Laboratorio 4 – Asociaciones y Migraciones

## Objetivos

* Continuar practicando Active Record y el uso de la consola en Rails
* Aprender a utilizar las migraciones
* Aprender a utilizar las directivas de asociación de Active Record

## Requerimientos

En este laboratorio deberá implementar un modelo de datos con Active Record de Rails para manejar cursos, alumnos y asignaciones a manera de ejemplo y luego un manejo de clientes, productos y órdenes como ejercicio.

## Procedimiento

Será necesaria una nueva aplicación que puede crear escribiendo en la terminal

rails new lab4

y hacer un nuevo repositorio de git en github (<https://github.com/repositories/new>) luego repetir los pasos que ya conocen del laboratorio anterior para git. En esta práctica no vamos a usar Heroku.

Configuración

Pueden utilizar el manejador de base de datos de su preferencia, por defecto Rails utiliza sqlite y en la práctica anterior vimos la configuración de MySQL y recomendamos que lo sigan utilizando durante el resto de las prácticas, si en caso su preferencia fuera otra, siéntase libre de elegir manejador e investigar la forma de configurar la conexión con Rails.

Migraciones

En esta práctica empezaremos a trabajar con migraciones. En Rails, las migraciones son una forma sencilla y ordenada de llevar control de la estructura de la base de datos. Para la ejecución nos ayudaremos de rake. Si tuviera alguna duda del manejo de línea de comandos (rake, rails generate, rails console, etc) puede visitar <http://guides.rubyonrails.org/command_line.html>

Vamos a crear modelos para cursos y alumnos, además, vamos a crear una migración para las asignaciones, para este caso (muchos alumnos pueden asignarse muchos cursos, relación muchos a muchos) no hay un modelo para la asignación pero si una tabla en la base de datos(y que no tiene llave primaria :-O)

Al generar los modelos, puede no especificarse nada o bien desde allí indicar las características de la migración. Primero vamos a hacer un modelo con migración vacía para estudiantes ejecutando

rails g model student

al ejecutar el comando se creara el archivo del modelo en

app/models/student.rb

y además, un archivo de migración en

db/migrate

el nombre de este archivo incluye al inicio un numero que representa la fecha y hora de la creación y luego \_create\_students, en mi caso, el path completo es

db/migrate/20110210161223\_create\_students.rb

al abrir este archivo, encontrara una clase de Ruby con 2 métodos llamados up y down, estos métodos se utilizan para *subir* o *bajar* en la migración de datos. Dentro de cada método tiene una llamada a la función que crea o borra una tabla, también podría tener aquí una llamada para modificar una tabla. A esta función se le envía el nombre de la tabla y luego se utiliza un bloque para realizar lo que deseamos. En nuestro caso queremos guardar nombre, carnet y estatus del estudiante siendo los 2 primeros texto y el último un valor verdadero o falseo.

Nombre de la tabla: Students

Campos

* name (text)
* student\_id (text)
* status (boolean)

para realizar esto dentro del archivo, debe colocar la definición de los campos de la base de datos dentro del bloque de llamada a create\_table de la forma

t.tipo :nombre

Dentro de este bloque ya se incluye t.timestams, esto es una sugerencia de Rails, al incluirlo en la migración se agregarán columnas con la fecha de creación y actualización de los registros. Al finalizar, su clase debería quedar así:

class CreateStudents < ActiveRecord::Migration

def self.up

create\_table :students do |t|

t.text :name

t.text :student\_id

t.boolean :status

t.timestamps

end

end

def self.down

drop\_table :students

end

end

una vez lista la migración, le indicamos a rails que queremos actualizarnos con un comando de rake, pero primero es necesario crear la base de datos. Recuerde, **debe configurar la conexión previo a ejecutar este comando.**

rake db:create

si tuviera algún problema revise el archivo config/database.yml una vez creada la base de datos no es necesario volver a ejecutar el db:create de rake pero cada vez que quiera que el sistema migre a una nueva versión de estructura de almacenamiento de datos es necesario

rake db:migrate

si no tuvo errores, debería ver en la terminal que la tabla estudiantes fue creada exitosamente.

== CreateStudents: migrating =================================================

-- create\_table(:students)

-> 0.0033s

== CreateStudents: migrated (0.0037s) ========================================

posterior a esto, puede ir a revisar su base de datos, a través de cualquier forma (línea de comandos, phpmyadmin, query browser, etc) y verá la tabla creada.

Para crear el modelo de cursos vamos a indicarle desde la creación del modelo que queremos en la tabla. Por cada curso queremos guardar nombre del curso y un código alfanumérico que lo identifica.

Nombre de la tabla: Courses

Campos

* name (text)
* code (text)

En este caso, el comando sería

rails g model course name:text code:text

y al ver el archivo de la migración debería contener lo siguiente

class CreateCourses < ActiveRecord::Migration

def self.up

create\_table :courses do |t|

t.text :name

t.text :code

t.timestamps

end

end

def self.down

drop\_table :courses

end

end

ahora procedemos a migrar hacía el nuevo modelo de datos con la tabla cursos, ejecute otra vez

rake db:migrate

y deberá recibir el feedback de que la tabla fue creada exitosamente. En el momento en que quiera hacer rollback de la migración puede hacerlo con

rake db:rollback

en este caso, el rollback significa eliminar la tabla de cursos. Si quisiera ir a una migración en específico puede hacerlo indicando la versión al migrar, la versión es el número antepuesto al nombre de la migración, en mi caso la migración para crear cursos es *20110210164613\_create\_courses.rb* por lo que ejecuto el siguiente comando

rake db:migrate VERSION=20110210164613

Si por alguna razón, es necesario cambiar la estructura, es necesario hacer migraciones nuevas, vamos a agregar ahora que tanto el carnet del estudiante como el código del curso sean únicos a través de un índice.

Primero es necesario crear la migración

rails g migration AddIndexConstraints

y dentro de este archivo, agregarle índice a cada campo que queremos que sea único. Es importante notar que al referirse a la tabla se utiliza el plural.

class AddIndexConstraints < ActiveRecord::Migration

def self.up

add\_index :students, :student\_id, :unique => true

add\_index :courses, :code, :unique => true

end

def self.down

remove\_index :students, :student\_id

remove\_index :courses, :code

end

end

al terminar la migración, ejecute de nuevo

rake db:migrate

Por último vamos a agregar una tabla asignaciones para la relación de muchos a muchos. Esta tabla tendrá la característica de referenciar a estudiantes y cursos y no tendrá llave primaria. Empiece creando la migración

rails g migration CreateEnrollments

dentro de esta migración se creará una tabla nueva llamada courses\_students, el estándar es que la tabla se llame como sus las tablas a las que referencia en orden alfabético que no queremos que lleve llave primaria y con referencias a estudiantes y cursos.

class CreateCoursesStudents < ActiveRecord::Migration

def self.up

create\_table :courses\_students, :id => false do |t|

t.references :course, :student

t.timestamps

end

end

def self.down

drop\_table :courses\_students

end

end

Alternativamente a t.references :students podría utilizar t.integer student\_id al terminar la migración, ejecute de nuevo

rake db:migrate

puede ampliar información sobre las migraciones en <http://guides.rubyonrails.org/migrations.html>

Asociaciones

Asociar modelos es una operación común y Rails busca facilitar las cosas a través de asociaciones en código para el acceso entre objetos que están relacionados. Cada asociación es una conexión entre dos modelos, la forma de hacerlo es colocar en el código de los modelos involucrados de forma declarativa, como un macro, la forma en que se asociación.

Por ejemplo, si yo tuviera 2 modelos: Niño y Mascota, puedo decir que están asociados de la forma: “El niño tiene algunas mascotas” y “La mascota pertenece al niño”, si lo viéramos desde la perspectiva de la base de datos entonces en la tabla niños habría un id y en la tabla mascotas habría un niño\_id con una llave foránea hacia la tabla niños.

En Rails manejamos 4 asociaciones básicas que son:

* belongs\_to
* has\_one (y has\_one :through)
* has\_many (y has\_many :through)
* has\_and\_belongs\_to\_many

En el laboratorio estamos trabajando asignaciones y específicamente, queremos que haya una relación de muchos a muchos entre estudiantes y cursos a través de asignaciones. Para esto, vamos a editar los archivos de modelo y en cada una agregamos un has\_and\_belongs\_to\_many hacia la otra clase, por ejemplo, para los estudiantes debe editar

app/models/student.rb

y dentro del modelo agregar la asociación para que quede

class Student < ActiveRecord::Base

has\_and\_belongs\_to\_many :courses

end

Repita el procedimiento para los cursos.

Pruebas

Al crear las asociaciones, Active Record reconoce las propiedades asociadas, en esta caso, al tener una asociación de muchos a muchos tendremos en cada objeto un arreglo de instancias del otro objeto. Si fuera un belongs\_to/has\_many tendríamos un dato miembro que referencia la asociación.

Ingrese a la consola de rails y haga un estudiante y un curso nuevo como se vio en la práctica anterior, luego asócielos a través de un push al arreglo courses dentro del estudiante o al arreglo students dentro del curso.

Por ejemplo, si tiene un estudiante S y un curso C, en código sería

s.courses << c

o bien

c.students << s

pero solamente una de las 2 es necesaria y si ejecuta las 2 se agregará el objeto 2 veces y no queremos eso. Al no tener una llave primaria, la integridad debe ser llevada por el código de la aplicación.

Luego, para revisar si se agregó puede imprimir el arreglo de la asociación, por ejemplo

s.courses

le permitirá ver los cursos que tiene el estudiante asignado. Puede ampliar información sobre las asociaciones en <http://guides.rubyonrails.org/association_basics.html>

## Ejercicio #1

Debe implementar una clase de ActiveRecord que le permita manejar clientes y productos mediante acciones CRUD. Para ello, creará el esquema de la base de datos utilizando la herramienta migrations de Active Record por cada modelo que cree.

El esquema de su tabla clientes debe tener lo siguiente:

* Nombre de la tabla: clients
* id: Identificador único de cada cliente (llave primaria).
* name: Nombre del cliente.
* address: Dirección del cliente en texto

El esquema de su tabla productos debe tener lo siguiente:

* Nombre de la tabla: products
* id: Identificador único de cada producto (llave primaria).
* code: Código que identifica al producto dentro del almacén. Debe ser alfanumérico y único.
* price: Precio del producto(flotante).
* description: Descripción del producto.

El esquema de las órdenes de compra debe tener lo siguiente:

* Nombre de la tabla: orders
* id: Identificador unico de cada orden (llave primaria).
* code: Codigo que identifica a la orden dentro del almacen. Debe ser unico. Por ejemplo: abcd-1 ó 001.
* date: Fecha en que se realizo la orden de compra.
* client\_id: Referencia al cliente que realizo la orden de compra.
* amount: Cantidad ordenada de producto.

Además, para este ejemplo, relacionaremos a los productos con las ordenes por lo que deberá modificar la tabla de productos y agregar un campo order\_id para tener referencia a la orden. Como el nombre del cliente no nos alcanza para identificarlo unívocamente, agregaremos un campo NIT.

* nit: NIT del cliente. Debe ser único

Todo esta debe ser realizado utilizando migraciones.

Luego, debe implementar un método en el modelo que le permita ingresar ordenes de compra, para este ejercicio puede asumir que cada orden de compra se hace por un único producto. Mas adelante se le pedira que soporten varios productos.

* boolean createOrder(clientNit, productCode)

Este método crea una orden de compra para el nit del cliente recibido y el código del producto recibido. Debe devolver verdadero si la orden se ingresó exitosamente o falso si ocurrió algun error, por ejemplo, si el nombre del cliente no está registrado en la tabla de clientes o si el producto no existe.

## Ejercicio #2

Ahora modificará su aplicación para que la orden tenga multiples productos, recuerde cambiar las asociaciones antes establecidas. Dado este cambio, es necesario modificar su método *createOrder* y ahora recibirá un listado de productos (podrá utilizar sobrecarga de métodos?).

## Ejercicio #3

Por último, agregará una tabla para almacenar proveedores, estos proveedores venden muchos productos a la tienda y cada producto puede venir de varios proveedores.

* Nombre de la tabla: suppliers
* id: Identificador unico de cada orden (llave primaria).
* name: Nombre del proveedor
* address: Dirección del cliente en texto

## Entrega

Enviar un archivo.zip que contenga el folder de la aplicación y un archivo de texto con el url de su repositorio en Github