

# ActiveRecord

Traducido al español por **Rodolinux**: <http://www.rodolinux.com.ar>  
Correcciones y Arreglos Estéticos por [Pablo Luis Martinez](#) – Sgo del Estero

*Es una implementación de un patrón de diseño ‘Active Record’, el cual es un mecanismo de Mapeo Objeto-Relacional (Object Relational Mapping - ORM).*

ActiveRecord es una de esas implementaciones de este patrón de diseño que nos ayuda a realizar todas las tareas de nuestras bases de datos (Definición de Datos y Manipulación de Datos a través de sus capacidades de metaprogramación) sin tener que escribir una sentencia SQL. AR representa a las tablas de la base de datos a través de clases e instancias.

Una clase que trabaja en relación a una tabla de base de datos es llamada “modelo”, las instancias de estas clases son “instancias de modelo”. Los modelos encapsulan la lógica de negocios.

Algunos ejemplos más de implementaciones ORM son ‘Hibernate’ para Java y ‘Castle’ en .Net.

AR es una librería incorporada a Rails, la cual puede ser utilizada con otros frameworks también. Pero ahora la utilizaremos para Rails. A modo informativo, el código de AR es la parte más grande del código total de Rails.

AR soporta la mayoría de las bases de datos tales como Oracle, SqlServer, db2, Postgresql, Sybase, etc.

## **Que sucede debajo de AR:**

1. Genera el SQL
2. El SQL es pasado al RDBMS (Relational Database Management System - Sistema de Gestión de Base de Datos Relacional ) subyacente.
3. El RDBMS compila este SQL
4. Entonces lo ejecuta.
5. Devuelve un enumerable (objeto como array)
6. Crea un objeto modelo AR para cada fila.

Como uno puede ver, el proceso entero de ejecución del SQL podría ser mas lento comparado con la ejecución de un ‘procedimiento almacenado’ en la base de datos (el cual es un código precompilado). Uno siempre puede preguntarse, ¿Podemos correr un “procedimiento almacenado” desde AR directamente?, Sí, podemos. Pero como los procedimientos almacenados son específicos a la base de datos, esto puede volverse una limitación.

Podemos (y frecuentemente necesitamos) correr SQL directamente de AR para acelerar el proceso, pero entonces un debe aprender a escribir buenas consultas SQL e implementar ANSI SQL también para hacer a nuestras aplicaciones independientes del RDBMS.

Antes que nos sumerjamos en el aprendizaje de AR, prestemos atención a los siguientes puntos:

**1. AR es capa de Modelos, es decir, una interface que nos ayuda a manejar nuestra base de datos. Todas las operaciones CRUD (Create, Read, Update, Delete) pueden ser ejecutadas a través de AR. La base de datos relacional puede ser usada con propósitos de almacenaje de datos.**

**2. El código que escribimos aquí consistiría en su mayoría de la lógica de negocios. Obviamente esto es mucho más reutilizable a través de las aplicaciones.**

**3. Mientras trabajamos en cualquier aplicación, uno debería analizar, cuanto necesita ser independiente de la base de datos, cuando utilizar el poder de la flexibilidad, cuando necesitar la velocidad de ejecución y cuando balancear esto.**

**4. La clase ActiveRecord::Migration se usa para fines de Definición de Datos y ActiveRecord::Base se usa para fines de Manipulación de Datos.**

Se cubrirán los siguientes puntos:

1. Conectividad AR-Base de datos.
2. Convenciones sobre configuración.
3. Creando un modelo y varias características CRUD disponibles en AR.
4. Validaciones.
5. Callbacks.
6. Asociaciones de Modelos. Uno-a-uno(one-to-one), uno-a-muchos(one-to-many), muchos-a-muchos (usando opciones **:has-and-belongs-to-many** y **:through**).

## 1. Como AR se conecta a la base de datos

El archivo 'database.yml' tiene todas las opciones de configuración asociadas a la base de datos, las cuales AR utiliza para conectarse.

```
<development>:
  adapter: mysql
  encoding: utf8
  database: sample1_development
  username: root
  password: root
  host: localhost

<production>:
  adapter: mysql
  encoding: utf8
  database: sample1_production
  username: root
  password: root
  host: localhost

<testing>:
  adapter: mysql
  encoding: utf8
  database: sample1_testing
  username: root
  password: root
  host: localhost
```

ActiveRecord puede conectarse a diferentes bases de datos. Solo se requieren los adaptadores apropiados para conectarse.

La separación dada de ambientes (Development/Test/Prod.) a AR como herramienta ORM, les da a los desarrolladores flexibilidad para testear/desplegar aplicaciones en diferentes bases de datos.

Uno podría maravillarse del modo en que rails prefiere usar .yml sobre XML. La razón es la simplicidad en la lectura y la usabilidad de la estructura .yml sobre XML.

Observe los dos formatos con el mismo contenido:

.XML File	.YML File
<pre>&lt;user id="rodo" on="cpu1"&gt;   &lt;first_name&gt;Rodo&lt;/first_name&gt;   &lt;last_name&gt;Schonhals&lt;/last_name&gt;   &lt;user_name&gt;rodolinux&lt;/user_name&gt; &lt;/user&gt;</pre>	<pre>id: rodo computer: cpu1 first_name: Rodo last_name: Schonhals user_name: rodolinux</pre>

Por favor observe que el archivo .yml como un par 'clave - valor'. La sintaxis Ruby puede muy fácilmente convertir este archivo a una **tabla hash** y llevar la conectividad. <http://yaml4r.sourceforge.net/cookbook/> es una gran fuente para los interesados en saber todo acerca de archivos YML.

Rails usa ampliamente los archivos YAML para especificar accesorios los cuales podemos ver en las siguientes sesiones.

`ActiveRecord::Base.establish_connection` es el método que establece la conexión a la base de datos. Acepta un hash como entrada donde la llave `:adapter` debe ser especificada con el nombre del adaptador de base de datos (en minúsculas)

```
ActiveRecord::Base.establish_connection(
  :adapter => "mysql",
  :host => "localhost",
  :username => "root",
  :password => "root",
  :database => "sample1"
)
```

## 2. Convenciones sobre configuración:

Para que la magia de AR trabaje, debemos nombrar nuestras tablas y columnas de acuerdo a convenciones específicas.

**A. Los nombres de los Modelos de clase deberían ser CamelCase (frases o palabras compuestas eliminando los espacios y poniendo en mayúscula la primera letra de cada palabra ) y ser mapeados a la Tabla, la cual es una forma pluralizada de los nombres de los modelos de clase.**

*Por ejemplo:* Si el nombre de un modelo es ‘User’, Rails asumiría que el nombre de la tabla es ‘users’ (el nombre de la tabla en minúsculas). Si nuestro modelo de clase es ‘Person’, el mapeado al nombre de la tabla sería ‘people’, la pluralización de la palabra ‘Person’. ( Las tablas van en plural y el modelo asociado va en singular)

¿Como sabe Rails esto? Con la ayuda de la clase ‘Inflector’, obtenemos la respuesta.

```
c:\sample1> ruby script/console
Loading development environment.

>> Inflector.pluralize('test')
=> "tests"
>> Inflector.pluralize('mouse')
=> "mice"
>> Inflector.pluralize('person')
=> "people"
>> "company".pluralize
=> companies
```

**B. El nombre de la clave primaria debería ser siempre ‘id’, la cual debería ser entero auto-incrementado, case sensitive (se dice que un texto es ‘case sensitive’ si distingue mayúsculas de minúsculas ).**

**C. Las claves foráneas unen las tablas de la base de datos.** Cuando necesitamos un campo de clave foránea en una tabla, la clave toma la forma “{nombre de tabla foránea en singular}\_id”.

Ej.; person\_id con un singular en ingles y un sufijo \_id.

**D. Si se tienen campos “created\_on” o “created\_at” en la tabla, AR automáticamente incluirá la fecha y hora en la creación inicial de la fila. Si se tienen campos “updated\_on” o “updated\_at”, AR automáticamente registrará la**

fecha y hora de modificación del registro. Se debe usar 'datetime' como tipo de dato, no timestamp. Estos campos no deberían tener valores por defecto, ya que Rails se toma el trabajo por nosotros. Estos campos deberían estar disponibles para ser seteados como NULL

Para asociaciones de tablas (también conocido como tablas de intersección) que resuelven relaciones de tipo muchos-a-muchos, usaremos la siguiente convención de nombrado: {nombre de la primer entidad}\_{nombre de la segunda entidad} Ej.: products\_categories.

**Nota:**

Esta convención agrega estandarización, la cual proporciona una gran velocidad al desarrollo. Pero aún, puede uno volver a sorprenderse, si debe obviar/saltar estas convenciones, debido a algunos problemas prácticos (digamos, en caso de que algunas tablas ya existan). Si, hay una forma, que listaremos al final del capítulo.

### 3. Crear Un modelo

Veamos un simple ejercicio para probar este punto.

1. Necesitamos crear una tabla (users) y un modelo (User)
2. La convención de Rails automáticamente mapeará el modelo (User) en la tabla (users).
3. Haremos uso del script 'console' para agregar datos a esta tabla sin tener que escribir ni una línea de SQL.

1) Por favor, cree una tabla en mySql con la siguiente estructura (no se preocupe por el tamaño de cada uno de los campos en este momento, tome las opciones por defecto):

**users**

id	integer
first_name	string
last_name	string
user_name	string
login	string
password	string
phoneNo	integer

2) Luego creamos el modelo "User":

```
railsApps\sample1> ruby script/generate model User
```

Deberíamos obtener la siguiente salida:

```
C:\railsApps\sample1>ruby script/generate model User
exists app/models/
exists test/unit/
exists test/fixtures/
create app/models/user.rb
create test/unit/user_test.rb
create test/fixtures/users.yml
create db/migrate
create db/migrate/001_create_users.rb
```

Como se puede observar, un número de carpetas y archivos fueron creados. No se preocupen demasiado respecto de los archivos/carpetas 'test', 'app' y 'db' por ahora. Concentrémonos en la carpeta app/models y el archivo 'user.rb'.

#### **user.rb**

```
class User < ActiveRecord::Base
end
```

ActiveRecord::Base es la clase de la cual heredan todos los modelos. Esta herencia le da a nuestra clase User una tremenda cantidad de funcionalidad. Como hemos aprendido, luego veremos que esas clases contendrán toda la lógica de negocios de nuestras aplicaciones.

En este caso, la clase 'User' automáticamente mapea a la tabla 'users' según las convenciones Rails.

### 3) Ahora juguemos con nuestro modelo, "User":

```
railsApps\sample1> ruby script/console
Loading development environment.
```

```
>> user1 = User.new
>> user1.first_name = "Satish"
>> user1.last_name = "Talim"
>> user1.email = "satish.talim@gmail.com"
>> user1.save
=>true
```

```
>> user2 = User.create(:first_name => 'Marcos' , :last_name =>
'Ricardo')
=>true
```

# Note la diferencia entre **New** y **Create**.

# El método 'new', crea un nuevo registro en memoria seguido de un método 'save' para salvarlo a la base de datos mientras que 'create' no solo genera sino que también salva el nuevo registro en la base de datos.

```
>> user.update_attributes(:first_name => 'Luiz', :last_name =>
'Biagi' )
==>true
```

# Pueden ir agregando/modificando nuevos registros de este modo.

Los objetos 'user1' y 'user2' son mapeados en dos registros de la tabla 'users'. Mientras que estamos ejecutando estas simples instrucciones, debajo, AR realiza la magia de generar/ejecutar las sentencias SQL. Estas operaciones son logeadas en un archivo log.

## Una nota respecto del logueado y los archivos Log.

Por favor chequee el archivo **/log/development.log** en una ventana separada para ver el SQL generado.

Rails tiene un dispositivo de logueado en el framework, este expone un objeto [Logger](#) a todo el código en una aplicación Rails. Donde sea que ejecutamos operaciones en AR, este genera un log de actividades.

Los logs nos ayudan a monitorear/administrar la conducta/performance de nuestra aplicación.

Un ambiente de desarrollo logueara a 'log/development.log', una aplicación bajo pruebas a 'log/test.log' y una aplicación en producción a 'log/production.log'.

Podemos generar mensajes a niveles de advertencia, informes, errores, y fatalidades.

### log/development.log:

Si se observa las sentencias SQL generadas, éstas tienen una secuencia criptica de caracteres conocidas como secuencias de escape que son utilizadas por defecto. Simplemente seteando el `colorize_logging` a falso hace que esto tenga una lectura mas sencilla.

```
ActiveRecord::Base.colorize_logging = false
```

O mejor aún, se puede abrir el archivo `config/environment.rb` y al final del archivo (última línea) tipear la sentencia descripta arriba y guardar el archivo. Necesitamos salir de la consola y entrar para ver los efectos.

**Nota:** después de cada ejecución de comandos en consola, continúe refrescando este log y observando el SQL generado por AR.

## Use los métodos Finder para obtener registros de la base de datos

```
>>User.find(1) # find selects by 'id' (default integer primary key)
>># find with ids 2, 4, and 7
>> User.find(1,4,7)

>> user = User.find(:all)
>> user.each{ |rec| puts rec.first_name}
>> user1 = User.find(:first) #will fetch you the first record
```

**Nota:** `find(:all)` debería devolver un objeto array el cual podemos iterar usando el método `'each'`.

Espero que capture alguna idea de cómo el modelo obtiene los registros deseados de la base de datos cuando un request del navegador es efectuado.

Uno puede usar SQL directamente:

```
>> User.find_by_sql "SELECT usr_name, email FROM users"
```

## Métodos Finder dinámicos

Observe el método `'find_by_first_name1'`. AR agrega un **finder dinámico** para cada atributo disponible en la tabla.

```
>> User.find_by_first_name 'Satish'
```

### **TIP: MÉTODOS FINDER DINÁMICOS**

Mientras que el usar algo como `User.find_by_user_name` es muy legible y fácil, esto es una sobrecarga para Rails. AR genera dinámicamente estos métodos en los métodos perdidos (`method_missing?`) y es bastante lento.

Usar `MyModel.find_by_sql` directamente, o `MyModel.find`, es más eficiente.

## Encontrando registros basados en 'condiciones'

```
>> User.find :all, :conditions => "first_name LIKE 'S%' ", :limit => 10
```

**TIP:** Optimizar en tiempo de ejecución. Deberíamos obtener solo la información que necesitamos. Se pierde gran cantidad de tiempo por correr `selects` por datos que no se usan.

Explorar la opción `:select` (extrae solo aquellos campos que vamos a usar), `:limit` y `:offset` (si el tamaño del registro es grande, esta opción es obligatoria).

## Métodos Personalizados

### **user.rb**

```
class User < ActiveRecord::Base
  def full_name
    name = first_name + ' ' + last_name
  end

  def self.find_all_ordered
    find :all, :order => 'last_name, first_name'
  end
end
```

Observe estos métodos cuidadosamente. Aquí no estamos escribiendo una sola línea que se encargue de mostrar los registros. Eso es tarea de las 'Vistas' las cuales aprenderemos luego. Puedo usar la misma clase para múltiples aplicaciones web si esta clase puede volverse genérica.

**La 'Reusabilidad' es el núcleo de la Programación Orientada a Objetos.**

## Validaciones

Agregar validaciones a nuestro código significa, que el registro no será creado o salvado si las condiciones no se cumplen o satisfacen. Las validaciones basadas en modelos nos ayudan a que nuestras bases de datos sean consistentes.

Cambie 'user.rb' como se muestra abajo:

### User.rb

```
class User < ActiveRecord::Base
  EMAIL_EXP = /^ [A-Z0-9._%~]+@[A-Z0-9.-]+\ . [ A-Z] {2,4} $/i
  validates_format_of :email, :with => EMAIL_EXP
  validates_presence_of :user_name, :message => "User name can not
    be Blank"
  validates_uniqueness_of :user_name
  validates_numericality_of :phoneNo, :allow_nil => true
end
```

Observe la validación **validates\_numerically\_of** con **:allow\_nil**. Esto salta la validación si el atributo es nil (por defecto es falso). Note que las columnas de cadenas vacías, los *fixnum* y *float* son convertidos a nil.

Dado que hemos modificado el User.rb, para recargar la clase debemos salir de la consola y reingresar.

```
>> user = User.new
>>User.new(:phoneNo => nil).valid?
=> true
>> user.save
>> false

>> user.errors
```

Un objeto [Errors](#) es automáticamente creado para cada AR.

```
User2 = User.new("first_name" => "David", "phoneNo" => "what?")
User2.save #=> false (and doesn't do the save)
User2.errors.empty? #=> false
User2.errors.count
User2.errors.on "user_name" #=> "User name can not be Blank"
User2.errors.on "email"
```

```
User2.errors.each_full {|msg| puts msg}
```

Para mas ejemplos y varias opciones de validación, se aconseja leer:

<http://rails.rubyonrails.com/classes/ActiveRecord/Validations/ClassMethods.html>

## Callbacks

Los callbacks son similares a los [triggers](#) de las bases de datos relacionales.

En cualquier aplicación de negocios, la ‘validación’ y ‘autenticación’ de un registro podría involucrar diferentes actividades/procesos. Las validaciones podrían estar relacionadas a los niveles de llenado, mientras que las autenticaciones ser aplicadas al formulario entero. Ej. En cualquier transacción de tarjeta de crédito, podemos agregar validaciones a nuestro modelo para chequear si un nombre, numero de tarjeta, fecha de expiración, etc., sean ingresadas apropiadamente, pero la autenticidad de la tarjeta (obtener una confirmación de validez de una tarjeta) necesita ser llevada, antes que la transacción sea llevada a cabo.

Este ‘pre-proceso’ puede ser realizado usando “[callbacks](#)”. El pos-proceso podría ser enviar emails a los clientes con el estado de la transacción. Es por eso que esto se llama [Callbacks](#).

Durante el ciclo de vida de una aplicación, podemos llegar a los siguientes eventos.

<code>after_validation</code>	Después que todos los procedimientos de validación estén completos.
<code>after_validation_on_create</code>	Después de la validación de los nuevos objetos.
<code>after_validation_on_update</code>	Después de la validación que los objetos existentes que sean actualizados.
<code>before_validation</code>	Antes que el objeto sea validado.
<code>before_validation_on_create</code>	Antes de la validación de nuevos objetos.
<code>before_validation_on_update</code>	Antes de la validación que los objetos existentes que sean actualizados.
<code>before_create</code>	Antes que el objeto sea agregado a la base de datos.
<code>before_destroy</code>	Antes que un objeto sea destruido.
<code>before_save</code>	Antes que un objeto sea salvado a la base de datos, tanto por creación como por actualización.

<code>before_update</code>	Antes que un objeto sea modificado en la base de datos.
----------------------------	---

## Cálculos

¿Se desea obtener el máximo, mínimo, promedios o sumas para los datos en las tablas? Los cálculos de AR hacen todo esto posible. La mayoría pueden ser realizados con opciones extras. Las mismas pueden leerse en:

<http://api.rubyonrails.org/classes/ActiveRecord/Calculations/ClassMethods.html>

## Asociaciones de Modelos

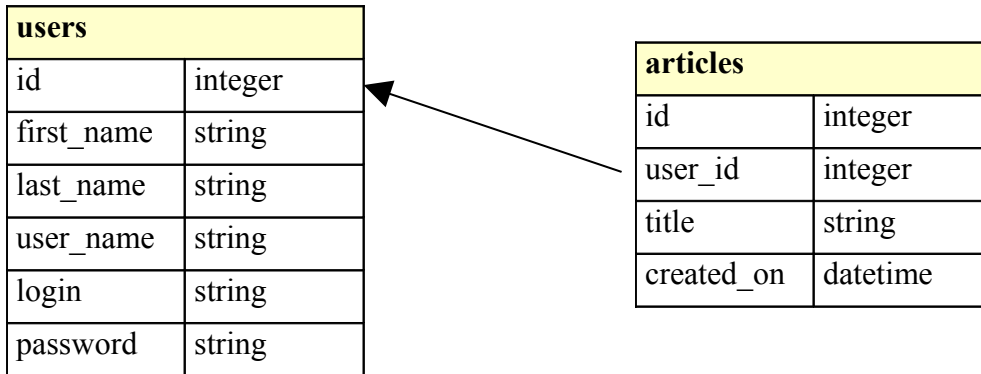
Somos conscientes que las aplicaciones web contienen un montón de tablas que están asociadas (relacionadas) unas con otras. Con la [normalización](#) como la llave, evitamos redundancia. Estas asociaciones son de diferentes tipos. Ej.: uno-a-uno, uno-a-muchos, muchos-a-muchos, polimórficas, etc. AR actualmente, no puede leer estas asociaciones automáticamente, por lo que como programadores, necesitamos definir las en los modelos, lo que nosotros llamamos **Asociaciones de Modelos**.

## Asociación one-to-one

Para el caso del ejemplo, por favor asumimos, **el usuario (o autor), en nuestra tabla 'users' puede escribir al menos un 'artículo'**. Para establecer esta asociación necesitamos

1. Tener 2 tablas, 'users' y 'articles' (ya tenemos la tabla 'users').
2. Necesitamos 2 modelos 'User' y 'Article' para mapear sobre 2 tablas ( ya tenemos el modelo/clase 'User')
3. Definir la asociación entre estos modelos.
4. Ver como trabaja la asociación en AR agregando/actualizando registros en estas tablas.

1) Definir la tabla 'articles' (ya tenemos 'users')



Crear la tabla 'articles' en mySql como se describe arriba. La columna 'user\_id' es nuestra clave foránea en la tabla 'articles' (siguiendo la convención de nombrado de Rails).

2) Ahora que las tablas están seteadas, necesitamos al modelo 'Article', y lo creamos de este modo:

```
railsApps\sample1> ruby script/generate model Article
```

3) Editar ambos modelos/clases y agregar los siguientes métodos asociativos.

#### user.rb

```
class User < ActiveRecord::Base
  has_one :article
end
```

#### article.rb

```
class Article < ActiveRecord::Base
  belongs_to :user # clave foránea user_id
end
```

**has\_one** :article y **belongs\_to** :user son métodos (no definiciones de métodos), que toman símbolos como parámetros. El método **belongs\_to** agrega una asociación, llamada **user**, a **article**.

Note que **:article** es singular ya que la asociación es de la forma 'has\_one'. En caso que la asociación sea uno-a-muchos, esto sería (has\_many **:articles**).

```
railsApps\sample1> ruby script/console
```

```
Loading development environment
```

```
>> a1 = Article.new
```

```

>> a1.title = "Rodo on Rails"
>> user = User.find(1)
>> a1.user_id = user
>> a1.save
>> a1.user.first_name
>> a2.Article.create(:title => "Learning Ruby with RodoRails")
>> a2.user_id = user
>> a2.user.user_name
>>

```

\*\* Observe el SQL generado en cada caso (use el archivo de log).

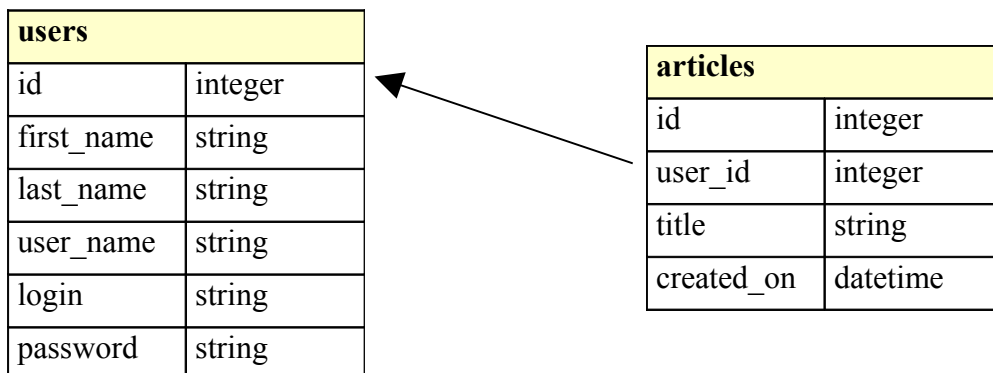
## Asociación uno-a-muchos

Considere un caso donde **‘Un usuario puede escribir muchos artículos’**. Esto indica que tenemos una asociación **uno-a-muchos**.

Para entender esta asociación haremos lo siguiente:

1. Crear 2 tablas, ‘users’ y ‘articles’ (ambas tablas están disponibles)
2. Necesitamos a los modelos ‘User’ y ‘Article’ para mapear sobre las 2 tablas.
3. Definir la asociación entre estos modelos.
4. Ver como la asociación trabaja en AR agregando/modificando registros en esas tablas.

- 1) Definir la tabla ‘articles’ (ya tenemos ‘users’)



2) Es tiempo de actualizar nuestros modelos. Lo ideal sería destruir el modelo previamente creado y crear uno nuevo con el mismo nombre:

```
railsApps\sample1> ruby script/destroy model User
```

Daremos el mismo tratamiento al modelo 'Article'. Esto me asegura que todas las referencias creadas anteriormente serán destruidas. Esto NO borra nuestras tablas.

Ahora crearemos ambos modelos con:

```
"ruby script/generate model User"
```

```
"ruby script/generate model Article"
```

3) Definiremos la correcta asociación entre los modelos.

#### article.rb

```
class Article < ActiveRecord::Base
  belongs_to :user
end
```

#### user.rb

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

**belongs\_to** :user y **has\_many** :articles son los métodos ( no definiciones de métodos) que toman al símbolo como un parámetro. El método **belongs\_to** agrega una asociación, llamada **user**, a **article**.

Preste atención que, **:articles** es plural, porque la asociación es de la forma 'has\_many', en caso de que sea one-to-one, esta podría ser (has\_one **:article**).

Chequee lo siguiente en su consola ruby: CHEQUEE los sql generados después de la ejecución de estas instrucciones también.

```
railsApps\sample1> ruby script/console
```

```
Loading development environment
```

```
>> a1 = Article.new
>> a1.title = "Instant Ruby"
>> user = User.find(1)
>> user.articles << a1
>> user.save
```

```
>>user
```

Del ejemplo de arriba, es bastante evidente que la instancia del modelo padre ('user') obtiene los métodos del modelo hijo ('articles').

Podemos agregar/actualizar registros de este modo:

```
>> user.articles.create( :title => "The Ruby Way")
```

Los métodos Finder trabajan como antes, pero noten la asociación.

```
>> user = User.find(1)
```

```
>> user.articles.count
```

```
>> user.articles
```

\*\* Observe el SQL generado en cada caso.

Una cuestión mas que se plantea, y ésta es cuando borramos un 'user' de la tabla users: ¿Qué pasa con los registros en 'articles', ya que hay una asociación, pueden ser borrados automáticamente? **NO**.

La solución es explorar los archivos de ayuda :) y ahí obtendremos lo que buscamos. Usaremos la opción de dependencia: **:dependent**.

### user.rb

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

```
>> u = User.create(:first_name => "John", :last_name => "Alter")
```

```
>> u.articles.create(:title => "Core Java")
```

```
>>u.articles.create(:title => "Developing Java Applications")
```

```
>>u.articles
```

```
>>u.destroy
```

Cuando se refresque la consola, se debería poder ver que el user 'John' y todos los libros de su autoría fueron borrados de la tabla 'articles'.

## Carga urgente

La [documentación](#) de ActiveRecord dice: "La carga urgente es una forma de encontrar objetos de cierta clase y un número de asociaciones nombradas con una simple llamada SQL."

Esta es una de las formas mas fáciles de prevenir el problema 1+N en el cual para obtener los 100 post de un autor, disparamos a la base de datos 101 consultas. A través de la carga urgente, estas 101 consultas puede reducirse a 1."

Para cortar esta corta historia, supongamos que queremos obtener todos los artículos de un usuario específico, con el conocimiento actual que hemos obtenido a lo largo de este sencillo manual.

```
>> user = User.find(:first)
>> user.articles
```

Se puede ver que las 2 consultas fueron lanzadas para obtener el resultado. Una mejor forma es hacerlo mediante la carga urgente, usando la opción **:include**.

```
>>user1 = User.find(:first, :include=> :articles)
>>user1.articles
```

En una sola consulta obtenemos el conjunto de resultados. ¿Observaste el ‘LEFT OUTER JOIN’?.

Una cosa, necesitamos recordar que AR ignora la directiva **:select** si usamos **:include**. Esto puede ser un poco complicado. Pero siempre hay una alternativa. Usar la opción **:joins**

```
>>user2 = User.find(:first,
:select => "users.first_name, articles.title",
:joins => "left outer join articles on users.id =
articles.user_id")
```

```
>>user2.first_name
```

La consulta generada por el ‘**find**’ es como sigue:

```
SELECT users.first_name, articles.title FROM `users` left outer
join articles on users.id = articles.user_id LIMIT 1
```

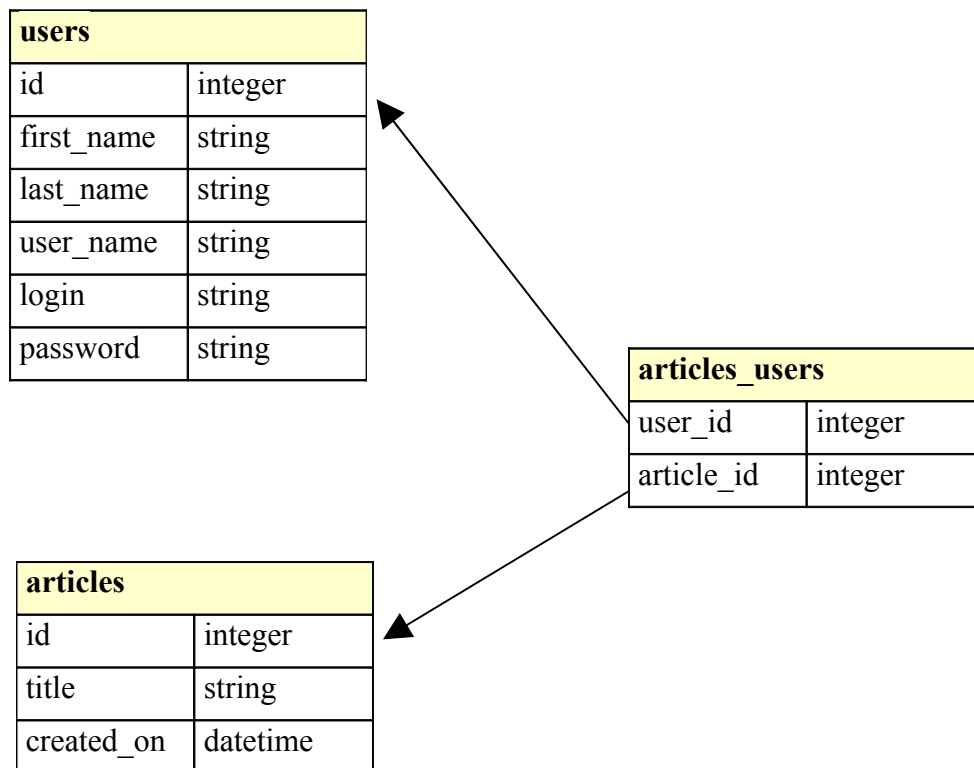
## Asociación many-to-many (muchos-a-muchos, usando `has_and_belongs_to_many`)

Continuando con nuestro modelo de estudio, en la práctica, sabemos que un ‘user’ (o autor) puede escribir ‘muchos artículos’ y un ‘artículo’ puede tener ‘muchos usuarios (o autores)’.

En términos de diseño podemos llamar a esto como una ‘asociación muchos-a-muchos’. Para ver la asociación descrita arriba, deberíamos:

1. Usar un método **has-and-belongs-to-many** (o ‘**has\_and\_belongs\_to\_many**’ cuando hablamos sobre esto)
2. Necesitamos crear una nueva tabla de concatenación (uniendo los nombres de las tablas en orden alfabético), la cual debería contener las claves foráneas a las tablas destino (mas convenciones, como se ve).

**Nota:** Es necesario continuar observando y estudiando el SQL generado en los archivos log.



### article.rb

```
class Article < ActiveRecord::Base
  has_and_belongs_to_many :users
end
```

### user.rb

```
class User < ActiveRecord::Base
  has_and_belongs_to_many :articles
end
```

```
>> author1 = User.create(:first_name => "Author - 1")
>> author2 = User.create(:first_name => "Author - 2")
```

```
>> a1 = Article.create(:title => "Book - 1")
>> a2 = Article.create(:title => "Book - 2")
```

```
>> author1.articles << a1
>> author1.articles << a2
>> a1.authors << author2
```

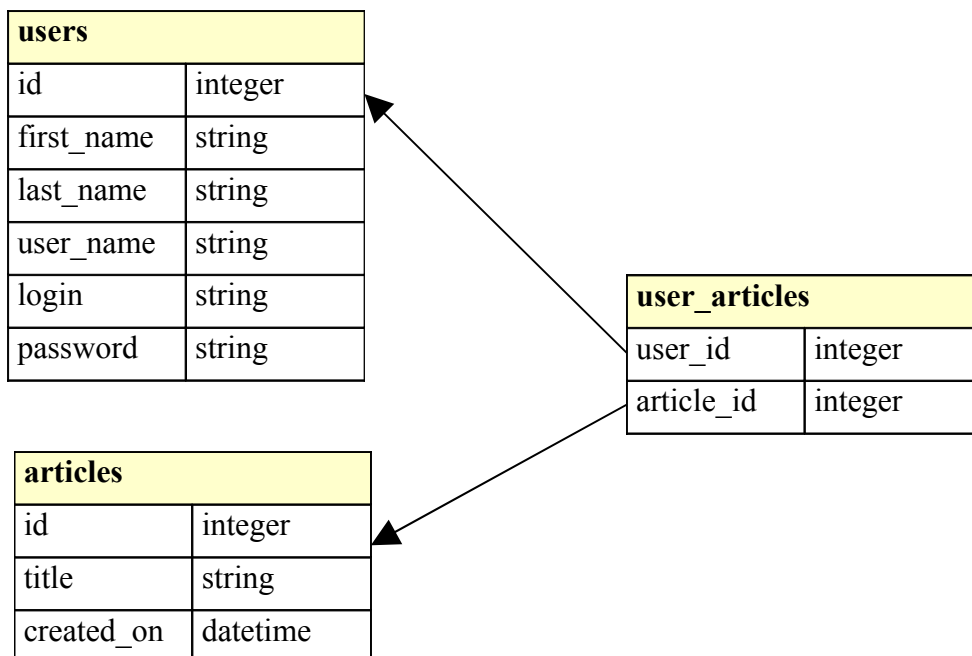
Vamos a sumergirnos un poco mas y explicar la misma asociación pero con diferentes diseño de base de datos y con un método opcional de la clase ActiveRecord::Base (:through).

## Asociación muchos-a-muchos (usando :through)

Necesitamos modificar de nuevo nuestro diseño (Este es un mundo ágil: **lo único constante es el cambio**).

Haremos 2 cosas

1. Crear una tabla mas llamada **'user\_articles'** y **\*NO\*** **'users\_articles'**
2. Crear una clase/modelo **'UserArticle'** y actualizar la correcta relación entre estos modelos.



```
railsApps\sample1> ruby script/generate model user_article
```

### user.rb

```
class User < ActiveRecord::Base
  has_many :user_articles (*)
end
```

### user\_article.rb

```
class UserArticle < ActiveRecord::Base
  belongs_to :user
  belongs_to :article
end
```

### article.rb

```
class Article < ActiveRecord::Base
  has_many :user_articles (*)
end
```

(\*) Notar el plural **user\_articles** y **\*NO\*** como **user\_article**

Asumimos que ya **hemos agregado algunos registros y las tablas de usuarios y artículos.**

Asumimos que tenemos que encontrar todos los usuarios (autores) para un artículo específico.

Idealmente deberíamos obtener este resultado en dos etapas:

```
>> a = Article.find(1) # el primer artículo en la tabla
>> a.users             # y esto debería devolver todos los usuarios
                       # de ese artículo
```

Pero con el mapeado descrito arriba, nuestros pasos serían:

```
>> a = Article.find(1) # el primer artículo en la tabla
>> a.user_articles     # esto debería devolvernos un arreglo, por último
>> a.user_articles[0].user # aquí obtenemos el nombre del usuario actual
```

Como se ve, necesitamos pasos extras para realizar el trabajo? Observen el sql generado, necesitaremos optimizarlo también.

ActiveRecord tiene una solución para esto, y es usando la relación **has\_many :through**

Las asociaciones **through** le permiten a un modelo ser accedido a través de una asociación intermedia.

Modifiquemos nuestro modelo para que se vea así:

#### **article.rb**

```
class Article < ActiveRecord::Base
  has_many :user_articles
  has_many :users, :through => user_articles
end
```

Ahora nos tomamos el trabajo de recargar el ambiente de desarrollo.

La consulta tendrá un [Inner Join](#), es decir lo que debería ser. Podemos además agregar otro usuario al mismo artículo.

```
>> a = Article.find(1)
>> a.users
>> a.users << User.find(3) # esto insertará un usuario un usuario para ese
                          # artículo agregando un registro en
                          #la tabla user_articles.
```

#### **Cesión:**

1. Discuta las ventajas de utilizar la opción `:through`.
2. Explore la Web/Libros y discuta las asociaciones polimórficas.
3. Lea y Discuta sobre las “[Transacciones](#)” de ActiveRecord.

## TIPS: SOBRECARGANDO CONVENCIONES.

**1. Un método alternativo de nombre de tablas:** Si no se desea nombrar a una tabla en plural, es posible sobrecargarlo en el archivo de la clase/modelo usando el método 'set\_table\_name'.

```
class User < ActiveRecord::Base
  set_table_name "usuarios"    # por convención sería 'users'
                              # alternativamente podríamos decir
                              # self.table_name = "usuarios"
end
```

**2. Una clave primaria alternativa**

```
class User < ActiveRecord::Base
  set_primary_key "usuarioID"  # por convención sería 'id'
                              # pero debe ser una clave Integer
                              # alternativamente podríamos decir
                              # self.primary_key "usuarioID"
end
```

**3. Si deseamos desactivar globalmente la pluralización,** debemos abrir config/environment.rb y al final del archivo, escribir:

```
ActiveRecord::Base.pluralize_table_names = false
```

Lea más: <http://wiki.rubyonrails.com/rails/pages/HowToUseLegacySchemas>

Como AR es la parte mas grande de Rails 2.0, hemos intentado cubrir los detalles al máximo. El siguiente capitulo se refiere a la parte de Definición de Datos con las migraciones.