



Interactuando con los datos

Rodrigo Rodriguez
rorodr@gmail.com
Skype: rarodriguezr

Active Record



- ORB (object-relational Mapping): Una técnica para convertir fácilmente tablas en objetos compatibles con lenguajes orientados a objetos.
- Permite obtener, crear, actualizar y listar objetos desde/hacia la Base de Datos

Migraciones de BD



- Es una clase de Ruby que se encarga de efectuar cambios en la Base de Datos
- Cada archivo de migración tiene un consecutivo (datestamp), el cual luego de ejecutada, se guarda en la tabla schema_migration
- Las migraciones se pueden crear desde 0 o utilizar el generador para crear una:
 - rails generate migration NAME

Migraciones de BD



- Cada migración debe de ser ejecutada en todas las BD relacionadas al proyecto
- Existen 2 formas distintas de declarar una migración, con “self_up”/“self_down” o con “Change”
- Para ejecutar las migraciones es necesario ejecutar
 - rake db:migrate

Migraciones de BD



- Cada migración debe de ser ejecutada en todas las BD relacionadas al proyecto
- Existen 2 formas distintas de declarar una migración, con “self_up”/“self_down” o con “Change”
- Para ejecutar las migraciones es necesario ejecutar
 - rake db:migrate

Tipos de datos en Migraciones



- Las migraciones de datos permiten generar columnas de diversos tipos de datos, entre los que se citan:

:binary

:boolean

:date

:datetime

:decimal

:float

:primary_key

:integer

:string

:text

:time

:timestamp

Tipos de datos en Migraciones



SQL Type

int, integer

float, double

decimal, numeric

char, varchar, string

interval, date

datetime, time

clob, blob, text

boolean

Ruby Class

Fixnum

Float

BigDecimal

String

Date

Time

String

See text

Métodos en Migraciones



- Algunos de los métodos válidos para la ejecución de cambios en la BD son:

`add_column`

`add_index`

`add.timestamps`

`create_table`

`remove_timestamps`

`rename_column`

`rename_index`

`rename_table`

- En el caso de utilizar el método up/down, se puede agregar otros métodos específicos como “`execute`”, `remove_index`, entre otros

Ejemplo de migración



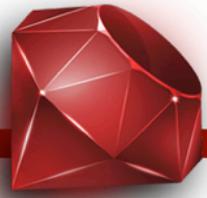
```
class CreateContacts < ActiveRecord::Migration
  def change
    create_table :contacts do |t|
      t.string :name
      t.string :email
      t.string :phone
      t.text :address
      t.integer :store_id
      t.timestamps null: false
    end
  end
end
```

Ejercicio #1



- Descargar y configurar el proyecto del curso
del URL:
[https://github.com/rarodriguez/
media store cr](https://github.com/rarodriguez/media_store_cr)
- Crear una migración en el proyecto que
permite crear la información de los productos
pertenecientes a las tiendas

Uso de tablas y Columnas



- Al igual que en cualquier otra sección de Rails, se tiene una convención en los nombres de las tablas/clases.

Clase

Contact

Store

Nombre de la tabla

contacts

stores

- En caso de no cumplir el estándar, se utiliza:

```
class Sheep < ActiveRecord::Base  
  self.table_name = "sheep"
```

```
end
```

Columnas proveídas por AR



- Active Record utiliza como atributos del objeto las columnas de la tabla, sin embargo hay algunas adicionales que AR agrega.
 - user.superuser?: AR agrega un método “ATTR?”
 - created_at, created_on, updated_at, updated_on: Actualiza los tiempos automáticamente
 - yyy_id: Nombre por defecto de una referencia de una tabla con el nombre de la forma plural de “yyy”
 - yyy_count: mantiene un conteo de la tabla hija yyy

Identificando registros



- En AR, el método “Store.find(1)”, va a devolver la tienda cuyo identificador primario sea 1.
- Por defecto, los identificadores primarios son numéricos con el nombre “id”

```
class LegacyBook < ActiveRecord::Base  
  self.primary_key = "isbn"  
end
```

- Rails considera 2 objetos como iguales (usando ==) si son instancias de la misma clase y tienen la misma llave primaria.

Especificando relaciones

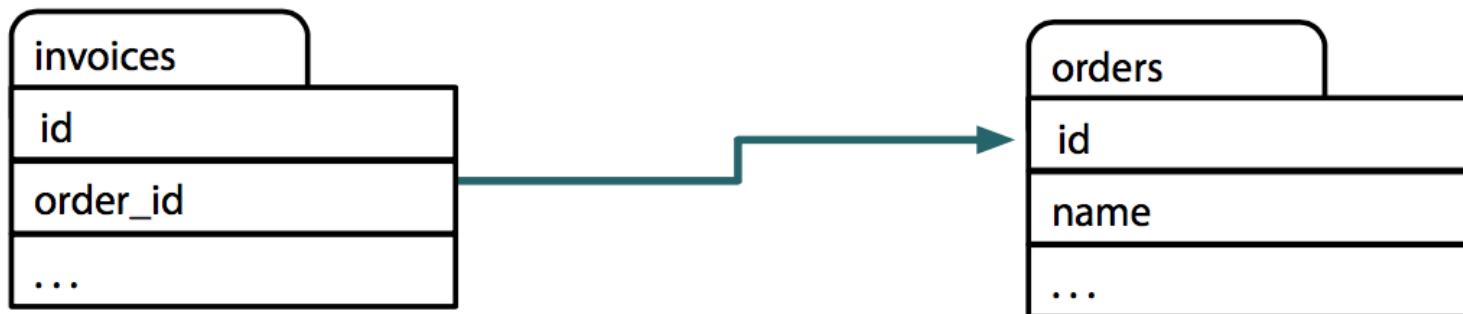


- Active Record soporta 3 tipos de relaciones entre las tablas, 1 a 1, 1 a muchos y muchos a muchos.
- Para declarar esas relaciones, se utilizan: `has_one`, `has_many`, `belongs_to` y `has_and_belongs_to_many`
- Por lo general se definen relaciones en ambos lados de la relación, con el fin de facilitar el proceso de búsqueda de información.

Relaciones 1 a 1



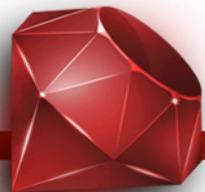
- Es implementado utilizando una llave foránea en un registro de una tabla, la cual referencia un único registro de otra tabla.



```
class Invoice < ActiveRecord::Base  
  belongs_to :order  
  # ...  
end
```

```
class Order < ActiveRecord::Base  
  has_one :invoice  
  # ...  
end
```

Relaciones 1 a N



- Permite representar una colección de objetos. En Active Record, la clase padre utiliza “has_many” para declarar su relación con la tabla hija. La tabla hija, utiliza “belongs_to” para indicar su parent.



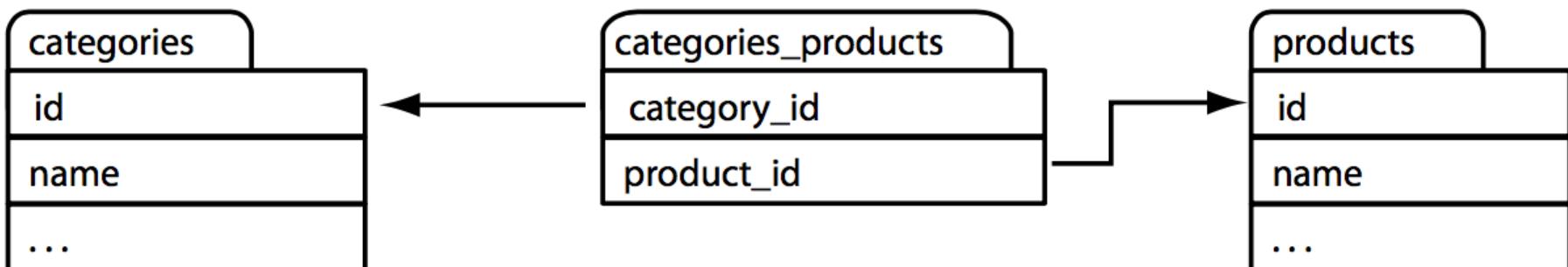
```
class LineItem < ActiveRecord::Base
  belongs_to :order
  # ...
end
```

```
class Order < ActiveRecord::Base
  has_many :line_items
  # ...
end
```

Relaciones N a M



- Existen casos donde ocupamos relacionar muchos elementos de una tabla con muchos elementos de otra (ej. Productos y tiendas)
- Son asociaciones simétricas, ambas de sus objetos declaran la relación



```
class Category< ActiveRecord::Base  
  has_and_belongs_to_many :products  
  # ...  
end
```

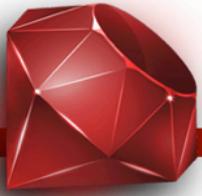
```
class Product< ActiveRecord::Base  
  has_and_belongs_to_many :categories  
  # ...  
end
```

Ejercicio #2



- Crear las migraciones, los modelos y las relaciones requeridas para efectuar los módulos principales de la aplicación

Creando nuevos registros



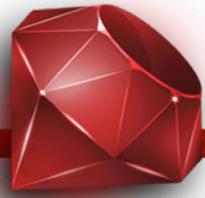
- Al considerar que cada objeto de una clase representa un registro en la BD, crear un nuevo registro requiere únicamente la creación de una instancia de la clase que queremos crear y luego salvarla. Ej.:

```
s = Store.new  
s.name = "RRR Store"  
s.url = "http://misitioweb.com"  
s.save
```

```
s = Store.create(name: "RRR Store",  
url: "http://misitioweb.com")
```

```
Store.new do |s|  
  s.name = "RRR Store"  
  s.url = "http://misitioweb.com"  
  s.save  
end
```

Obteniendo registros (1/5)



- La forma más fácil de buscar elementos es utilizando el método “find”, el cual busca los elementos de acuerdo con su llave primaria:
 - Store.find(ID)
- También es posible leer todos los registros de una clase específica utilizando “all”:
 - Store.all

IMPORTANTE: Siempre utilizar el método `to_i`, cuando se hace un `find` que utiliza parámetros desde el cliente

Obteniendo registros (2/5)



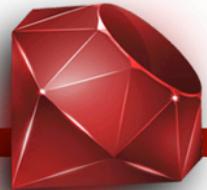
- Se pueden hacer búsquedas más complejas utilizando “where”, lo que nos permite hacer consultas ya sea utilizando objetos o por medio de SQL. NOTA: por defecto los where son unidos por medio de ANDs

```
Contact.where(email: 'email@domain.com', store_id: 5)
```

```
Contact.where("email = ? AND store_id = ?",  
'email@domain.com', 5)
```

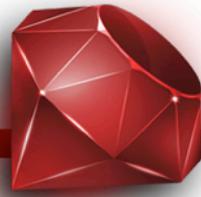
```
Contact.where("email = :email AND store_id  
= :store_id", email:'email@domain.com', store_id:5)
```

Obteniendo registros (3/5)



- Algunas veces hay que usar condiciones que únicamente conocemos una parte de la palabra (por ejemplo, una búsqueda). Para ello, en SQL se utiliza “like”, en Rails se utiliza por medio de:
- `Contact.where("email like ?", 'email%')`

Obteniendo registros (4/5)



- Además de WHERE, a veces es necesario efectuar otras acciones a nuestra colección de datos, algunos de esos métodos son:
- Order: Permite ordenar los resultados
 - Contact.where(name: ‘juan’).order(“name DESC”)
- Limit: Permite limitar los resultados
- Offset: Permite establecer la primera línea que se retorna. Por lo general se utiliza con limit.
 - Contact.limit(10).offset(5)

Obteniendo registros (5/5)



- Joins/includes: Permiten indicar en el SQL que se desea incluir objetos relacionados con el fin de obtener los datos filtrados.
 - `Contact.joins(:store).where("stores.name = ?","RRR store")`
- Average/maximum/minimum/sum/count: permiten realizar cálculos sobre los registros numéricos. Por defecto únicamente retornan un único registro con la información.

Ejercicio #3



- Con el uso de la consola de Rails:
 - “rails console”
- Escribir la consulta para obtener todos los Contactos de una tienda

Scopes



- Algunas veces se tienen consultas genéricas que queremos reutilizar o simplemente acomodar mejor las consultas SQL, para ello utilizamos los scopes (en los modelos).

```
class Order < ActiveRecord::Base
  scope :last_n_days, lambda { |days|
    where('updated < ? ', days) }
  scope :checks, -> { where(pay_type: :check) }
end
```

- orders = Order.checks.last_n_days(7)

Reload



- En ciertas ocasiones existe la posibilidad de que al menos uno de los elementos que cargamos haya sido cambiado durante la ejecución, por lo que ocupamos recargarlo, para ello se utiliza el método “reload”.

```
s = Store.find(5)  
sleep(60)  
s.reload #Reload the element
```

Actualizando registros



- Existen múltiples formas de actualizar un registro:
 - Cargando el objeto desde la base de datos, modificar sus atributos y salvar
 - Utilizando el método update:
 - `Store.first.update(name: "RRR new store")`
 - Utilizando otra versión del Update
 - `Store.update(1, name: "RRR new store")`
 - Utilizando update_all:
 - `Store.update_all("name = name + NEW", "name is not null")`

Eliminando registros



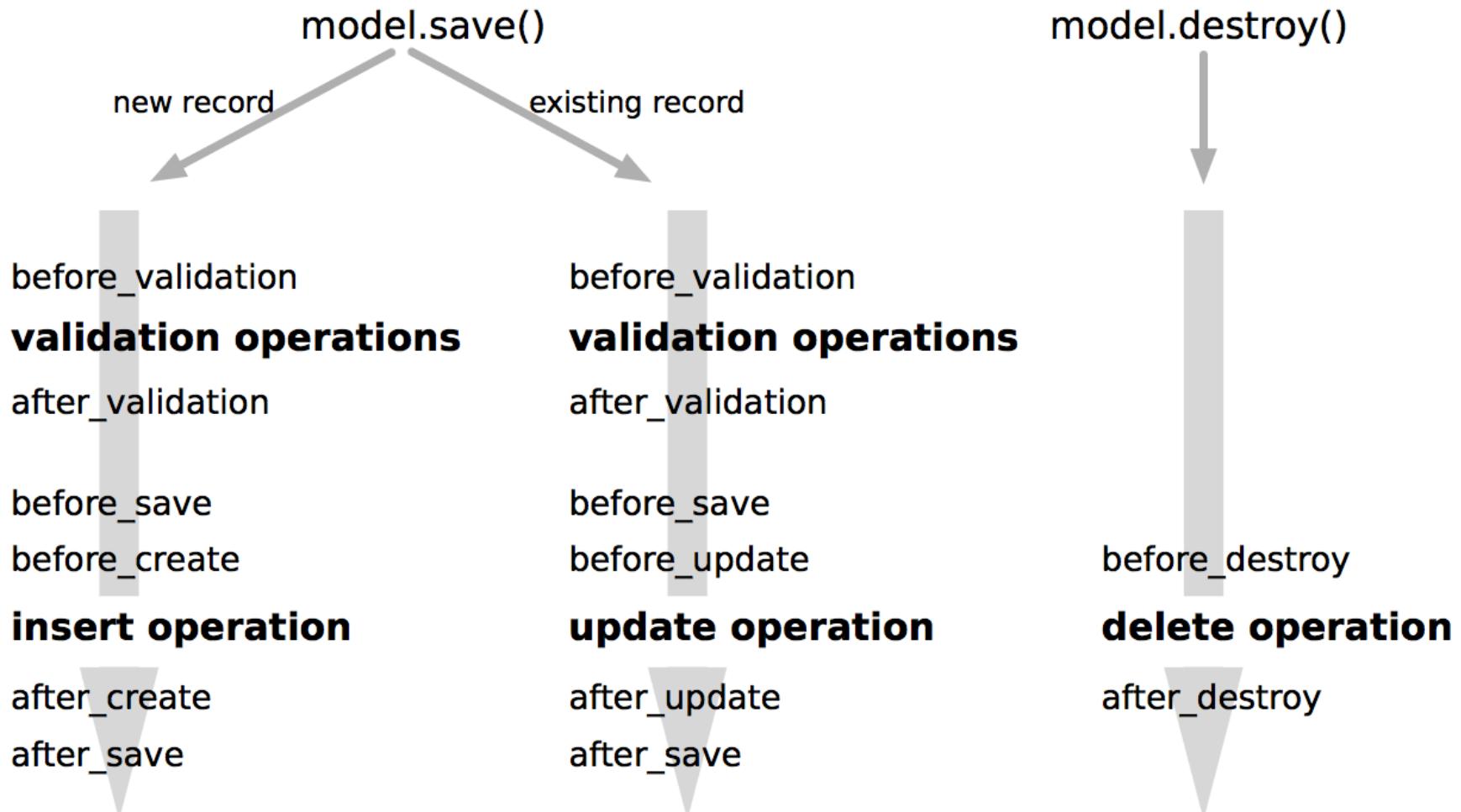
- Existen múltiples formas de eliminar un registro:
 - `delete/delete_all`: Elimina el registro directamente desde la base de datos
 - `Store.delete(1)`
 - `Store.delete_all(['id > ?', 10])`
 - `destroy/destroy_all`: Elimina los elementos respetando algunas validaciones y “callbacks” propios del modelo asociado
 - `Store.destroy(1)`
 - `Store.destroy_all(['id > ?', 10])`

Callbacks



- AR, provee una forma para controlar el ciclo de vida de los objetos de un modelo, para ello utiliza los callbacks.
- En total, existen 20 callbacks que son invocados ya sea antes, durante o después de hacer una acción específica.
 - Ej.: `before_destroy` se ejecuta antes de eliminar el objeto.
- Si uno de los callbacks retorna “false” o eleva una excepción, las acciones siguientes no se ejecutan

Callbacks



Callbacks



```
class Order < ActiveRecord::Base
  before_validation :normalize_credit_card_number
  after_create do |order|
    logger.info "Order #{order.id} created"
  end
  protected
  def normalize_credit_card_number
    self.cc_number.gsub!(/[-\s]/, '')
  end
end
```

Transacciones



- A veces es necesario realizar acciones que se deben efectuar en conjunto (por ejemplo, transacciones bancarias), las cuales requieran que se ejecuten ambas acciones, de lo contrario revertir los cambios, eso se efectúa con transacciones.

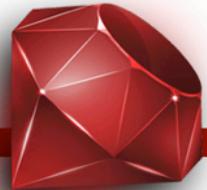
```
Account.transaction do
    account1.deposit(100)
    account2.withdraw(100)
end
```

Transacciones



- Transacciones revierten los cambios cuando se elevan excepciones de cualquier tipo.
- **Recomendación:** En transacciones, tratar de elevar una excepción de tipo:
ActiveRecord::Rollback, la cual únicamente revierte el cambio de BD, sin necesidad de rescatar la aplicación de una excepción

Validando registros (1/4)



- Rails provee validaciones nativas a los modelos, de modo que antes de crear o actualizar un elemento, se verifique que cumpla con lo declarado.

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

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

Validando registros (2/4)



- Existen en cada modelo 2 métodos q son de importancia para verificar si un registro es válido o no:
 - valid?: Ejecuta la validación del objeto. Ej: Contact.create(name: nil).valid?
 - errors: variable que almacena los errores de validación encontrados. Ej:
 - Contact.create.errors.messages
 - Contact.create.errors[:name]
 - Contact.create.errors[:name].any?

Validando registros (3/4)



- Algunos de los helpers de validación:
 - acceptance: se haya marcado un checkbox
 - validates_associated: un elemento asociado
 - confirmation: confirmación de un elemento
 - format: valida el formato (REGEX)
 - inclusion: forme parte de una lista
 - length: tamaño de un string
 - numericality: solo números
 - presence: valida que exista
 - uniqueness: valida que solo haya un elemento

Validando registros (3/4)



```
class Product < ActiveRecord::Base
  validates :title, :description,
            :image_url, :presence => true
  validates :price, :numericality =>
{ :greater_than_or_equal_to => 0.01}
  validates :title, :uniqueness => true
  validates :image_url, :format => { :with =>
%r{\.(gif|jpg|png)$}i, :message => 'must be a
URL for GIF, JPG or PNG image.' }
end
```

Ejercicio #4



- Agregar a los modelos del sistema las validaciones básicas requeridas (formato para email, campos requeridos, etc).

Recursos adicionales



- *Libro: Agile Web Development with Rails 4*