

MIGRACIONES

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

Las migraciones es la parte DDL de ActiveRecord. Las migraciones pueden gestionar la evolución de un esquema usado en varias bases de datos físicas. Es una solución al problema común de agregar un campo para hacer que una característica nueva trabaje en nuestra base de datos, pero estando inseguros de cómo afectará este cambios a los otros desarrolladores y al servidor de producción. Las migraciones combinan poder y simplicidad a la hora de coordinar los cambios en los esquemas y en los datos.

Con las migraciones, uno puede describir las transformaciones en clases autocontenidas que pueden ser chequeadas por sistemas de control de versión y ser ejecutadas contra otras bases de datos que pueden estar una, dos o 5 versiones detrás.

Esto tiene muchos usos, incluyendo:

- Equipos de desarrolladores – si una persona realiza un cambio en un esquema, los otros desarrolladores solo necesitan actualizar y correr “`rake db:migrate`”.
- Servidores de producción – correr “`rake db:migrate`” cuando liberamos una nueva versión le permite a la base de datos estar al día también.
- Múltiples máquinas – si uno desarrolla tanto en un desktop como en una laptop, o en mas de un lugar, las migraciones pueden ayudar a tenerlos todos sincronizados.

ActiveRecord::Migration es la clase base que hace la magia.

Lo que planeamos cubrir en este capítulo.

1. Como trabajan las migraciones
2. Una nota sobre el esquema de base de datos
2. Algunos tips prácticos
3. Una nueva adición de Migraciones Sexys

Nota del autor: Usa un MySQL Front, una herramienta windows MySQL para mejor administración.

El formato general del archivo de migración es como sigue:

```
def self.up
  create_table :table, :force => true do |t|
    t.column :name, :string
    t.column :age, :integer, { :default => 42 }
    t.column :description, :text
    # :string, :text, :integer, :float, :datetime,
    :timestamp, :time, :date,
    # :binary, :boolean
  end

  add_column :table, :column, :type
  rename_column :table, :old_name, :new_name
  change_column :table, :column, :new_type
  execute "SQL Statement" # se puede ejecutar las sentencias
  SQL directamente
  add_index :table, :column, :unique => true, :name =>
'some_name'
  add_index :table, [ :column1, :column2 ]
end

def self.down # rollbacks changes
  rename_column :table, :new_name, :old_name
  remove_column :table, :column
  drop_table :table
  remove_index :table, :column
end
```

La documentación completa de la API para Migrations se encuentra en:

<http://api.rubyonrails.com/classes/ActiveRecord/Migration.html>

Tip: Utilice bases de datos indexadas para acelerar las consultas. Rails solo indexa los índices primarios, por lo que necesitamos tomar cada esfuerzo para optimizar la velocidad de ejecución.

Las Migraciones soportan todos los tipos de datos básicos en MySQL:

Rails	MySql
:binary	blob
:boolean	tinyint
:date	date
:datetime	datetime
:decimal	decimal
:float	float
:integer	int11

:string	varchar(255)
:text	text
:time	time
:timestamp	datetime

¿Como creo un archivo de Migración?

Para jugar con las migraciones, creemos un proyecto separado:

```
c:\railsApps> rails -d mysql sample2
```

Asumimos que se han hecho los cambios apropiados en database.yml

```
C:\railsApps\sample2> rake db:create:all
```

Esto creara las bases de datos para todos los ambientes

Hay varias maneras de ir generando los archivos de migración:

1. Usar “**ruby script/generate migration** <migration_file_name>”
2. Cuando generamos un ‘modelo’ usando ‘*ruby script/generate model <model name>*’ un script de migración para la tabla es creado automáticamente.

Los archivos de migración son almacenados en la carpeta <project>/db/migrate/

Hagamos un ejercicio. Supongamos que queremos crear nuestra tabla ‘users’ en nuestra base de datos ‘sample2_development’. Usaremos migraciones para realizar este trabajo.

```
c:\railsApps\sample2> ruby script/generate migration add_user_table
```

Encontraremos un archivo llamado **001_add_user_table.rb** creado en **sample2\db\migrate** con el siguiente detalle:

001_add_user_table.rb

```
class AddUserTable < ActiveRecord::Migration
  def self.up
    end

  def self.down
    end
end
```

El archivo muestra como todos los archivos de migración tienen **dos métodos de clase**, **up** y **down**, que describen las transformaciones requeridas para **implementar** o **remover** la migración. Estos métodos consisten de métodos específicos tales como `add_column`, `remove_column`, etc, pero pueden también contener un código Ruby regular para la generación de datos requeridos para las transformaciones.

Cuando corremos las migraciones, los archivos son prefijados con un número de secuencia. Por lo tanto, la primer migración que creamos se llamará “001_{file_name}.rb”, el siguiente será “002_{file_name}.rb” y así.

Edite `\db\migrate\001_add_user_table.rb` como sigue:

001_add_user_table.rb

```
class AddUserTable < ActiveRecord::Migration
  def self.up
    create_table :users do |t|
      t.column :first_name, :string, :limit => 20
      t.column :last_name, :string, :limit => 20
      t.column :created_on, :timestamp
      t.column :updated_on, :timestamp
    end
  end
  def self.down
    drop_table :users
  end
end
```

Omitiremos las columnas ‘user_name’ y ‘password’ por ahora.

```
c:\railsApps\sample2> rake db:migrate
```

La siguiente imagen muestra la migration exitosa (con el numero de segundo que tomó el efectuar la operación)

```
C:\railsApps\sample2>rake db:migrate
<in C:/railsApps/sample2>
== 1 AddUserTable: migrating =====
-- create_table(:users)
   -> 0.0470s
== 1 AddUserTable: migrated (0.0470s) =====

C:\railsApps\sample2>
```

Observe la siguiente imagen de nuestra base de datos:

Table	Records	Size	Created	Updated	Engine
schema_info	1	2 KB	2008-04-13 05:40:40	2008-04-13 05:40:41	MyISAM
users	0	2 KB	2008-04-13 05:40:40	2008-04-13 05:40:40	MyISAM

Muestra 2 tablas creadas. Una es la tabla ‘users’ y la otra es ‘schema_info’. Hablaremos de ‘schema_info’ en un momento.

Nota: Ya sea al haber creado una base de datos o haber ejecutado una migración por primera vez, se crea un archivo ‘db\schem.rb’. Este ‘schem.rb’ es la fuente autorizativa para nuestro esquema de base de datos. Mas de esto, después.

La siguiente imagen es de nuestra tabla ‘users’. Observe el campo clave primario ‘id’. Pensemos que no hay referencia al campo ‘id’ en nuestro archivo de migración. Fue creado automáticamente. Sabemos que, como una convención Rails, cada tabla debería tener un entero autoincremental como clave primaria.

Name	Type	Null	Default	Extra
id	int(11)	No		auto_increment
first_name	varchar(20)	Yes		
last_name	varchar(20)	Yes		
created_on	datetime	Yes		
updated_on	datetime	Yes		

¡No hemos especificado aun que migración correr! ¿Cómo tomamos el archivo 001_add.... Y que pasará cuando los otros archivos de la migración estén disponibles? La razón es otra vez la misma, la conveniencia de “**la convención sobre configuración**”!!

1. Cuando ejecutamos “`rake db:migrate`” for primera vez, una tabla “`schema_info`” se autogenera con un solo campo llamado ‘versión’ y tendrá un único registro. El valor de este registro contiene la versión actual de la migración de la base de datos, por lo que en este caso, ‘1’ es almacenado ahí.
2. La siguiente vez cuando corramos ‘`rake db:migrate`’, este buscará un número superior al ultimo disponible en la tabla ‘`schema_info`’ y esa migración será ejecutada.
3. **Scale UP:** en otras palabras, supongamos que tenemos creados 5 archivos de migración, pero que actualmente hemos migrado los 2 primeros. Para este caso podemos pasar “`rake db:migrate VERSION = 5`” y nuestros datos serán escalados secuencialmente desde 3 a 4 a 5 (asumiendo que nuestros archivos .rb son sintácticamente correctos).
4. **Roll back:** trabaja de de la misma manera que antes ‘`rake db:migrate VERSION=X or 0`’, donde X viene a ser cualquier entero que queremos degradar, siendo 0 para limpiar completamente la base de datos.
5. El estilo de nombrado de archivos según convenciones es **AddNombreColumnaToNombreTabla**

Alterando las columnas de la tabla

Alteremos la definición de la tabla y agreguemos las dos columnas faltantes (`user_name` y `password`) a nuestra tabla ‘`users`’. Observemos la convención de nombrado de archivos que usaremos.

```
c:\railsApps\sample2> ruby script/generate migration AddUser_namePasswordToUser
user_name:string Password:String
```

Encontraremos que lo generado tiene todo el código relevante para nosotros:

```
002_add_user_name_password_to_user.rb

class AddUserNamePasswordToUser < ActiveRecord::Migration
  def self.up
    add_column :users, :user_name, :string
    add_column :users, :Password, :string
  end

  def self.down
    remove_column :users, :Password
    remove_column :users, :user_name
  end
end
```

```
c:\railsApps\sample2> rake db:migrate
```

Ahora si observamos en nuestra base de datos, la tabla fue alterada tal como esperábamos.

Hasta ahora tenemos 2 migraciones. Supongamos que necesitamos volver hacia atrás un paso (por el solo hecho de saber como se hace), por lo que escribimos:

```
c:\railsApps\sample2> rake db:migrate VERSION = 1
```

Hemos vuelto al estado donde las 2 columnas (user_name y password) no existían. Es claro a esta altura, que el **método self.down es ejecutado para todos los archivos de migración hasta alcanzar la versión 1.**

Nota: una vez que se haya realizado esta operación, volver a la versión 2, con el comando 'rake db:migrate'.

Agregar índices a la base de datos a través de migraciones.

Si una tabla es grande, y si necesitamos buscar en otra columna que no sea **id**, idealmente deberíamos tener un índice en esa columna. Para agregar un índice, crearemos un archivo de migración que creará los índices. Esto mantiene nuestras migraciones enfocadas.

```
class AddIndexToUserTable < ActiveRecord::Migration
  def self.up
    add_index :users, :user_name, :unique => true
  end

  def self.down
    remove_index :users, :user_name
  end
end
```

Agregar datos a la tabla

David Hannson recomienda crear archivos de migración separados que cargan los datos en los esquemas existentes antes que cambiar los esquemas en si.

Las migraciones tienen acceso a nuestras clases/modelo. Esto hace que sea fácil crear migraciones que manipulen los datos en nuestra base de datos de desarrollo.

```

class AddUsersData < ActiveRecord::Migration
  def self.up
    down # delete the previous entries and create new ones.
    user = User.create(:first_name => "John" , last_name => "Alter")
    user.save!
  end

  def self.down
    User.delete_all
  end
end

```

Una nota sobre nuestro esquema de base de datos Schema.rb

```

\db\schema.rb – No editar este archivo

ActiveRecord::Schema.define(:version => 2) do

  create_table "users", :force => true do |t|
    t.string "first_name", :limit => 20
    t.string "last_name", :limit => 20
    t.datetime "created_on"
    t.datetime "updated_on"
    t.string "user_name"
    t.string "Password"
  end
end

```

Como mencionamos antes, encontraremos que el esquema de la base de datos se actualiza cuando corremos una migración.

Este archivo es autogenerado desde el estado actual de la base de datos. En vez de editar este archivo, se debe usar la característica de migraciones de AR para modificar incrementalmente nuestra base de datos ,y entonces regenerar esta definición de esquemas.

Notemos que este schema.rb es el fuente autorizante de nuestro esquema de base de datos. Si necesitamos crear la base de datos en otro sistema, deberíamos usar **db:schema:load** y no correr todas las migraciones desde el comienzo, ya que esto es un intento fallido e insostenible (debido principalmente a que cuantas mas migraciones, corramos, mas lento correrán para aplicar los cambios).

Migraciones sexys

Rails 2.0 viene con una optimización de código en las migraciones, conocido como migraciones sexys o Sexy Migrations.

Con esta nueva **sintaxis** podemos declarar el tipo de la columna primero, lo que nos permite declarar múltiples columnas del mismo tipo en una sola línea.

Con este ejemplo tomamos la oportunidad de analizar como una clave foránea puede agregarse a una migración.

```
c:\railsApps\sample2> ruby script/generate migration CreateAddresses
```

003_create_addresses.rb (agregar el código descrito debajo)

```
class CreateAddresses < ActiveRecord::Migration
# Create a table holding addresses of Users
  create_table :addresses do |t|
    t.references :user, :null => false (*)
    t.string :add1, :add2, :city, :state
    t.timestamps
  end
  def self.down
    drop_table :posts
  end
end
```

(*) aquí vemos como se crea la clave foránea.

El código que sigue es la forma original, pero observemos la diferencia entre los dos archivos.

```
class CreateAddresses < ActiveRecord::Migration
# Create a table holding blog posts
  create_table :address do |t|
    t.column :user_id, :integer, :null => false
    t.column :address, :string
    # Standard auto-magic columns
    t.column :created_at, :datetime
    t.column :updated_at, :datetime
  end
  def self.down
    drop_table :posts
  end
end
```

Comparemos esta migración con la otra que creamos al principio. ¿Acaso no es un poco mas concisa?

Table-Properties for sample2_development: addresses

Name	Type	Null	Default	Extra
id	int(11)	No		auto_increment
user_id	int(11)	No	0	
add1	varchar(255)	Yes		
add2	varchar(255)	Yes		
city	varchar(255)	Yes		
state	varchar(255)	Yes		
created_at	datetime	Yes		
updated_at	datetime	Yes		

Algunos puntos para tener en cuenta.

- 1) Mantener las migraciones simples y limitadas a una sola tarea por vez. Idealmente, deberíamos poder reconocer el cambio en el nombre de un archivo sin tener que ver su contenido.
- 2) No crear archivos de migración con nombres que contengan "?". No funciona.
- 3) Podemos especificar 3 opciones cuando se definen la mayoría de las columnas en una migración.

Las opciones son:

:null => true or false

Si es falso, la columna subyacente tiene una limitación no nula agregada. (si la base de datos lo soporta).

:limit => size

Define el limite en el tamaño del campo, Esto básicamente agrega la cadena (size) a la definición del tipo de columna de la base de datos.

:default => value

Setea el valor por defecto para la columna. Notemos que el default es calculado. Las 'columnas decimales' toman dos opciones adicionales: **:precision** y **:scale**. Siempre que fuese necesario, deberíamos dar los valores por defecto a nuestras columnas, y cualquier otra información que consideremos importante. Ej : t.column :fname, :string, :limit => 10, :null => false

Por defecto, las columnas serán *null*, las cadenas en SQL serán *varchar(255)*.

Por lo tanto es nuestra responsabilidad **setear la longitud deseada**.

4) Leer <http://wiki.rubyonrails.org/rails/pages/UsingMigrations>.

Ahora que hemos cubierto casi todo AR, centremos nuestra atención en los Controladores y Vistas.