos con plantillas, motores, Rack[2] y modelos de formularios anidados. Las plantillas permiten a los desarrolladores generar esqueletos de aplicaciones con gemas y configuraciones personalizadas. Los motores dan a los programadores la habilidad de reusar piezas de aplicación con rutas, vistas y modelos. Rack web server interface permiten escribir piezas de código optimizadas que se enrutan a los controladores.

El 23 de Diciembre de 2008, *Merb*, otro framework de aplicaciones web, es lanzado; y *Ruby on Rails* anuncia que trabajará con el proyecto *Merb*[3] para "traer las mejores ideas de Merb" a *Ruby on Rails* versión 3. *Merb* fue mezclado con Rails como parte del lanzamiento de *Rails* 3.0.

Rails 3.1 fue lanzado el 31 de Agosto de 2011, presentando migraciones reversibles a la base de datos, *Asset Pipeline*, *Streaming*, *JQuery* como librería por defecto de *JavaScript* y la introducción de los nuevos *CoffeeScript* y *Sass*.

Rails 3.2 nació el 20 de Enero de 2012, con un modo rápido de desarrollo y un motor de enrutamiento (Journey Engine), *Automatic Query Expalain* y *Tagged Logging*. *Rails* 3.2.x es la última versión que soporta *Ruby* 1.8.7. *Rails* 3.2.12 soporta *Ruby* 2.0.

---

[1] Equipo de desarrollo de *Ruby on Rails*, `http://rubyonrails.org/community/#core`

[2] Rack es un interfaz de se servidor web en *Ruby*. `http://rack.github.io/`

[3] Sitio web de `https://archive.org/` con enlace a la página muerta del proyecto *Merb*, `https://web.archive.org/web/20081204002911/http://merbivore.com/`

Rails 4.0 fue estrenado el 25 de Junio de 2013, introduciendo *Turbolinks*, *Live Streaming*, así como haciendo opcionales *Active Resource*, *Active Record Observer* y otros componentes, separando por gemas.

Rails 4.1 se lanzó el 8 de Abril de 2014, introduciendo *Spring*, *Variants*, *Enums* y `secrets.yml`.

Rails 4.2 nació el 19 de Diciembre de 2014, introduciendo *Active Job*, emails asincronos, *Adequate Record*, *Web Console* y claves ajenas.

3.3. Descripción técnica general

Ruby on Rails, como muchos *web framework*, usa el patrón modelo-vista-controlador (MVC) para organizar la aplicación.

En la configuración por defecto, los modelos de *Ruby on Rails* mapean una tabla en una base de datos, y en un archivo de *Ruby*.

Un controlador es un componente de *Rails* en el servidor que responde a peticiones externas del servidor web de la aplicación y determina que archivo de vista debe visualizar o renderizar. El controlador puede tener que realizar consultas a uno o más modelos directamente y pasar la información a la vista. Un controlador puede proporcionar una o varias acciones. En *Ruby on Rails*, una acción es típicamente una unidad básica que describe como se debe responder a una petición externa y especifica del navegador web. La acción o el controlador sólo está disponible y accesible si las peticiones externas se corresponden con una ruta mapeada a la petición. *Rails* fomenta el uso de rutas *RESTful*, que incluyen acciones como: `create`, `new`, `edit`, `update`, `destroy`, `show`, e `index`.

Una vista, en la configuración por defecto de *Rails*, es un archivo `.erb`, que es evaluado y convertido a *HTML* en tiempo de ejecución.

Ruby on Rails incluye herramientas que hacen las tareas comunes de desarrollo mucho más sencillas, como *scaffolding*[4]. Además, *Rails* incluye *WEBbrick*[5] y *Rake*[6], proporcionando un entorno de desarrollo básico.

Además de *WEBbrick*, *Ruby on Rails* puede ser implementado sobre otros servidores web como *Mongrel*, *Lighttpd*, *Apache*, *Cherokee*, *Hiawatha*, *nginx*, así como muchos otros.

En *Ruby on Rails* es notable el uso que se hace de las librerías de de

[4] *Scaffolding* construye automáticamente algunos modelos, vistas y controladores necesarios para una aplicación web básica.

[5] *WEBbrick* es un servidor web sencillo implementado y distribuido en *Ruby*.

[6] Rake es un software de manejo y automatización de tareas, que es distribuido como una gema.

JavaScript, *Prototype*[7] y *Script.aculo.us*[8] para acciones de script de *Ajax*.

Ruby on Rails comenzó utilizando *SOAP* para servicios web; siendo reemplazado posteriormente por servicios web *RESTful*.

Ruby on Rails 3.0 utiliza una técnica llamada *Unobtrusive JavaScript* para separar funcionalidades o lógicas de la estructura de una página web. *JQuery* is totalmente soportado como reemplazo de *Prototype* como libreria de *JavaScript* por defecto en *Rails* 3.1, reflejando la tendencia de la industria moviéndose hacia *JQuery*. Adicionalmente, en *Rails* 3.1, *CoffeeScript* es introducido como lenguaje *JavaScript* por defecto.

Ruby on Rails es frecuentemente instalado usando *RubyGems*, un gestor de paquetes, que es incluido con las versiones actuales de *Ruby*.

Rails es montado frecuentemente con una base de datos como *MySQL* o *PostgreSQL*, y con un servidor web como *Apache* con el módulo *Phusion Passenger*.

3.4. Filosofía y diseño

Ruby on Rails enfatiza en la idea de "Convención sobre Configuración (CoC)", y el principio de "Don't repeat yourself (DRY)". "Convención sobre Configuración (CoC)" significa que el desarrollador sólo necesita especificar los aspectos no convencionales de la aplicación. Por ejemplo, si se crea una clase llamada `User`, se creará automáticamente una tabla en la base de datos correspondiente llamada `users`.

"Don't repeat yourself (DRY)" propone que la información está emplazada en un único lugar. Por ejemplo, las funciones de nuevo usuario y actualización de usuario, pueden utilizar el mismo archivo de vista del formulario de usuarios.

Otra filosofía importante de *Rails* es "Fat models, skinny controllers" que aconseja que casi toda la lógica de la aplicación debe ser implementada dentro del modelo dejando el controlador lo más simple y ligero posible.

[7] *Prototype JavaScript Framework* es un framework de *JavaScript* creado como parte del sustento para el soporte de Ajax en *Ruby on Rails*.

[8] *Script.aculo.us* es una libreria de *JavaScript* construida en el framework *Prototype JavaScript Framework*, que provee efectos visuales dinámicos y elementos de interfaz de usuario a través del *Document Object Model (DOM).*

3.5. Creando un nuevo proyecto de Rails

3.5.1. Instalando Rails

Hay numerosas formas de instalar *Ruby on Rails* pero en este episodio se mostrará el método que considero mejor que es de *GoRails*.[9] La ventaja del método frente a la forma de la documentación oficial de *Rails* es que llevando a cabo una serie de pasos adicionales, no aparecen problemas de permisos y usuarios; o problemas con el entorno del sistema operativo.

Se explicarán brevemente los pasos a seguir para instalar *Rails*, siguiendo la guía de *GoRails*. Dicha instalación se realiza en el sistema operativo Ubuntu 14.04 y sobre la base de datos que he elegido para la aplicación, que es PostgreSQL, pero puede ser instalada de forma similar en otras distribuciones de Linux e implantando otras bases de datos como MySQL o SQLite3.

3.5.1.1. Instalando Ruby

El primer paso es instalar las dependencias de *Ruby*.

```
sudo apt-get update
sudo apt-get install git-core curl zlib1g-dev build-
   essential libssl-dev libreadline-dev libyaml-dev
   libsqlite3-dev sqlite3 libxml2-dev libxslt1-dev
   libcurl4-openssl-dev python-software-properties
   libffi-dev
```

Después de instalar las dependencias de *Ruby*, procedemos a instalarlo mediante `rbenv`[10], que un sistema que prepara el entorno para las aplicaciones de *Ruby*.

Instalando con `rbenv` es un simple proceso de dos etapas. Primero se instala `rbenv` clonándolo desde su repositorio de GitHub y exportando variables para la *shell* de Linux. El siguiente paso es instalar la *build* de *Ruby*

```
cd
git clone git://github.com/sstephenson/rbenv.git .
   rbenv
echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.
   bashrc
echo 'eval "$(rbenv init -)"' >> ~/.bashrc
```

[9]Documentación completa para la instalación de *Rails* en Ubuntu 14.04 disponible en: `https://gorails.com/setup/ubuntu/14.04`

[10]Documentación completa de `rbenv` disponible en GitHub en la url `https://github.com/rbenv/rbenv`.

```
exec $SHELL

git clone git://github.com/sstephenson/ruby-build.
   git ~/.rbenv/plugins/ruby-build
echo 'export PATH="$HOME/.rbenv/plugins/ruby-build/
   bin:$PATH"' >> ~/.bashrc
exec $SHELL

git clone https://github.com/sstephenson/rbenv-gem-
   rehash.git ~/.rbenv/plugins/rbenv-gem-rehash
```

Instalación de la *build* de *Ruby*.

```
rbenv install 2.2.3
rbenv global 2.2.3
ruby -v
```

Después lo configuramos para que RubyGems no instale la documentación para cada paquete localmente e instalamos la gema Bundler, que en gestor de gemas que dada una lista de gemas, automáticamente descarga e instala dichas gemas.

```
echo "gem: --no-ri --no-rdoc" > ~/.gemrc
gem install bundler
```

3.5.1.2. Instalando Rails

Dado que *Rails* es exportado con gran cantidad de dependencias, será necesario instalar un entorno JavaScript como NodeJS. Esto perimte usar CoffeeScript y el Asset Pipeline en *Rails* que combina y minimiza los archivos JavaScript para proporcionar un entorno de producción más rápido e eficiente.

Se instalará NodeJS a través de su repositorio oficial.

```
curl -sL https://deb.nodesource.com/setup_4.x | sudo
    -E bash -
sudo apt-get install -y nodejs
```

Después instalamos *Rails*.

```
gem install rails -v 4.2.4
```

Utilizando `rbenv`, se necesita ejecutar el siguiente comando para hacer *Rails* ejecutable.

```
rbenv rehash
```

En este punto, *Rails* debería estar instalado. Para comprobar que se ha instalado correctamente, ejecutamos el siguiente comando:

```
rails -v
# Rails 4.2.4
```

3.5.1.3. Configurando PostgreSQL

Para instalar la base de datos PostgreSQL, debe añadirse un nuevo repositorio para instalar la versión de PostgreSQL más reciente.

```
sudo sh -c "echo 'deb http://apt.postgresql.org/pub/
   repos/apt/ precise-pgdg main' > /etc/apt/sources.
   list.d/pgdg.list"
wget --quiet -O - http://apt.postgresql.org/pub/
   repos/apt/ACCC4CF8.asc | sudo apt-key add -
sudo apt-get update
sudo apt-get install postgresql-common
sudo apt-get install postgresql-9.3 libpq-dev
```

La instalación de PostgreSQL no proporciona un usuario, por lo que la siguiente tarea será crear un usuario con permisos de creación de bases de datos.

```
sudo -u postgres createuser [NOMBRE_USUARIO] -s

# Para establecer una password al usuario se puede
   hacer lo siguiente:
sudo -u postgres psql
postgres=# \password chris
```

3.5.1.4. Pasos finales

Puesto que la base de datos es PostgreSQL, se debe editar el archivo `config/database.yml` para establecer el usuario y su password, creado anteriormente. Después creamos una aplicación *Rails* a modo de test para comprobar que el *Ruby on Rails* está configurado correctamente.

```
rails new test -d postgresql
cd test
rake db:create #Creacion de la base de datos
rails server
```

Después visitamos `http://localhost:3000` para comprobar que el sistema funciona correctamente.

3.5.2. Creando la primera aplicación

Rails tiene implementados un gran número de *scripts* llamados generadores que son diseñados para hacer el desarrollo más sencillo, creando todo lo necesario para empezar a trabajar en una tarea más especifica. Uno de los generadores es el generador de nueva aplicación, el cual proporciona los cimientos y el esqueleto necesario de una aplicación nueva de *Rails* sin que el desarrollador tenga que escribirla.[11]

```
$ rails new hola
```

Este comando crea una aplicación de *Rails* llamada Hola en un directorio con nombre `hola` e instala las dependencias de las gemas que son mencionadas en el archivo `Gemfile` usando el comando `bundle install`.

El directorio creado contendrá una serie de archivos y carpetas autogeneradas que serán la estructura o la arquitectura de una aplicación de *Rails*.

[11] Para comprobar todas las opciones de linea de comandos que una aplicación de *Rails* acpeta cuando se ejecuta `rails new -h`.

Archivo/Directorio	Propósito
app	Contiene los controladores, modelos, vistas, helpers, mailers y assests para la aplicación. Es la carpeta donde más se trabaja.
bin	Contiene el script que inicia la aplicación y otros otros script para implantar, o ejecutar la aplicación.
config	Configura las rutas de la aplicación, la base de datos y otras cosas relativas a la configuración.
config.ru	Configuración de *Rack* para servidores basados en *Rack*.
db/	Contiene el esquema de la base de datos actual y las migraciones de la base de datos.
Gemfile Gemfile.lock	Estos archivos permite especificar que dependencias de las gemas son necesarias para la aplicación de *Rails*. Esos archivos son utilizados por la gema *Bundler*. [12]
lib/	Módulos ampliados de la aplicación.
log	Archivos de log de la aplicación.
public/	Es la única carpeta que se puede ver desde el exterior tal y como es. Contiene archivos estáticos y recursos compilados.
Rakefile	Este archivo localiza y carga las tareas que pueden ser ejecutadas desde la linea de comandos. Las definiciones de las tareas son definidas a través de los componentes de *Rails*. En vez de modificar el Rakefile, se debe añadir las tareas propias añadiéndolas al directorio lib/task de la aplicación.
README.rdoc	Es un breve manual de la aplicación. Se debe editar para decir a otros lo que la aplicación hace y como está configurada.
test/	Test unitarios y todo lo relativo al testeo son incluidos en este directorio.
tmp/	Directorio de archivos temporales como la cache, pid o los archivos de sesión.
vendor/	El lugar para el código third-party.

3.6. Ejecutando Rails

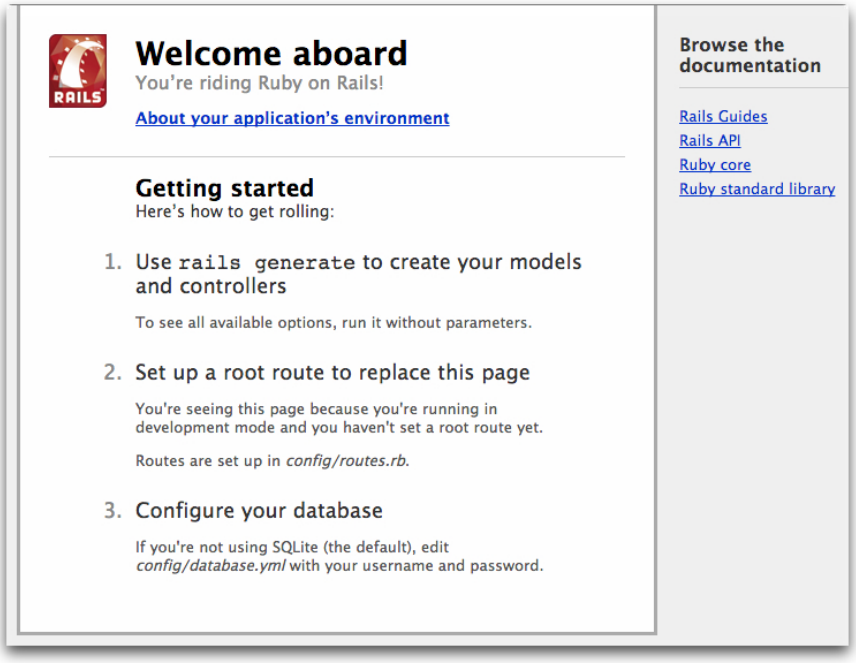
Una vez creada una aplicación de *Rails* con el generador anterior, podemos ejecutarla directamente.

3.6.1. Arrancando el servidor Web

Para arrancar el servidor web de *Rails*, debemos ejecutar el siguiente comando en el directorio de la aplicación que se ha creado.

```
$ bin/rails server
```

Esto producirá que WEBrick, el servidor web distribuido con *Ruby* por defecto se inicie. Para ver la aplicación en acción, se debe abrir una ventana del navegador web y navegar a la dirección `http://localhost:3000`.[13] El navegador mostrará la página de información por defecto de *Rails*.



La página de "Welcome aboard" es un test para una nueva aplicación de *Rails*. Con ello se asegura que *Ruby on Rails* ha sido configurado de

[13] Naturalmente si estamos ejecutando el servidor web y el navegador web en la misma máquina.

forma correcta. También se puede acceder al link "About your application's environment" para ver un resumen del entorno de la aplicación.

Para detener el servidor web, hay que presionar las teclas Ctrl+C en la ventana del terminal en la que el servidor web está siendo ejecutado.

3.7. Modelos

Como se ha dicho varias veces, *Rails* usa el patrón de arquitectura MVC. Los modelos son la capa o parte del sistema responsable de representar la lógica de negocio. Los modelos de *Rails* son diseñados y representados mediante Active Record.

3.7.1. Active Record

Active Record facilita la creación y uso de los objetos negocio cuyos datos precisan de ser persistentes en la base de datos.

Active Record es un objeto que envuelve una fila de una tabla o vista de una base de datos, encapsula el acceso a la base de datos, y añade lógica de negocio en esos datos envueltos. Active Record conlleva tanto tantos como comportamiento y muchos de estos datos deben de ser persistentes en la base de datos. Active Record añade acceso lógico a los datos en el dominio del objeto, de esta manera, todo el mundo sabe como leer y escribir los datos en la base de datos. [9]

Active Record es una descripción de ORM (Object-Relational-Mapping), que es una tecnica que conecta objetos de la aplicación con las tables de una base de datos relacional. Usando ORM, las propiedades de las relaciones de los objetos en una aplicación pueden ser facilmente almacenados y recuperados desde la base de datos, sin escribir consultas SQL directamente.

Active Record proporciona varios mecanismos propios de una técnica ORM:

1. Representar modelos y sus datos.
2. Representar asociaciones entre los modelos.
3. Representar jerarquías de herencia a través de los modelos relacionados.
4. Validar modelos antes de ser escritos de forma persistente a la base de datos.
5. Realizar operaciones en la base de datos con una sintaxis y un estilo de orientación a objetos.

3.7.2. Convención sobre configuración en Active Record

A diferencia de otros lenguajes de programación o frameworks, en *Rails* no es necesario escribir mucha configuración cuando se trabaja con modelos de Active Record.

Por defecto, Active Record utiliza convenciones en los nombres para entender como un modelo y la la tabla de la base de datos deberían ser creados. *Rails* pluraliza los nombres de las clases para encontrar la respectiva tabla de la base de datos. [14]

Active Record también utiliza convenciones para los nombres de las columnas de la base de datos, dependiendo del propósito de las columnas.

Claves ajenas Los campos deben ser llamamos siguiendo el patrón nombre de la tabla singularizado e id. Por ejemplo, `user_id` de la tabla `users`.

Claves primarias Por defecto, Active Record utilizará una columna de tipo en tero llamada `id` como clave primaria.

3.7.3. Creando modelos Active Record

Para crear modelos de Active Record, lo único que se precisa hacer es asignarlo como subclase de `ActiveRecord::Base`

```
class Product < ActiveRecord::Base
end
```

Esto permite crear un modelo `Product` que será referenciado a la tabla `products` de la base de datos. De esta manera, se posibilita la forma de referenciar columnas de cada fila de la tabla de la base de datos con los atributos de las instancias del modelo.

Suponiendo que la tabla productos ha sido creada utilizando la siguiente sentencia SQL,

```
CREATE TABLE products (
   id int(11) NOT NULL auto_increment,
   name varchar(255),
   PRIMARY KEY  (id)
);
```

Será posible escribir código como el siguiente:

[14] La pluralización de *Rails* es muy potente, siendo capaz de pluralizar o singularizar palabras regulares e irregulares.

```
p = Product.new
p.name = "El Quijote"
puts p.name # "El Quijote"
```

3.7.4. CRUD

En computación CRUD es el acronimo de los cuatro verbos que usamos para operar con datos: Create, Read, Update y Delete. Active Record automaticamente crea estos métodos para permitir a la aplicación leer y manipular los datos almacenados en sus tablas.

3.7.4.1. Create

Los objetos de Active Record pueden ser creados desde un hash o manualmente establecidos después de su creación. El método `new` devuelve un objeto nuevo mientras que `create` retornará el objeto y lo almacenará en la base de datos.

El siguiente fragmento de código utiliza el método `create` por lo que creará el usuario y lo guardará como un nuevo registro en la base de datos.

```
user = User.create(name: "Antonio", occupation: "
   Profesor")
```

Por el contrario, en este otro fragmento de código el usuario es creado pero no es guardado en la base de datos.

```
user = User.new
user.name = "Antonio"
user.occupation = "Profesor"
```

Si además se deseara guardar el usuario, se deberá utilizar el método `user.save`.

3.7.4.2. Read

Active Record proporciona una API potente para acceder a datos de la base de datos. A continuación se exponen una serie de ejemplos que ilustran los diferentes métodos facilitados por Active Record.

```
# Devuelve una colección con todos los usuarios.
users = User.all
```

```
# Devuelve el primer usuario.
user = User.first
```

```
# Retorna el primer usuario con nombre Antonio
david = User.find_by(name: 'Antonio')
```

```
#Encuentra todos los usuarios llamados Antonio que
   son profesores ordenados por fecha de creación de
    forma inversa.
users = User.where(name: 'Antonio', occupation: '
   Profesor').order(created_at: :desc)
```

3.7.4.3. Update

Una vez que un objeto de Active Record ha sido recuperado, sus atributos pueden ser modificados y pueden ser guardados en la base de datos.

```
user = User.find_by(name: 'David')
user.name = 'Antonio'
user.save
```

```
# Otra forma de realizarlo
user = User.find_by(name: 'David')
user.update(name: 'Antonio')
```

3.7.4.4. Delete

Análogamente, un objeto de Active Record recuperado puede ser destruido, lo que le borra de la base de datos.

```
user = User.find_by(name: 'Antonio')
user.destroy
```

3.7.5. Migraciones de la base de datos

Las migraciones de *Rails* son la forma mas conveniente de alterar el esquema de la base de datos de una manera consistente y sencilla. Las migraciones usan un Ruby DSL (Domain Specific Language) por lo que no hace falta escribir ni una sola instrucción SQL, permitiendo que el esquema de la base de datos y los cambios ser independientes de la base de datos.

Se puede pensar que una migración es una nueva "versión" de la base de datos. Active Record actualiza el archivo `db/schema.rb` para mantener la estructura actualizada de la base de datos.

Un ejemplo de una migración:

```
class CreateProducts < ActiveRecord::Migration
  def change
    create_table :products do |t|
      t.string :name
      t.text :description

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

Las migraciones pueden ser lanzadas hacia "delante" o hacia "detrás", esto significa que si, por ejemplo, la migración anterior es lanzada cuando no existe la tabla `products` en la base de datos, entonces, la tabla `products` con las columnas `name`, `descripción`, y `timestamps` será creada. Por el contrario, si lanzamos la migración cuando la tabla existe en la base de datos, lo que se producirá es un efecto de *rollback*.

En las bases de datos que soportan transacciones con instrucciones que modifiquen el esquema, las migraciones son envueltas en una transacción, de forma que si una migración falla, se hará un *rollback* automáticamente de la transacción.[15]

Algunas migraciones contienen ciertas instrucciones que Active Record no sabe como hacer el inverso de dicha operación. El ejemplo más típico es el cambio de nombre.

```
class ChangeProductsPrice < ActiveRecord::Migration
  def up
    change_table :products do |t|
      t.change :price, :string
    end
  end

  def down
    change_table :products do |t|
      t.change :price, :integer
    end
  end
end
```

[15]Si la base de datos no soporta transacciones y la migración falla, los cambios para hacer el rollback deben ser realizados manualmente.

3.7.5.1. Creando una migración simple

Las migraciones son almacenadas en ficheros en el directorio `db/migrate`, un fichero por cada migración. El nombre del fichero contiene el UTC timestamp que es la fecha en formato YYYYMMDDHHMMSS (año, mes, día, hora, minuto, segundo) seguido de guión bajo y el nombre de la migración. *Rails* usa el UTC timestamp para determinar el orden en el que las migraciones deben ser ejecutadas.

Se puede generar una migración de la siguiente forma:

```
$ bin/rails generate migration
   AddPartNumberToProducts
```

Ese comando creará un nuevo archivo en `db/migrate` que contendrá:

```
class AddPartNumberToProducts < ActiveRecord::
   Migration
  def change
  end
end
```

Si el nombre de la migración sigue el patrón AddXXXToYYY o RemoveXXXFromYYY seguido de una lista de nombres de columnas y sus tipos, entonces la migración será creada con las instrucciones `add_column` y `remove_column`, respectivamente.

```
$ bin/rails generate migration
   AddPartNumberToProducts part_number:string
```

```
lass AddPartNumberToProducts < ActiveRecord::
   Migration
  def change
    add_column :products, :part\_number, :string
  end
end
```

Generadores de Modelos

El modelo y los *scaffold generators* crean las migraciones apropiadas para el nuevo modelo. Esta migración contendrá instrucciones para crear la tabla de la base de datos.

```
$ bin/rails generate model Product name:string
   description:text
```

El generador anterior creará el nuevo modelo y una migración nueva con el siguiente contenido:

```
class CreateProducts < ActiveRecord::Migration
  def change
    create_table :products do |t|
      t.string :name
      t.text :description

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

3.7.5.2. Sintaxis de las migraciones

Tablas

El método `create_table` es el más fundamental y casi siempre es creado por un generador de modelos o un *scaffold generator*. Implícitamente crea una columna `id` que será la clave primaria.

```
create_table :products do |t|
  t.string :name
end
```

Join Table

El método de migración `create_join_table` crea una tabla join para una relación de muchos a muchos o HABTM.[16]

```
create_join_table :products, :categories
```

```
create_join_table :products, :categories do |t|
  t.index :product_id
  t.index :category_id
end
```

Modificación de columnas

Los métodos de modificación de columnas son los siguientes:

```
change_column :products, :part_number, :text
```

```
add_column :products, :description, :string
```

```
remove_column :products, :description
```

[16]Has and belongs to many

Modificadores de columnas Los modificadores de columnas pueden aplicados cuando se crea o se modifica una columna.

`limit` Establece el tamaño máximo de los campos de tipo `string / text / binary / integer`

`precision` Define la precisión para los campos de tipo `decimal`, representando el numero total de dígitos en el número.

`scale` Define la escala para los campos de tipo `decimal`, representando el número de dígitos después del punto decimal.

`polymorphic` Este parámentro añade una columna del tipo `type` para las asociaciones `belongs_to`.

`null` Habilita o inhabilita valores de tipo `NULL` en la columna.

`default` Permite establecer un valor por defecto en la columna.

`index` Añade un índice a la columna.

`required` Añade `required: true` para las asociaciones `belongs_to` y `null: false` a la columna en la migración.

Claves ajenas

Las claves ajenas ayudan a mantener la integridad referencial de una base de datos.

```
add_foreign_key :articles, :authors
```

Añade la clave ajena, introduciendo la columna `author_id` en la tabla `articles`. La clave referencia a la columna `id` en la tabla `authors`.

Active Record solo soporta claves ajenas de una sola columna. Para claves ajenas compuestas se precisa `execute` y `structure.sql`.

Hay varias opciones para eliminar claves ajenas:

```
# Active Record buscará el nombre de columna.
remove_foreign_key :accounts, :branches

#Elimina la clave ajena de una columna especifica.
remove_foreign_key :accounts, column: :owner_id

# Elimina la clave ajena por su nombre.
remove_foreign_key :accounts, name: :special_fk_name
```

Ejecutar SQL

Si Active Record no fuera suficiente, se pueden ejecutar instrucciones directamente en lenguaje SQL mediante el método `execute`.[17]

```
Product.connection.execute('UPDATE `products` SET `
    price`=`free` WHERE 1')
```

Rollback

Como se ha mencionado anteriormente, *Rails* soporta rollbacks de las migraciones a la base de datos, porporcionando diversos métodos:

Método `change` El método `change` es el método primario y por defecto de las migraciones. Funciona en la mayoría de los casos que son aquellos en los cuales Active Record sabe cual es la migración inversa. El método `change` soporta las siguientes definiciones:

1. add_column
2. add_index
3. add_reference
4. add_timestamps
5. add_foreign_key
6. add_table
7. add_join_table
8. drop_table (si se proporciona un bloque con el contenido)
9. drop_join_table (si se proporciona un bloque con el contenido)
10. remove_timestamps
11. rename_column
12. rename_index
13. rename_reference
14. rename_table

[17]Documentación relacionada en la API de *Ruby on Rails*, `http://api.rubyonrails.org/classes/ActiveRecord/ConnectionAdapters/SchemaStatements.html`

Método `reversible` Otros métodos o algunas migraciones complejas requieren procesado que Active Record no sabe como hacer la operación inversa. Para solucionarlo, el método `reversible` permite especificar que se debe de hacer cuando se ejecuta una migración y cuando se revierte una migración.

```
class ExampleMigration < ActiveRecord::Migration
  def change
    create_table :distributors do |t|
      t.string :zipcode
    end

    reversible do |dir|
      dir.up do
        # add a CHECK constraint
        execute <<-SQL
          ALTER TABLE distributors
            ADD CONSTRAINT zipchk
              CHECK (char_length(zipcode) = 5) NO
                 INHERIT;
        SQL
      end
      dir.down do
        execute <<-SQL
          ALTER TABLE distributors
            DROP CONSTRAINT zipchk
        SQL
      end
    end

    add_column :users, :home_page_url, :string
    rename_column :users, :email, :email_address
  end
end
```

En lo casos en los que una migración sea totalmente irreversible, como cuando se destruyen algunos datos, se puede lanzar una excepción de `ActiveRecord::IrreversibleMigration` en el bloque `down`, de forma que si alguien intenta realizar la migración, un mensaje de error será emitido.

Método `up/down` Además de el método reversible, existe el método `up/down` que similar funcionalmente pero la sintaxis es distinta.

```
class ExampleMigration < ActiveRecord::Migration
  def up
    create_table :distributors do |t|
      t.string :zipcode
    end

    # add a CHECK constraint
    execute <<-SQL
      ALTER TABLE distributors
        ADD CONSTRAINT zipchk
        CHECK (char_length(zipcode) = 5);
    SQL

    add_column :users, :home_page_url, :string
    rename_column :users, :email, :email_address
  end

  def down
    rename_column :users, :email_address, :email
    remove_column :users, :home_page_url

    execute <<-SQL
      ALTER TABLE distributors
        DROP CONSTRAINT zipchk
    SQL

    drop_table :distributors
  end
end
```

3.7.5.3. Ejecución de migraciones

Rails proporciona un conjunto de tareas Rake para ejecutar migraciones. La migración sencilla ejecuta el método `change` o el método `up` para todas las migraciones que aún no han sido ejecutadas. Si no hay migraciones, estas se ejecutarán en base al orden de creación de su UTC timestamp.

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

Si se especifica una versión, Active Record ejecutará las migraciones requeridas (`change`, `up`, `down`) hasta que se alcance la versión especificada.

```
$ bin/rake db:migrate VERSION=20150906120000
```

Rollback

Un rollback habitual es volver a la migración anterior debido a un error, por ejemplo.

```
$ bin/rake db:rollback
```

Mediante el parámetro `STEP`, especificamos cuantas migraciones se deben revertir.

```
$ bin/rake db:rollback STEP=3
```

Si se desea revertir una o varias migraciones y luego migrar otra vez, existe un atajo:

```
$ bin/rake db:migrate:redo STEP=3
```

Por último, y como consecuencia lógica de la funcionalidad cuando se especifica una version concreta, si se pretende revertir hasta cierto punto o momento, se debe especificar con el parámetro `VERSION`.

```
$ bin/rake db:migrate VERSION=20150906120000
```

Otras tareas Rake

Otras tareas y operaciones con Rake frecuentes son las siguientes:

`rake db:setup` Esta tarea crea la base de datos, carga el esquema y la inicializa con los datos semilla.

`rake db:drop` Esta operación destruye la base de datos y el esquema.

`rake db:reset` Esta operación es similar a realizar `rake db:drop` y después `rake db:setup`.[18]

Por defecto, `rake db:migrate` es una instrucción que siempre se ejecutará en modo desarrollo (`development`). Si se desea ejecutar migraciones en otro entorno que no sea el de desarrollo se puede especificar usando la variable `RAILS_ENV` mientras se ejecuta el comando.

```
$ bin/rake db:migrate RAILS_ENV=test
```

```
$ bin/rake db:rollback RAILS_ENV=production
```

[18]Esta operación no es lo mismo que ejecutar todas las migraciones. Sólo usará los contenidos del archivo `schema.rb` actual. Si una migración no puede ser revertida, `rake db:reset` no será útil probablemente.

Migraciones y datos semilla

Se pueden usar migraciones para añadir datos a la base de datos:

```
class AddInitialProducts < ActiveRecord::Migration
  def up
    5.times do |i|
      Product.create(name: "Product ##{i}",
         description: "A product.")
    end
  end

  def down
    Product.delete_all
  end
end
```

Sin embargo, *Rails* tiene una funcionalidad de semillas que será usada para poblar bases de datos. Se trata añadir código al archivo `db/seeds.rb` y después, ejecuta el comando `rake db:seed`.

```
5.times do |i|
  Product.create(name: "Product ##{i}", description:
      "A product.")
end
```

3.7.6. Validaciones

3.7.6.1. Introducción

Validación es el proceso que garantiza que un dato o conjunto de datos cumple con los requisitos y especificaciones necesarias para ser introducido en el sistema de forma persistente.

```
class Person < ActiveRecord::Base
  validates :name, presence: true
end

Person.create(name: "Thomas A. Anderson").valid? #
   => true
Person.create(name: nil).valid? # => false
```

Las validaciones a nivel de modelo son la mejor forma en *Rails* para asegurar que los datos datos que se guardan en la base de datos son correctos.

Además de validar los datos a nivel de modelo, existen otras formas de validar los datos:

1. Restricciones y procedimientos almacenados que hacen la validación dependiente de la base de datos por lo que el testeo y el mantenimiento es más complicado. Sin embargo, si la base de datos es utilizada por otras aplicaciones, es ventajoso poner las restricciones a nivel de la base de datos.

2. Las validaciones en el cliente pueden ser útiles pero no fiables si se usan por si mismas.

3. Las validaciones al nivel del controlador son difíciles de testear y mantener. Siguiendo una de las filosofias de *Ruby on Rails*, lso controladores deben ser simples en cuanto a complejidad mientras que los modelos deben complejos.

Hay muchas formas de cambiar los estados de los objetos en la base de datos. Algunos métodos implican la ejecución de disparadores de validación, pero otros no. Por tanto, es posible guardar un objeto inválido en la base de datos si no se trata con cuidado.

Los siguientes métodos lanzan validaciones. La diferencia es que los que tienen exclamación lanzan una excepción si el objeto es inválido mientras que los otros retornan false (`save` y `update`) o el objeto (`create`).

1. `create`

2. `create!`

3. `save`

4. `save!`

5. `update`

6. `update!`

Los siguientes métodos saltan las validaciones, y por tanto, guardarán el objeto en la base de dato independientemente de su validez:

1. `decrement!`

2. `decrement_counter`

3. `increment!`

4. `increment_counter`
5. `toggle!`
6. `update_all`
7. `update_attribute`
8. `update_column`
9. `update_columns`
10. `update_counters`
11. Además `save` también tiene la habilidad de saltar validaciones si utiliza con el argumento `validate: false`.

Para verificar si un objeto es válido o inválido, *Rails* usa el método `valid?`.

```
class Person < ActiveRecord::Base
  validates :name, presence: true
end

Person.create(name: "Thomas A. Anderson").valid? #
   => true
Person.create(name: nil).valid? # => false
```

Un objeto instanciado con el método `new` no reportará errores de validación incluso si es inválido debido a que las validaciones no son disparadas cuando se utiliza el método `new`.

```
class Person < ActiveRecord::Base
  validates :name, presence: true
end

>> p = Person.new
# => #<Person id: nil, name: nil>
>> p.errors.messages
# => {}

>> p.valid?
# => false
>> p.errors.messages
# => {name:["can't be blank"]}
```

```
>> p = Person.create
# => #<Person id: nil, name: nil>
>> p.errors.messages
# => {name:["can't be blank"]}

>> p.save
# => false

>> p.save!
# => ActiveRecord::RecordInvalid: Validation failed:
    Name can't be blank

>> Person.create!
# => ActiveRecord::RecordInvalid: Validation failed:
    Name can't be blank
```

Para verificar si un atributo en particular de un objeto es válido, se puede utilizar el método `errors[:attribute]`. El método retorna un vector con todos los errores del atributo `:attribute`. Si no hay errores en dicho atributo, un array vació será retornado.

```
class Person < ActiveRecord::Base
  validates :name, presence: true
end

>> Person.new.errors[:name].any? # => false
>> Person.create.errors[:name].any? # => true
```

3.7.6.2. Procedimientos de validación

Active Record proporciona un conjunto de procedimientos de validación que pueden ser directamente usados en las clases. Estos procedimientos proporcionan reglas comunes de validación. Siempre que una validación falla, un mensaje de error es añadido a la colección de errores del objeto con un mensaje asociado al atributo validado.

`acceptance`
Éste método valida que una checkbox en el interfaz de usuario ha sido seleccionada cuando el formulario ha sido enviado. El mensaje de error por defecto es "must be accepted".

```
class Person < ActiveRecord::Base
  validates :terms_of_service, acceptance: true
end
```

`validates_associated`

El método `validates_associated` debe ser utilizado cuando el modelo tiene asociaciones con otros modelos que también necesitan ser validados.[19] El mensaje de error por defecto del método es "is invalid".

```
class Library < ActiveRecord::Base
  has_many :books
  validates_associated :books
end
```

`confirmation`

Éste procedimiento se puede utilizar cuando dos campos de texto reciben exactamente el mismo contenido como en una validación de una contraseña. El mensaje de error por defecto es "doesn't match confirmation".

```
#Modelo
class Person < ActiveRecord::Base
  validates :email, confirmation: true
end

#Vista
<%= text_field :person, :email %>
<%= text_field :person, :email_confirmation %>
```

`exclusion`

El procedimiento `exclusion` valida que los valores de los atributos no estén incluidos en un conjunto dado. Éste método, además, incorpora una opción `:in` que recibe un conjunto de valores que no serán aceptados por el atributo al ser validado.[20] El mensaje de error es ìs reserved".

```
class Account < ActiveRecord::Base
  validates :subdomain, exclusion: { in: %w(www us
     ca jp),
    message: "%{value} is reserved." }
```

[19]Es importante no utilizar el método `validates_associated` en ambos extremos de la asociación, puesto que se llamarán unos a otros creando un bucle infinito.

[20]La opción `:in` posee un alias `:within` que realiza la misma funcionalidad.

```
end
```

`format`
El método `format` valida los valores de los atributos probando si coinciden o no con una expresión regular dada con la opción `:with` o `:without`, respectivamente. Por defecto, su mensaje de error es "is invalid".

```
class Product < ActiveRecord::Base
  validates :legacy_code, format: { with: /\A[a-zA-Z
     ]+\z/,
    message: "only allows letters" }
end
```

`inclusion`
Éste procedimiento es inverso al método `exclusion` y por tanto, valida que los valores de los atributos son incluidos en un conjunto dado. De la misma manera que `exclusion`, éste método posee las opciones `:in` o `:within`.

```
class Coffee < ActiveRecord::Base
  validates :size, inclusion: { in: %w(small medium
     large),
    message: "%{value} is not a valid size" }
end
```

`length`
El método `length` valida que los valores de los atributos tengan una longitud concreta. Provee una serie de opciones, por lo que se puede especificar constantes de longitud de diversas formas.

`:minumum` El atributo no puede ser menor que la longitud especificada.

`:maximum` El atributo no puede ser mayor que la longitud especificada.

`:in` **o** `:within` El atributo debe estar incluido en el rango de la longitud especificada.

`:is` El atributo deber ser igual que la longitud especificada.

```
class Essay < ActiveRecord::Base
  validates :content, length: {
    minimum: 300,
```

```
    maximum: 400,
    tokenizer: lambda { |str| str.split(/\s+/) },
    too_short: "must have at least %{count} words",
    too_long: "must have at most %{count} words"
  }
end
```

`numericality`
El procedimiento valida que los atributos solo contengan valores numéricos. Por defecto, acepta cualquier valor numérico pero puede ser restringido a sólo enteros o flotantes. El error por defecto es "is not a number".

`:greater_than` Especifica que el valor deber ser mayor que el valor proporcionado.

`:greater_than_or_equal_to` Especifica que el valor deber ser mayor o igual que el valor proporcionado.

`:equal_to` Especifica que el valor deber ser igual que el valor proporcionado.

`:less_than` Especifica que el valor deber ser menor que el valor proporcionado.

`:less_than_or_equal_to` Especifica que el valor deber ser menor o igual que el valor proporcionado.

`:odd` Especifica que el valor debe ser impar.

`:even` Especifica que el valor debe ser par.

`presence`
Éste método valida que los atributos especificados con él no son vacíos.

```
class Person < ActiveRecord::Base
  validates :name, :login, :email, presence: true
end
```

También puede ser utilizado para asegurar que una asociación es presente.

```
class LineItem < ActiveRecord::Base
  belongs_to :order
  validates :order, presence: true
end
```

```ruby
class Order < ActiveRecord::Base
  has_many :line_items, inverse_of: :order
end
```

`absence`
De forma contraria al método `presence`, éste procedimiento asegura los atributos especificados no son presentes.

```ruby
class Person < ActiveRecord::Base
  validates :name, :login, :email, absence: true
end
```

Analogamente al procedimiento `presence`, también puede ser utilizado para asegurar asociaciones.

```ruby
class LineItem < ActiveRecord::Base
  belongs_to :order
  validates :order, absence: true
end

class Order < ActiveRecord::Base
  has_many :line_items, inverse_of: :order
end
```

`uniqueness`
El procedimiento `uniqueness` valida que los valores de atributo son únicos antes de que el objeto sea guardado en la base de datos. El mensaje de error por defecto es "has already been taken".

```ruby
class Account < ActiveRecord::Base
  validates :email, uniqueness: true
end
```

La opción `:scope` permite especificar otros atributos que son utilizados para limitar la validación

```ruby
class Holiday < ActiveRecord::Base
  validates :name, uniqueness: { scope: :year,
    message: "should happen once per year" }
end
```

La opción `:case_sensitive` define si la unicidad debe ignorar mayúsculas y minúsculas.

`validates_with`
Éste método pasa el registro a otra clase para emprender la validación.

```
class GoodnessValidator < ActiveModel::Validator
  def validate(record)
    if record.first_name == "Evil"
      record.errors[:base] << "This person is evil"
    end
  end
end

class Person < ActiveRecord::Base
  validates_with GoodnessValidator
end
```

3.7.6.3. Opciones de validación

Las opciones de validación más comunes son:

`:allow_nil` Salta la validación cuando el valor evaluado es `nil`.

```
  class Coffee < ActiveRecord::Base
    validates :size, inclusion: { in: %w(small
       medium large),
      message: "%{value} is not a valid size" },
          allow_nil: true
  end
```

`:allow_blank` Ésta opción ignora la validación si el valor es `nil` o una cadena de texto vacía.

```
class Topic < ActiveRecord::Base
  validates :title, length: { is: 5 },
     allow_blank: true
end

Topic.create(title: "").valid?  # => true
Topic.create(title: nil).valid? # => true
```

`:message` La opción `:message` permite especificar el mensaje que será añadido a la colección de errores cuando la validación falla.

`:on` Permite especificar cuando la validación debe ocurrir.

```ruby
class Person < ActiveRecord::Base
  # it will be possible to update email with a
     duplicated value
  validates :email, uniqueness: true, on: :
     create

  # it will be possible to create the record
     with a non-numerical age
  validates :age, numericality: true, on: :
     update

  # the default (validates on both create and
     update)
  validates :name, presence: true
end
```

3.7.6.4. Validación condicional

Se puede validar un objeto solamente cuando cierta condición es satisfecha. Para ello, *Rails* habilita los métodos `:if` y `unless`.

`Symbol` con `:if` y `:unless` Se puede asociar `:if` y `:unless` con un símbolo que corresponde con el nombre un método que será llamado cuando la validación ocurre.

```ruby
class Order < ActiveRecord::Base
  validates :card_number, presence: true, if: :
     paid_with_card?

  def paid_with_card?
    payment_type == "card"
  end
end
```

Cadena de caracteres con `:if` y `:unless` Es posible utilizar una cadena que será evaluada con el método `eval`.

```ruby
class Person < ActiveRecord::Base
  validates :surname, presence: true, if: "name.
     nil?"
end
```

Proc con `:if` y `:unless` También es posible asociar `:if` y `:unless` con un objeto de la clase `Proc` que se será llamado.

```
class Account < ActiveRecord::Base
  validates :password, confirmation: true,
    unless: Proc.new { |a| a.password.blank? }
end
```

Agrupado y combinado de validaciones
Resulta útil y clarificador poder combinar múltiples condiciones en una sola condición.

```
class User < ActiveRecord::Base
  with_options if: :is_admin? do |admin|
    admin.validates :password, length: { minimum: 10
        }
    admin.validates :email, presence: true
  end
end
```

De la misma manera se pueden combinar condiciones múltiples cuando una validación debe ocurrir o no.

```
class Computer < ActiveRecord::Base
  validates :mouse, presence: true,
                    if: ["market.retail?", :desktop
                       ?],
                    unless: Proc.new { |c| c.
                       trackpad.present? }
end
```

3.7.6.5. Validaciones personalizadas

Las validaciones personalizadas son creadas a través de clases que heredan de la clase `ActiveModel::Validator`. Éstas clases deben implementar un método `validate` que toma un argumento y le aplica la validación. Las validaciones personalizadas son ejecutadas llamando al método `validates_with`.

```
class MyValidator < ActiveModel::Validator
  def validate(record)
    unless record.name.starts_with? 'X'
      record.errors[:name] << 'Need a name starting
         with X please!'
```

```
    end
  end
end

class Person
  include ActiveModel::Validations
  validates_with MyValidator
end
```

También se pueden crear métodos personalizados que verifiquen el estado de los modelos y añadan mensajes a la colección de errores cuando son inválidos.

```
class Invoice < ActiveRecord::Base
  validate :expiration_date_cannot_be_in_the_past,
    :discount_cannot_be_greater_than_total_value

  def expiration_date_cannot_be_in_the_past
    if expiration_date.present? && expiration_date <
        Date.today
      errors.add(:expiration_date, "can't be in the
        past")
    end
  end

  def discount_cannot_be_greater_than_total_value
    if discount > total_value
      errors.add(:discount, "can't be greater than
        total value")
    end
  end
end
```

3.7.7. Callbacks

Los *callbacks* son métodos que son invocados en ciertos momentos de la vida de los objetos. Con ellos es posible escribir código que será ejecutado cuando un objeto Active Record es creado, guardado, actualizado, borrado, validado, o cargado de la base de datos.

```
class User < ActiveRecord::Base
  validates :login, :email, presence: true
```

```
  before_validation :ensure_login_has_a_value

  protected
    def ensure_login_has_a_value
      if login.nil?
        self.login = email unless email.blank?
      end
    end
end
```

Callbacks disponibles
Los *callbacks* de *Rails* disponibles son los siguientes:

- Creación del objeto
  - `before_validation`
  - `after_validation`
  - `before_save`
  - `around_save`
  - `before_create`
  - `around_create`
  - `after_create`
  - `after_save`
  - `after_commit` o `after_rollback`
- Actualización del objeto
  - `before_validation`
  - `after_validation`
  - `before_save`
  - `around_save`
  - `before_update`
  - `around_update`
  - `after_update`
  - `after_save`

    - `after_commit` o `after_rollback`
- Destrucción del objeto
    - `before_destroy`
    - `around_destroy`
    - `after_destroy`
    - `after_commit` o `after_rollback`
- Otros *callbacks*
    - `after_initialize` es un *callback* que será invocado cuando el objeto de Active Record es instanciado, ya sea mediante el método `new` o cargado desde la base de datos.
    - `after_find` es un *callback* llamado cuando un objeto Active Record es cargado desde la base de datos.
    - `after_touch` es un *callback* invocado cada vez que un objeto de Active Record es tocado.

Ejecución de callbacks
En *Rails*, los métodos siguientes disparan *callbacks*:

- `create`
- `create!`
- `decrement!`
- `destroy`
- `destroy!`
- `destroy_all`
- `increment!`
- `save`
- `save!`
- `save(validate: false)`
- `toggle!`

- `update_attribute`
- `update`
- `update!`
- `valid?`

Adicionalmente, el *callback* `after_find` es disparado por los siguientes métodos de búsqueda.

- `all`
- `first`
- `find`
- `find_by`
- `find_by_*`
- `find_by_*!`
- `find_by_sql`
- `last`

Callbacks relacionales

Los *callbacks* de *Rails* pueden funcionar en relaciones de modelos, y pueden incluso ser definidas por las propias relaciones. Puede darse el caso de que cuando un objeto es destruido, también deberian ser destruidos los objetos que dependen del susodicho objeto.

```
class User < ActiveRecord::Base
  has_many :articles, dependent: :destroy
end

class Article < ActiveRecord::Base
  after_destroy :log_destroy_action

  def log_destroy_action
    puts 'Article destroyed'
  end
end
```

```
>> user = User.first
=> #<User id: 1>
>> user.articles.create!
=> #<Article id: 1, user_id: 1>
>> user.destroy
Article destroyed
=> #<User id: 1>
```

Callbacks condicionales

De la misma forma que las validaciones, podemos ejecutar un *callback* siempre y cuando se ejecute una condición concreta. Análogamente a las validaciones, los mecanismos empleados para los *callbacks* condicionales son los mismos.

`Symbol` Las opciones `:if` y `:unless` pueden ser asociadas con un símbolo correspondiente al nombre de un método que será llamado antes de realizar el *callback*.

```
class Order < ActiveRecord::Base
  before_save :normalize_card_number, if: :
     paid_with_card?
end
```

Cadena de caracteres Una cadena de caracteres de código *Ruby* es admitida utilizando el método eval.

```
class Order < ActiveRecord::Base
  before_save :normalize_card_number, if: "
     paid_with_card?"
end
```

`Proc` *Rails* posibilita la asociación a un objeto `Proc`.

```
class Order < ActiveRecord::Base
  before_save :normalize_card_number,
    if: Proc.new { |order| order.paid_with_card?
        }
end
```

Callbacks con condiciones múltiples

```
class Comment < ActiveRecord::Base
  after_create :send_email_to_author, if: :
     author_wants_emails?,
```

```
    unless: Proc.new { |comment| comment.article
       .ignore_comments? }
end
```

Clases callback
Rails permite crear clases que encapsulan métodos *callback*, lo que facilita que otros modelos reutilicen esos *callbacks*.

```
class PictureFileCallbacks
  def after_destroy(picture_file)
    if File.exist?(picture_file.filepath)
      File.delete(picture_file.filepath)
    end
  end
end

class PictureFile < ActiveRecord::Base
  after_destroy PictureFileCallbacks.new
end
```

3.7.8. Asociaciones de Active Record

Las asociaciones facilitan y clarifican la manipulación y las operaciones sobre los objetos. Las asociaciones Active Record de *Rails* permiten declarar a *Rails* una conexión entre dos modelos especificando además de que tipo es.

3.7.8.1. Tipos de asociaciones

En *Rails*, una asociación es una conexión entre dos modelos de Active Record. Las asociaciones son implementadas usando el estilo macro, por lo que es posible especificar y añadir características a los modelos.

Asociación `belongs_to`
Ésta asociación establece una conexión de un modelo α de uno-a-muchos con otro modelo β. Para especificar el tipo de relación, la sintaxis[21] se realiza de la siguiente forma:

```
class Order < ActiveRecord::Base
  belongs_to :customer
end
```

[21] La asociación `belongs_to` emplea el singular del modelo al especificado.

La migración será de la siguiente forma:

```
class CreateOrders < ActiveRecord::Migration
  def change
    create_table :customers do |t|
      t.string :name
      t.timestamps null: false
    end

    create_table :orders do |t|
      t.belongs_to :customer, index: true
      t.datetime :order_date
      t.timestamps null: false
    end
  end
end
```

Asociación `has_one`

La asociación `has_one` es una conexión de uno-a-uno con otro modelo. Aunque parecida a la asociación anterior, ésta indica que cada instancia del modelo α contiene o posee una y sólo una instancia del modelo β. Para declarar

dicha asociación, se procede de la siguiente forma:[22]

```
class Supplier < ActiveRecord::Base
  has_one :account
end
```

Universidad de Alcalá

La migración será de la siguiente forma:

```
class CreateSuppliers < ActiveRecord::Migration
  def change
    create_table :suppliers do |t|
      t.string :name
      t.timestamps null: false
    end

    create_table :accounts do |t|
      t.belongs_to :supplier, index: true
      t.string :account_number
      t.timestamps null: false
    end
```

[22] Análogamente a la asociación `belongs_to`, ésta también utiliza el singular del modelo al que se refiere.

```
  end
end
```

Asociación `has_many`

La asociación `has_many` indica la relación de un modelo α de uno-a-muchos con otro modelo β. Es común encontrar esta asociación con la relación `belongs_to` en el otro modelo. La asociación indica que instancia del modelo tiene cero o varias instancias del otro modelo.[23]

```
class Customer < ActiveRecord::Base
  has_many :orders
end
```

Universidad de Alcalá

La migración será de la siguiente forma:

```
class CreateCustomers < ActiveRecord::Migration
  def change
    create_table :customers do |t|
      t.string :name
      t.timestamps null: false
```

[23]Se escribir nombre del modelo en forma plural en la declaración de la relación.

```
    end

    create_table :orders do |t|
      t.belongs_to :customer, index:true
      t.datetime :order_date
      t.timestamps null: false
    end
  end
end
```

Asociación `has_many :through`
Ésta asociación es usada frecuentemente cuando se pretende establecer una relación de muchos-a-muchos con otro modelo. La relación indica que el modelo α de la declaración puede tener cero o varias instancias del otro modelo β a través (`:thrrough`) de un modelo γ intermedio.

```
class Doctor < ActiveRecord::Base
  has_many :appointments
  has_many :patients, through: :appointments
end

class Appointment < ActiveRecord::Base
  belongs_to :doctor
  belongs_to :patient
end

class Patient < ActiveRecord::Base
  has_many :appointments
  has_many :doctors, through: :appointments
end
```

Universidad de Alcalá

La migración será de la siguiente forma:

```
class CreateAppointments < ActiveRecord::Migration
  def change
    create_table :doctors do |t|
      t.string :name
      t.timestamps null: false
    end

    create_table :patients do |t|
      t.string :name
      t.timestamps null: false
    end

    create_table :appointments do |t|
      t.belongs_to :doctor, index: true
      t.belongs_to :patient, index: true
      t.datetime :appointment_date
      t.timestamps null: false
    end
  end
```

```
end
```

Una de las grandes ventajas de esta técnica es que permite realizar la operación *join* de la base de datos con una sintaxis de orientación a objetos clara.

```
#Esta instrucción asigna una lista de pacientes a un
    doctor.
doctor.patients = patients
```

Asociación `has_one :through`
La asociación `has_one :through` establece una conexión de un modelo α de uno-a-uno con otro modelo β a través de un tercer modelo γ.

```
class Supplier < ActiveRecord::Base
  has_one :account
  has_one :account_history, through: :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  has_one :account_history
end

class AccountHistory < ActiveRecord::Base
  belongs_to :account
end
```

La migración será de la siguiente forma:

```
class CreateAccountHistories < ActiveRecord::
   Migration
  def change
    create_table :suppliers do |t|
      t.string :name
      t.timestamps null: false
    end

    create_table :accounts do |t|
      t.belongs_to :supplier, index: true
      t.string :account_number
      t.timestamps null: false
    end

    create_table :account_histories do |t|
      t.belongs_to :account, index: true
      t.integer :credit_rating
      t.timestamps null: false
    end
```

```
  end
end
```

Asociación `has_and_belongs_to_many`
La asociación `has_and_belongs_to_many`, comúnmente llamada **HABTM**, crea una conexión directa de un modelo α y otro modelo β.

```
class Assembly < ActiveRecord::Base
  has_and_belongs_to_many :parts
end

class Part < ActiveRecord::Base
  has_and_belongs_to_many :assemblies
end
```

Universidad de Alcalá

La migración será de la siguiente forma:

```
class CreateAssembliesAndParts < ActiveRecord::
   Migration
  def change
    create_table :assemblies do |t|
      t.string :name
```

```
      t.timestamps null: false
    end

    create_table :parts do |t|
      t.string :part_number
      t.timestamps null: false
    end

    create_table :assemblies_parts, id: false do |t|
      t.belongs_to :assembly, index: true
      t.belongs_to :part, index: true
    end
  end
end
```

Asociaciones polimorficas
Gracias a las asociaciones polimorficas, es posible definir un modelo que pertenece a más de un solo modelo, con una sola asociación.

```
class Picture < ActiveRecord::Base
  belongs_to :imageable, polymorphic: true
end

class Employee < ActiveRecord::Base
  has_many :pictures, as: :imageable
end

class Product < ActiveRecord::Base
  has_many :pictures, as: :imageable
end
```

Universidad de Alcalá

La migración será de la siguiente forma:

```
class CreatePictures < ActiveRecord::Migration
  def change
    create_table :pictures do |t|
      t.string  :name
      t.integer :imageable_id
      t.string  :imageable_type
      t.timestamps null: false
    end

    add_index :pictures, :imageable_id
  end
end
```

Las migraciones polimorficas también pueden ser simplificadas con el método `references`.

```
class CreatePictures < ActiveRecord::Migration
  def change
    create_table :pictures do |t|
      t.string :name
```

```
      t.references :imageable, polymorphic: true,
         index: true
      t.timestamps null: false
    end
  end
end
```

3.7.9. Interfaz de consultas

Active Record ejecuta consultas en la base de datos a través de su interfaz. El interfaz es compatible con la mayor parte de las bases de datos (MySQL, PostgreSQL, y SQLite), por lo que independientemente que se utilice, Active Record ejecutará las consultas, siempre, de la misma forma.

3.7.9.1. Recuperación de objetos

Para recuperar objetos de la base de datos, Active Record proporciona diversos métodos que permiten ejecutar consultas sin la necesidad de escribir directamente en SQL.

Objetos únicos Active Record proporciona diversas formas de recuperar un único objeto.

`find` Recupera el objeto correspondiente a la clave primaria proporcionada como parámetro.

```
# Encontrar el cliente con clave primaria 10.
client = Client.find(10)
# => #<Client id: 10, first_name: "Jose Luis
   ">
```

```
SELECT * FROM clients WHERE (clients.id = 10)
    LIMIT 1;
```

`take` Recupera el registro sin ningún orden implícito.

```
client = Client.take(2)
# => [
  #<Client id: 1, first_name: "Lifo">,
  #<Client id: 220, first_name: "Sara">
]
```

```
SELECT * FROM clients LIMIT 2;
```

`first` Recupera el primer registro ordenado por la clave primaria.

```
client = Client.first
# => #<Client id: 1, first_name: "Lifo">
```

```
SELECT * FROM clients ORDER BY clients.id ASC
    LIMIT 1;
```

`last` Analogamente, recupera el último registro ordenado por la clave primaria.

```
client = Client.last
# => #<Client id: 221, first_name: "Russel">
```

```
SELECT * FROM clients ORDER BY clients.id
   DESC LIMIT 1;
```

`find_by` Recupera el registro que cumple ciertas condiciones.

```
Client.find_by first_name: 'Lifo'
# => #<Client id: 1, first_name: "Lifo">
```

Objetos múltiples Active Record permite recuperar colecciones de objetos. Los métodos están implementados para poder operar por lotes, es decir, para poder operar con grandes cantidades de datos sin comprometer el rendimiento del sistema.

`find_each` Éste método recupera un lote de registros.

```
User.find_each do |user|
  NewsMailer.weekly(user).deliver_now
end
```

`find_in_batches` Éste método opera y tiene un comportamiento similar al método `find_each` ya que ambos recuperan un lote de registros. La diferencia es que `find_in_batches` proporciona un `array` al bloque en vez de un único objeto cada vez.

`find_each` y `find_in_batches` tienen dos opciones que modifican su comportamiento:

`:batch_size` Permite especificar el número de registros recuperados por lote.

`:start` Permite especificar donde debe comenzar el lote que va a ser recuperado.

3.7.9.2. Condiciones

El método `where` permite especificar condiciones para limitar los registro recuperados. Representa el `WHERE` de una instrucción SQL. Las condiciones pueden ser especificadas de diversas maneras.

Condiciones de cadena
Es posible añadir las condiciones directamente como parametro en forma de cadena. Ésta técnica no es recomendable ya que deja el sistema vulnerable a SQL injection[24] que pueden comprometer el sistema.

Condiciones de Array
Active Record permite especificar condiciones en forma de `array` de la siguiente forma:

```
Client.where("orders_count = ? AND locked = ?",
   params[:orders], false)
```

Cuando es ejecutado, el carácter *?* es sustituido por el parámetro correspondiente a la posición del carácter. Ésta técnica, si es utilizada debidamente, no es vulnerable al mencionado SQL injection. *Rails* permite cambiar el carácter *?* por un identificador con la intención de clarificar consultas complejas con gran número de condiciones..

```
Client.where("created_at >= :start_date AND
   created_at <= :end_date",
  {start_date: params[:start_date], end_date: params
     [:end_date]})
```

Condiciones Hash
Active Record permite pasar condiciones *hash* que incrementan la legibilidad sintáctica de las condiciones. Se deben proporcionar un *hash* con las claves del los campos que deben ser condicionados y los valores condicionantes. Son varios los tipos de *hash* que podemos proporcionar:

Igualdades
```
Client.where(locked: true)
```

En el caso de la relación `belongs_to`, la clave puede ser utilizada para especificar el modelo si un objeto Active Record es usado como valor. También funciona con relaciones polimorficas.

[24]SQL injection es una técnica de inserción de código que se utiliza para atacar aplicaciones que trabajan con bases de datos, en las que código malicioso SQL es insertado dentro del campo de ejecución.

```
Article.where(author: author)
Author.joins(:articles).where(articles: { author
    : author })
```

Rangos `Client.where(created_at: (Time.now.`
`midnight - 1.day)..Time.now.midnight)`

El código anterior, en SQL sería lo siguiente:

```
SELECT * FROM clients WHERE (clients.created_at
    BETWEEN '2008-12-21 00:00:00' AND '2008-12-22
     00:00:00');
```

Subconjuntos `Client.where(orders_count: [1,3,5])`

El código anterior, en SQL sería lo siguiente:

```
SELECT * FROM clients WHERE (clients.
    orders_count IN (1,3,5));
```

Condiciones negadas
Las condiciones SQL negadas se construyen simplemente con el método `where.not`.

```
Article.where.not(author: author)
```

3.7.9.3. Ordenación

Active Record permite la recuperación de los registros de la base de datos de forma ordenada mediante el uso del método `order`.

```
Client.order(:created_at)
# otro modo
Client.order("created_at")
```

El método se complementa con diversas opciones para especificar la recuperación de registros.

- Especificación orden normal o orden inverso.

```
Client.order(created_at: :desc)
# otro modo
Client.order(created_at: :asc)
# otro modo
```

```
Client.order("created_at DESC")
# otro modo
Client.order("created_at ASC")
```

- Ordenado de múltiples campos.

```
Client.order(orders_count: :asc, created_at: :
   desc)
# otro modo
Client.order(:orders_count, created_at: :desc)
# otro modo
Client.order("orders_count ASC, created_at DESC"
   )
# otro modo
Client.order("orders_count ASC", "created_at
   DESC")
```

3.7.9.4. Selección

Por defecto, `Model.find` selecciona todos los campos del resultado de la consulta utilizando `select *`. Para seleccionar un subconjunto de campos del conjunto de resultados, puede especificarse el subconjunto mediante el método `select`.

```
Client.select("viewable_by, locked")
```

Que se traduce en lenguaje SQL como:

```
SELECT viewable_by, locked FROM clients;
```

Rails también permite recuperar un único registro por valor único en un campo determinado como si en el lenguaje SQL se utilizara `DISTINCT`.

```
Client.select(:name).distinct
```

3.7.9.5. Limit y Offset

Active Record provee los métodos `limit`, que especifica el número de registros que deben ser recuperados; y `offset`, que especifica el número de registros que deben ser saltados antes de empezar la operación de recuperación de registros.

```
Client.limit(5).offset(30)
```

3.7.9.6. Agrupación

Para aplicar la operación de agrupación de SQL, `GROUP BY`, se puede especificar con Activer Record mediante el método `group`.

```
Order.select("date(created_at) as ordered_date, sum(
    price) as total_price").group("date(created_at)")
```

La consulta SQL que será ejecutada es la siguiente:

```
SELECT date(created_at) as ordered_date, sum(price)
    as total_price
FROM orders
GROUP BY date(created_at);
```

3.7.9.7. Join

Active Record proporciona un método de búsqueda llamado `joins` para especificar instrucciones `JOIN` de SQL. Active Record provee varias formas de realizar la operación `JOIN`:

Formato SQL puro
Se aporta una cadena de caracteres de código SQL puro en método `joins`.

```
Client.joins('LEFT OUTER JOIN addresses ON addresses
    .client_id = clients.id');
```

Formato Array/Hash
Active Record permite usar los nombres de las asociaciones definidas en los modelos para facilitar la especificación de las instrucciones `JOIN` cuando se utiliza el método `joins`. Éste método solo funciona con la instrucción `INNER JOIN`.

Considerando los siguientes modelos:

```
class Category < ActiveRecord::Base
  has_many :articles
end

class Article < ActiveRecord::Base
  belongs_to :category
  has_many :comments
  has_many :tags
end
```

```
class Comment < ActiveRecord::Base
  belongs_to :article
  has_one :guest
end

class Guest < ActiveRecord::Base
  belongs_to :comment
end

class Tag < ActiveRecord::Base
  belongs_to :article
end
```

Asociación única `Category.joins(:articles)`

Produce:

```
SELECT categories.* FROM categories
INNER JOIN articles ON articles.
  category_id = categories.id;
```

Asociación múltiple `Article.joins(:category, :comments)`

Produce:

```
SELECT articles.* FROM articles
INNER JOIN categories ON articles.
  category_id = categories.id
INNER JOIN comments ON comments.article_id =
  articles.id;
```

Asociación anidada `Article.joins(comments: :guest)`

Produce:

```
SELECT articles.* FROM articles
INNER JOIN comments ON comments.
  article_id = articles.id
INNER JOIN guests ON guests.comment_id =
  comments.id;
```

Asociaciones anidadas (varios niveles)

```
Category.joins(articles: [{
comments: :guest }, :tags])
```

Produce:

```
SELECT categories.* FROM categories
INNER JOIN articles ON articles.
   category_id = categories.id
INNER JOIN comments ON comments.
   article_id = articles.id
INNER JOIN guests ON guests.comment_id =
    comments.id
INNER JOIN tags ON tags.article_id =
   articles.id;
```

3.7.9.8. Scopes

Active Record permite definir *scopes* que son consultas definidas debido a que tendrán un uso habitual en la aplicación, y que pueden ser referenciadas como llamadas a métodos de los objetos de la asociación.

Para definir un *scope*, se debe utilizar el método `scope` dentro de la clase junto a la consulta que debe ser ejecutada en dicho *scope*.

```
class Article < ActiveRecord::Base
  scope :published, -> { where(published: true) }
end
```

Active Record permite la anidación de *scopes*, lo que simplifica la codificación de *scopes* complejos.

```
class Article < ActiveRecord::Base
  scope :published,                -> { where(
     published: true) }
  scope :published_and_commented, -> { published.
     where("comments_count > 0") }
end
```

Es también posible definir un scope al que se le pueden pasar argumentos:

```
class Article < ActiveRecord::Base
  scope :created_before, ->(time) { where("
     created_at < ?", time) }
end
```

3.8. Vistas

En *Rails*, las vistas están gestionadas Action View. Action View es junto con Action Controller, uno de los componentes más importantes de Action Pack. En *Ruby on Rails*, las peticiones web son gestionadas por Action Pack, que divide el trabajo entre el controlador y la vista.

Action View utiliza plantillas o *templates* que están codificadas usando código *Ruby* embebido entre las etiquetas HTML. El própisito de éstas plantillas es visualizar los resultados de cada acción del controlador asociado.

En *Rails*, por cada controlador, existe un directorio en `apps/views` que mantiene los ficheros de las plantillas que construyen las vistas asociadas a dicho controlador.

3.8.1. Plantillas, parciales y layouts

La producción final de HTML está compuesta de tres elementos en *Rails*: Plantillas, parciales y layouts, que son utilizados para visualizar vistas comunes de las plantillas.

3.8.1.1. Plantillas

Las plantillas de Action View pueden ser codificadas de diversas formas:

ERB Dentro de un plantilla ERB, el código *Ruby* puede ser incluido usando las etiquetas `<%%>` y `<%=%>`. La etiqueta `<%%>` es utilizada cuando el código *Ruby* no devuelve nada; como es el caso de condiciones o bucles; mientras que `<%=%>` es usada cuando se requiere una salida.

```
<h1>Names of all the people</h1>
<% @people.each do |person| %>
        Name: <%= person.name %><br>
<% end %>
```

Builder Las plantillas de builder son mucho más útiles para generar contenido XML.

```
xml.em("emphasized")
xml.em { xml.b("emph & bold") }
xml.a("A Link", "href" => "http://
   rubyonrails.org")
xml.target("name" => "compile", "option"
    => "fast")
```

Rails, por defecto, compilará cada plantilla a un método para visualizarla. Cuando una plantilla es alterada. *Rails* comprueba de que el archivo ha sido modificado y lo recompila en modo desarrollo.

3.8.1.2. Parciales

Los parciales son un instrumento para dividir el proceso de visualización de una plantilla en piezas más simples y manejables. Con los parciales, se puede extraer piezas de código de las plantillas e insertarlas en archivos separados, simplificando las plantillas y aún más importante, garantizando la reutilización de código.

Para renderizar un parcial, se debe utilizar el método `render` dentro de la vista:

```
<%= render "menu" %>
```

Una forma útil de utilizar los parciales es a modo de subrutinas con el objetivo de tener un código más claro y legible.

```
<%= render "shared/ad_banner" %>

<h1>Products</h1>

<p>Here are a few of our fine products:</p>
<% @products.each do |product| %>
  <%= render partial: "product", locals: {product:
     product} %>
<% end %>

<%= render "shared/footer" %>
```

Es común que una plantilla itere sobre una colección y muestre una subplantilla por cada elemento de la colección. Este patrón puede ser implementado con un sólo método que acepte un vector y visualice un parcial para cada uno de los elementos del vector.

```
<% @products.each do |product| %>
  <%= render partial: "product", locals: { product:
     product } %>
<% end %>

O aún en un estilo más Rails:
<%= render partial: "product", collection: @products
    %>
```

3.8.1.3. Layouts y layouts parciales

Los layouts de Active View son elementos utilizados para visualizar y mostrar vistas comunes de una plantilla en torno a los resultados de la acción del controlador. Ejemplos típicos de layouts son páginas comunes en web como FAQ, precios, contacto o términos y condiciones.

Los parciales puede tener asociados sus propios layouts. Estos layouts son diferentes a los aplicados en el controlador aunque funcionan de forma similar.

Para mostrar un ejemplo de ellos, supongamos la siguiente situación:

```
Article.create(body: 'Esto es un layout parcial')
```

La plantilla de la acción `show`, se visualizará el parcial `_article` encapsulado en un layout llamado `box`.

articles/show.html.erb

```
<%= render partial: 'article', layout: 'box', locals
   : {article: @article} %>
```

El layout `box` encapsula el parcial `_article` en una etiqueta HTML `div`:

articles/_box.html.erb

```
<div class='box'>
  <%= yield %>
</div>
```

articles/_article.html.erb

```
<%= div_for(article) do %>
  <p><%= article.body %></p>
<% end %>
```

El resultado final en HTML será el siguiente:

```
<div class='box'>
  <div id='article_1'>
    <p>Esto es un layout parcial</p>
  </div>
</div>
```

3.9. Controladores

Los controladores son el componente de la arquitectura MVC que determinan que acción debe aplicarse ante una entrada concreta, generando una salida apropiada. Los controladores puede ser pensados como elementos entre

los modelos y las vistas. El controlador hace disponibles los datos del modelo a la vista, de forma que, ésta última muestre los datos al usuario, y guarda o actualiza los datos introducidos por el usuario al modelo. El controlador de *Rails* es implementado por Action Controller.

3.9.1. Action Controller

Después de que el enrutamiento haya determinado que controlador debe de ser aplicado para una determinada petición, dicho controlador es responsable de producir la salida apropiada. Action Controller realiza casi todo el trabajo utilizando convenciones para facilitar el proceso al desarrollador.

Para las aplicaciones RESTful convencionales, la forma más común en la que un controlador funciona es: el controlador recibe una petición, extrae o guarda datos en el modelo pertinente e invoca una vista para generar una salida HTML.

Teniendo en cuenta que una de las filosofías de *Rails* es convención sobre configuración, los controladores de *Rails* están sujetos a una convención en sus nombres. Los controladores de *Rails* favorecen la pluralización de la última palabra del nombre del controlador. Siguiendo ésta convención, se permite utilizar generadores de rutas por defecto sin necesidad de especificar la ruta o el controlador, manteniendo la URL y los *helpers* consistentes a través de la aplicación.

3.9.1.1. Métodos y acciones

A nivel de código, un controlador es una clase de *Ruby* que hereda de la clase `ApplicationController`. Cuando la aplicación recibe una petición, el enrutamiento detrermina que controlador y que acción del controlador debe ejecutarse. Después *Rails* crea una instancia de ese controlador y ejecuta el método con el mismo nombre que la acción.

```
class ClientsController < ApplicationController
  def new
        @client = Client.new
  end
end
```

En el ejemplo anterior, si un usuario va a `URL_APLICACIÓN/clients/new` para añadir un nuevo cliente, *Rails* creará una instancía del controlador `ClienteController` y ejecutará el método `new` creando un nuevo cliente. Un método vacío puede funcionar porque *Rails*, por defecto, visualizara la vista `new.html.erb` a menos que se especifique lo contrario.

`ApplicationController` es una clase que hereda de una clase superior, `ActionController::Base`, que tiene implementados una serie de métodos útiles para la codificación de controladores.

3.9.1.2. Parámetros

En *Rails*, naturalmente, se puede acceder a datos enviados por el usuario o otros parámetros en las acciones de un controlador. Hay dos tipos de parámetros posibles en una aplicación web: Los parámetros enviados junto con la URL de la aplicación, denominados parámetros "query string"; y los parámetros que son referidos como datos POST. Esta información suele venir desde un formulario HTML que ha sido rellenado por el usuario. Es llamado datos POST puesto que solo puede ser enviado como parte de una petición HTTP POST. *Rails* no hace distinciones entre parámetros "query string" y POST, y ambos son disponibles en el hash `params` del controlador.

```
class ClientsController < ApplicationController
  #Esta acción usa query string parameters porque es
      ejecutado por una petición GET de HTML, pero
     no hace diferencias en la forma de que los pará
     metros son accedidos. La URL de esta acción con
      lista ordenada será similar a: clients: /
     clients?status=activated
  def index
    if params[:status] == "activated"
      @clients = Client.activated
    else
      @clients = Client.inactivated
    end
  end

  #Esta acción usa parámetros POST. La URL para la
     petición RESTful será "/clients" y los datos
     serán enviados como parte del cuerpo de la
     petición.
  def create
    @client = Client.new(params[:client])
    if @client.save
      redirect_to @client
    else
      render "new"
    end
```

```
  end
end
```

Parámetros `Hash` y `Array` En *Rails* se permite el uso de hashes y array como parametros. El hash `params` no está limitado a una dimensión de claves y valores. Puede contener hashes anidados.

```
GET /clients?ids[]=1&ids[]=2&ids[]=3
```

Un hash debe ser enviado incluyendo el nombre de la clave y el valor entre corchetes. Suponiendo un formulario en el que los datos son enviados de la siguiente forma:

```
<form accept-charset="UTF-8" action="/clients"
  method="post">
      <input type="text" name="client[name]"
        value="Acme" />
      <input type="text" name="client[phone]"
        value="12345" />
      <input type="text" name="client[address
        ][postcode]" value="12345" />
      <input type="text" name="client[address
        ][city]" value="Carrot City" />
</form>
```

Cuando el formulario es enviado, el valor de `params[:client]` será:

```
{ "name" => "Acme", "phone" => "12345", "address
  " => { "postcode" => "12345", "city" => "
  Carrot City" } }
```

Parámetros JSON En una aplicación web es interesante aceptar parámetros en un formato JSON. Si la cabecera "Content-Type" de la petición es establecida como "application/json", *Rails* automáticamente convierte los parámetros en el hash `params`.

Suponiendo el siguiente contenido JSON:

```
{ "company": { "name": "acme", "address": "123
  Carrot Street" } }
```

Se obtendrá `params[:company]` como:

```
{ "name": "acme", "address": "123 Carrot Street"
   }
```

Strong parameters Con los "strong parameters", los parámetros de Action Controller están prohibidos para ser usados en la asignación masiva de Active Model a menos que se especifique lo contrario. Esto significa, que el progrmador debe hacer un elección sobre qué parámetros son permitidos para asignación masiva, evitando la exposición accidental de atributos del modelo.

```
class PeopleController < ActionController::Base
  def create
    Person.create(params[:person])
  end

  def update
    person = current_account.people.find(params
       [:id])
    person.update!(person_params)
    redirect_to person
  end

  private
    def person_params
      params.require(:person).permit(:name, :age
         )
    end
end
```

Los "strong parameters" pueden ser definidos para los casos anidados de la siguiente forma:

```
params.permit(:name, { emails: [] },
              friends: [ :name,
                         { family: [ :name ],
                           hobbies: [] }])
```

3.9.1.3. Sesiones

Las aplicaciones de *Rails* tienen una sesión por cada usuario en la que se almacena una cantidad pequeña de datos que será persistente entre las diferentes peticiones. La sesión solo está disponible en el controlador y la vista, y puede usar uno de los siguientes mecanismos de almacenamiento:

- `ActionDispatch::Session::CookieStore` Almacena todos los datos en el cliente.

- `ActionDispatch::Session::CacheStore` Almacena los datos en el cache de *Rails*.
- `ActionDispatch::Session::ActiveRecordStore` Almacena los datos en la base de datos utilizando ActiveRecord. Para su funcionamiento, se precisa la gema `activerecord-session_store`.
- `ActionDispatch::Session::MemCacheStore` Almacena los datos del clúster de *memcached*.

Todas las sesiones utilizan una *cookie* para almacenar un ID único por cada sesión. La cookie puede ocupar en torno a 4kB de datos que a pesar de no ser demasiado es suficiente, y además almacenar grandes cantidades de información no es recomendable puesto que el servidor puede no ser capaz de reensamblar en la cookie todas las peticiones, lanzando un error correspondiente.

Flash

El *flash* es una parte especial de la sesión que es despejada con cada petición. Esto significa que los valores almacenados sólo están disponibles en la siguiente petición, lo que resulta útil para el lanzamiento de mensajes de error.

Un ejemplo de flash es la acción de desloguear, el controlador puede enviar un mensaje que será mostrado al usuario en la siguiente petición:

```
class LoginsController < ApplicationController
  def destroy
    session[:current_user_id] = nil
    flash[:notice] = "You have successfully logged
       out."
    redirect_to root_url
  end
end
```

Los mensajes flash también pueden ser asignados como parte de la redirección. Se puede asignar `:notice`, `:alert` o `:flash` para propósitos generales.

```
redirect_to root_url, notice: "You have successfully
    logged out."
redirect_to root_url, alert: "You're stuck here!"
redirect_to root_url, flash: { referral_code: 1234 }
```

3.9.1.4. Filtros

Filtros son métodos que se ejecutan antes, después o "sobre" un acción de un controlador. Los filtros son heredados, por lo que si se establece un filtro en la clase `ApplicationController`, podrá ser ejecutado en cualquier controlador de la aplicación.

Filtro Before
Éste filtro es ejecutado antes de que se ejecute la acción por lo que puede detener la petición web. Un filtro típico es aquel que requiere que el usuario esté logueado antes de ejecutar la acción.

```
class ApplicationController < ActionController::Base
  before_action :require_login

  private

  def require_login
    unless logged_in?
      flash[:error] = "You must be logged in to
         access this section"
      redirect_to new_login_url # halts request
         cycle
    end
  end
end
```

Filtro After
Éste filtro en contraposición al filtro `before`, es ejecutado después de que la acción haya tenido lugar. Obviamente no pueden detener la ejecución de la acción.

Filtro Around
El filtro `around` es responsable de ejecutar sus acciones asociadas durante la ejecución de la acción.

3.9.2. Enrutamiento en Rails

El enrutador de *Rails* reconoce URLs y las envía a la acción de un controlador. Puede también generar rutas y URLs, sin tener la necesidad de codificarlas a mano en las vistas.

El propósito del enrutador de *Rails* es doble, conectar determinadas URLs al código y generar rutas y URLs desde el código:

Conexión de URLs al código Se refiere a que *Rails* recibe una petición, y consulta al enrutador qué acción y controlador cuadra con dicha petición, después es enviada a la acción del controlador especificado por el enrutador.

Suponiendo la petición y la ruta siguientes:

```
GET /patients/17
```

```
get '/patients/:id', to: 'patients#show'
```

la petición será enviada a la acción `show` del controlador `patients` con los siguientes datos en `params`.

```
{ id: '17' }
```

Generación de rutas y URLs desde el código *Rails* posibilita la generación de rutas y URLs. Ésta característica es especialmente útil para una mejora de la claridad y solidez del código. Suponiendo la ruta del ejemplo anterior, si modificada, sería:

```
get '/patients/:id', to: 'patients#show', as: '
   patient'
```

Suponiendo, también, que la aplicación contiene el código en el controlador y las vista siguientes:

```
@patient = Patient.find(17)
```

```
<%= link_to 'Patient Record', patient_path(
   @patient) %>
```

Entonces, el enrutador de *Rails* generará la ruta `patients/17`.

3.9.3. Enrutamiento con recursos

El enrutamiento con recursos permite declarar todas las rutas comunes para un controlador determinado. En vez de declarar las rutas de las acciones de forma separada, `index`, `show`, `new`, `edit`, `create`, `update` y `destroy`; con el enrutamiento con recursos se declaran en una sola linea.

3.9.3.1. Recursos en la Web

Los navegadores hacen peticiones de páginas a *Rails* haciendo una petición a una URL usando un método concreto de HTTP, como `GET`, `POST`, `PATCH`, `PUT`, `PUT`, y `DELETE`. Cada método es requerido para hacer una operación en el recurso.

Cuando una aplicación *Rails* recibe la siguiente petición:

```
DELETE /photos/17
```

Y la primera ruta encontrada es:

```
resources :photos
```

Rails enviará esa peticion al método `destroy` en el controlador `photos` con el siguiente valor en `params`.

```
{ id: '17' }
```

3.9.3.2. CRUD, verbos y acciones

En *Rails*, una ruta de recursos proporciona una tabla entre los verbos HTTP y las URL a las acciones del controlador. Por convenio, cada acción se enlaza con una operación CRUD particular de la base de datos. La siguiente entrada en el archivo de enrutamiento:

```
resources :photos
```

Genera las siguientes rutas diferentes en la aplicación con relación al controlador `photos`.

Verbo HTTP	Ruta	Controlador#Acción
GET	/photos	photos#index
GET	/photos/new	photos#new
POST	/photos	photos#create
GET	/photos/:id	photos#show
GET	/photos/:id/edit	photos#edit
PATCH / PUT	/photos/:id	photos#update
DELETE	/photos/:id	photos#destroy

3.9.3.3. Namespaces

Rails posibilita la organización por grupos de los controladores bajo los *namespaces*. Suponiendo que se pretender agrupar una serie de controladores administrativos bajo un *namespace* de administración, se emplazarían los controladores bajo el directorio `app/controllers/admin`, y se agruparían en juntos en el enrutador:

```
namespace :admin do
  resources :articles, :comments
end
```

Para el caso de `Admin::ArticlesController`, *Rails* crearía la siguiente tabla:

Ruta	Controlador#Acción	Procedimiento
/admin/articles	admin/articles#index	admin_articles_path
/admin/articles/new	admin/articles#new	new_admin_article_path
/admin/articles	admin/articles#create	admin_articles_path
/admin/articles/:id	admin/articles#show	admin_article_path(:id)
/admin/articles/:id/edit	admin/articles#edit	edit_admin_article_path(:id)
/admin/articles/:id	admin/articles#update	admin_article_path(:id)
/admin/articles/:id	admin/articles#destroy	admin_article_path(:id)

3.9.3.4. Recursos anidados

Es común tener recursos que son hijos de otros recursos, este es el caso de los recursos anidados. Los recursos anidados permiten capturar las relaciones entre los objetos de Active Record en el enrutamiento.

Suponiendo que se tienen los siguientes modelos y las siguientes rutas:

```
class Magazine < ActiveRecord::Base
  has_many :ads
end

class Ad < ActiveRecord::Base
  belongs_to :magazine
end
```

```
resources :magazines do
  resources :ads
end
```

La tabla generada por el enrutador de *Rails* seria de la siguiente forma:

Verbo HTTP	Ruta	C#Acción
GET	/magazines/:magazine_id/ads	ads#index
GET	/magazines/:magazine_id/ads/new	ads#new
POST	/magazines/:magazine_id/ads	ads#create
GET	/magazines/:magazine_id/ads/:id	ads#show
GET	/magazines/:magazine_id/ads/:id/edit	ads#edit
PATCH	/magazines/:magazine_id/ads/:id	ads#update
DELETE	/magazines/:magazine_id/ads/:id	ads#destroy

A pesar de que es posible tener varios niveles de anidamiento, los recursos anidados no deberían tener más de un nivel de anidación debido a que las URLs y los helpers se vuelven complejos.

```
resources :publishers do
  resources :magazines do
    resources :photos
  end
end
```

Ésta aplicación reconocería una petición del estilo:

```
/publishers/1/magazines/2/photos/3
```

Una forma de evitar las anidaciones profundas es utilizar el mecanismo *shallow nesting* que genera una colección de acciones bajo un mismo recurso padre. La idea es quitar recursos innecesarios de las rutas. Por ejemplo en una acción `edit`, se sólo requerirá el ID del objeto que se pretende editar, y no se necesita el resto de recursos padre para localizar el objeto con dicho ID.

```
resources :articles do
  resources :comments, shallow: true
end
```

Capítulo 4

Análisis del sistema

4.1. Introducción

Esta parte del documento cubrirá el análisis del sistema, comenzando por la descripción de las funcionalidades del sistema que dará paso a la elaboración y especificación de los requisitos del sistema hasta los diagramas de eventos y diferentes flujos de datos del sistema.

Se utilizará la metodología de análisis estructurado para realizar el análisis del componente mediante diagramas de flujos de datos.

4.2. Requisitos del sistema

Los requisitos del sistema pueden ser agrupados en diferentes categorías debido a su naturaleza y en base a sus características.

4.2.1. Requisitos de interfaces de usuario

El interfaz de usuario y la experiencia del usuario juegan un papel fundamental en ésta aplicación debido a su forma de red social. A grandes rasgos, se debe lograr una interfaz intuitiva y que no abrume al usuario debido a la gran cantidad de información que debe ser mostrada.

Interfaz de usuario intuitiva
Se insistirá especialmente en conseguir una interfaz de usuario lo más intuitiva posible. La idea es que el usuario pueda manejar la aplicación sin tener amplios conocimientos informáticos o tecnológicos ni tampoco precise de instrucción o un tutorial para la interacción con la aplicación.

Otro concepto que se pretende transmitir al interfaz de usuario es que en la propia experiencia de usuario, el usuario sea capaz de entender si lo que está haciendo está bien o no. Esta es una idea traída del desarrollo de videojuegos en la cual hay que hacer entender al jugador si está jugando bien o mal, por ejemplo si en un cierto juego, aparece un item que dará una determinada mejora al personaje del jugador, debe ser visualmente atractivo e indicar que es algo "bueno" sin haberlo incluso cogido.

Interfaz de usuario sencilla
Es fundamental que la interfaz de usuario sea lo más clara y sencilla posible. Dado el carácter de red social, la información que se visualiza en cada pantalla es enorme. Por tanto, para lograr un interfaz no recargado, los elementos deben agruparse en diferentes planos según su importancia.

Para llevar a cabo dicho objetivo, se deberán disponer de menús, submenús, barras de navegación y otros elementos, así como, un estudio concienzudo sobre que elementos deben aparecer en los planos más importantes y cuales deben aparecer en planos de un carácter secundario.

Interfaz de usuario atractiva
La interfaz de usuario debe ser lo más atractiva posible. Para lograrlo, se debe disponer un diseño o layout que resulte visualmente interesante y que además, consiga el objetivo anterior de una interfaz sencilla. Además se intentará buscar un esquema de colores atractivo visualmente y que vaya acorde con la temática de la aplicación, en este caso, la escalada.

También se pueden disponer de elementos como iconos, imágenes o una fuente interesante que alivien y aligeren el diseño visual.

Interfaz de usuario con diseño reponsive
Responsive es un concepto de diseño web que busca la creación de sitios web que proporcionen una visualización optima del contenido y la experiencia de interacción. Fácil lectura y navegación con el mínimo posible de scrolling, cambios de tamaño, ... son características de un diseño responsive. El concepto detrás del diseño responsive es que el contenido de la web deber ser como el agua, el agua se adapta a su continente.

"*You put water into a cup it becomes the cup. You put water into a bottle it becomes the bottle. You put it in a teapot, it becomes the teapot. Be water my friend*" –Bruce Lee

4.2.2. Requisitos funcionales

4.2.3. Requisitos de validación y verificación

Teniendo en cuenta que se trata de una aplicación en la que los usuarios de la misma son los que introducen y pueblan con datos e información a la aplicación, es de especial interés la validez de los datos.

Validación de datos y tipos de datos
Los tipos de datos deben ser validados, en la medida de lo posible, para ser introducidos de forma satisfactoria en la aplicación, esto significa, que si un dato debe ser número entero, no se puede permitir que el usuario inserte una cadena de caracteres, por ejemplo. En otras palabras, el objetivo máximo a cumplir es que la aplicación no se vea comprometida con la inserción de ciertos datos.

Validación de la información
No se validará la información que los usuarios insertan, esto significa, que aunque los datos si serán validados, tal y como se especificó en el punto anterior; la información que suben los usuarios depende de ellos y no será comprobada. Poniendo un ejemplo, que un determinado usuario haya subido información falsa sobre la localización de un sitio de escalada o incluso que un usuario abuse verbalmente o publique contenido adulto, racista o sexista.

4.2.4. Mantenimiento de la aplicación

Se debe enfatizar en el desarrollo de una aplicación lo más sencilla de mantener posible. Aparte de "por buena programación", el motivo principal de este requisito es que se pretende que la aplicación siga siendo desarrollada en el futuro. Por tanto, debe llevarse un sistema de control de versiones, el código debe ser lo más claro y desacoplado posible, el proyecto ha de ser bien documentado y en general, hacer un esfuerzo para llevar a cabo las buenas practicas del desarrollo software para conseguir que la fase de mantenimiento sea efectiva.

4.2.5. Otros requisitos de la aplicación

Algunos de los requisitos de la aplicación no mencionados anteriormente son los siguientes:

Accesible vía navegador web
Como es bastante evidente, la aplicación será accesible desde navegadores web que actuarán como cliente, por tanto, es obvio que se precisará de un navegador web que procese las peticiones web.

RESTful
En su totalidad o en gran medida, la aplicación web debe ser RESTful. REST es el estilo de arquitectura software que consiste en en un conjunto de constantes aplicadas a componentes, conectores y datos. REST ignora los detalles de implementación de los componentes y del protocolo sintáctico.

Desarrollado en Ruby on Rails
Dado que uno de los objetivos de éste proyecto es aprender el framework *Ruby on Rails*, la aplicación debe ser desarrollada con dicha tecnología intentando abarcar lo máximo posible del framework para tener un aprendizaje lo más completo y sólido posible.

Requisitos de rendimiento
Dado el carácter de red social, se generan ciertos problemas de rendimiento debido a la gran cantidad de datos con la que se va a operar. En principio, no hay requisitos específicos ya que no se trata de una aplicación en el que los tiempos de respuesta sean lo más importante como en la programación en tiempo real, pero si que se pretende ser lo más eficiente posible, proporcionando tecnologías rápidas, una base de datos bien estructurada y un código eficiente en lo que a ejecución se refiere.

Capítulo 5

Conclusión

El final de la historia es sorprendente...

Apéndice A

Más cosas

Aún faltan cosas por decir.

Apéndice B

Y más cosas aún

Y más cosas aún.

Bibliografía

[1] Y. Matsumoto, "Email de Matsumoto con asunto "History of Ruby" en ruby-talk." `http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/382`, 1999. Accedido: 20-01-2016.

[2] N. Sieger, "Rubyconf: History of ruby." `http://blog.nicksieger.com/articles/2006/10/20/rubyconf-history-of-ruby`, 2006. Accedido: 23-01-2016.

[3] B. Venners, "A conversation with Yukihiro Matsumoto." `http://www.artima.com/intv/ruby.html`, 2003. Accedido: 25-01-2016.

[4] K. B. Bruce, *Foundations of object-oriented languages: types and semantics*, capítulo 2.1, p. 18. The MIT Press, 2002.

[5] M. Mintz y R. Ekendahl, *Hardware Verification with C++: A Practitior's Handbook*, p. 22. Springer, 2006.

[6] S. Ruby, D. Thomas, y D. Hansson, *Agile Web Development with Rails 4 (Pragmatic Programmers)*, capítulo Introducción. The Pragmatic Bookshelf, 2013.

[7] L. Grimmer, "Interview with David Heinemeier Hansson from Ruby on Rails," 2006.

[8] D. H. Hansson, "Ruby on Rails will ship with OS X 10.5 (Leopard)," 2006.

[9] M. Fowler, *Patterns of Enterprise Application Architecture*, p. 160. Addison-Wesley Professional, 2002.