Ruby on Rails

Contenido

1. Introducción a Ruby
    1. Instalación de Ruby on Rails
    2. Clases básicas
    3. Bloques e iteraciones
    4. Expresiones, Funciones y Rangos
    5. Clases
2. Comenzando con Rails
    1. Modelo Vista Controlador
    2. Comenzando con Rails (Creación de Carrito)
    3. Usando el comando scaffold
    4. Migraciones
    5. Modelo
    6. Controlador
    7. Vistas
3. Bases de Datos, plugins y layouts
    1. Base de Datos de la aplicación Carrito
    2. Plugins (Paperclip)
    3. Layouts
4. Sesiones, Relaciones y Active Record
    1. Usando sesiones
    2. Relaciones
    3. La consola
    4. ActiveRecord

1 Introducción a Ruby

Ruby es un lenguaje de programación dinámico con gramática expresiva y compleja y clases base con una API muy completa y poderosa creado por Yukihiro Matsumoto. Ruby obtuvo su inspiración de programas como Lisp, Smalltalk y Perl, Ruby es un lenguaje orientado a objetos pero en el cual se puede realizar programación procedimental. Ruby una de sus principales características es la metaprogramación.

1.1 *Instalación de Ruby y Rails*

Para poder instalar Ruby es necesario dirigirse a la siguiente dirección: http://rubyonrails.com/down en esta dirección existe los vínculos para descargar así como para poder instalar Ruby y Rails en distintos ambientes en este curso no tocaremos el tema de la instalación de Rails debido a que llevaría mucho tiempo explicar como se instala Ruby on Rails en las distintas plataformas y ya existe una gran cantidad de tutores que pueden ayudar a esto no mostraremos como hacerlo, pero en caso de que no deseen complicarse con la instalación de Ruby on Rails, pueden dirigirse a la dirección: http://bitnami.org/stacks en donde podrán encontrar instaladores que instalan de forma automática Ruby on Rails para cualquier plataforma, solo deben ver la lista **Infraestructure,** y seleccionar ***RubyStack*** que tiene instaladores de Apache, PHP, MySQL, Ruby, phpMyAdmin.

1.2 *Clases básicas*

Las clases básicas de Ruby son:
1.class # => Fixnum: Clase para números decimales
1.1.class # => Float: Clase para manejar número decimales
true.class # => TrueClass: Clase verdadera, singleton de la clase TrueClass
nil.class # => NilClass : Clase nula
"hola".class # => String: Clase que maneja caracteres
[1,2,3,'si'].class # => Array:
:simbolo.class # => Symbol: Clase especial mas reducida que strings generalmente usada para poner indices en la clase Hash
{:ind1 => 'uno', 'ind2' => 1}.class # => Hash: Clase que almacena datos con indices únicos

Para poder iniciar el modo consola de Ruby solo debemos escribir en la consola *irb* y esto nos mostrara un entorno en el cual podemos realizar pruebas con Ruby. Comencemos haciendo algunas pruebas, ejecutamos los siguientes comandos:

```
irb(main):002:0> "esto es un mensaje".length
=> 18
irb(main):003:0> "esto es un mensaje".upcase
=> "ESTO ES UN MENSAJE"
irb(main):003:0> a = "esto es un mensaje"
=>"esto es un mensaje"
irb(main):003:0> a.upcase
=> "ESTO ES UN MENSAJE"
```

Código 1.1

Como pueden ver en este caso hemos creado un String "esto es un mensaje", del cual podemos ver su longitud de caracteres accediendo a la propiedad *length*, o podemos hacer que todas las letras sean mayúsculas con el procedimiento *upcase*. En Ruby se puede llamar a las funciones con paréntesis o sin paréntesis todo depende del contexto en el cual se encuentre la llamada, en este caso podemos hacer que todas las letras se vuelvan mayúsculas llamando *a.upcase()*.

1.3 *Bloques e iteraciones*

En Ruby existen muchos procedimientos para los *Arrays y Hashes* que permiten iterar a raves de los mismos

```
# En caso de que iteremos
[1,2,3].each{|v| print v} # Nos imprime 123
# Es posible realizar los mismo en bloque
[1,2,3].each do |v|
  print a
end
# En caso de que queramos realizar una operación podemos usar map
[1,2,3].map{|v| v*v} # => [1, 4, 9]
#En caso de que queramos modificar el una variable array hacemos lo siguiente
a = [1,2,3]
a.map!{|v| v*v} # => [1, 4, 9]
#Ahora vamos a calcular la suma de todos los elementos
a.inject { |sum, v| sum+v} # 14
```

Código 1.2

Como podemos ver es el código es muy sencillo y simple, como generalmente dicen los que programan en Ruby “elegante”, Ruby permite realizar las aplicaciones con una menor cantidad de líneas de código que lenguajes como JAVA, PHP o C#, al tener menos líneas de código uno tienes menos que revisar y es menor la probabilidad de cometer errores. Los Hashes son muy similares a los arrays solo que manejan indices y utilizan llaves en ves de corchetes.

```
# Definición de un Hash
a= {:uno => 1, :dos => 2, :texto => 'texto', 'indice' => 'Indice tipo Texto'}
# Iterando a traves del Hash
a.each {|k,v| print "a[#{k}] = #{v}; "} # Nos devuelve => a; [dos] = 2; a[texto] = texto; a[indice] = Indice tipo
#Texto;
```

Código 1.3

En caso de que estemos trabajando con Hashes podemos utilizar como indices muchos tipos de clases como Fixnum, Float, Symbol, String, etc. Es también posible crear Arrays con Hashes que verán es muy común cuando se realizan consultas a Rails recibir este tipo de datos como respuesta de una consulta. También es muy fácil acceder y leer archivos

```
# Como leer un archivo
f = File.open 'archivo.txt'
f.each{|l| print l}
# Iterando a traves del Hash
```

Código 1.4

1.4 *Expresiones, Funciones y Rangos*

En general todas las sentencias en Ruby son expresiones que retornan valores, las expresiones en Ruby son muy similares a las de otros lenguajes

```
# Sentencia if
if a > 3
  print 'Mayor que 3'
elsif a == 3
  print “Igual a 3”
else
  print 'Menor que 3'
end
# Rango
r = 'a'..'z'
r.each{|v|  print v} # => abcdefghijklmnopqrstuvwxyz
```

Código 1.5

Para poder crear funciones, como lo hacemos en este caso con la función recursiva

```
# Creación de una función
def factorial v
  if v == 0
    1 # tambien podemos usar return 1
  else
    v * factorial(v-1) # aquí también podemos usar return v * factorial(v-1)
  end
end
factorial 4 # => 24
```

Código 1.6

1.5 *Clases*

Las clases en Ruby funcionan de forma similar a las clases de otros lenguajes como JAVA o PHP, permiten tener métodos privados o protegidos y pueden realizar herencia simple

```
# Creación de una función
class Uno
  # Initialize es como el método constructor de la clase
  def initialize v
    @val = v
  end
  # Similar a getter
  def val
    @val
  end
 # Similar a setter
 def val=(v)
    @val = v
  end
# Todos los métodos que estén después de esta declaración son protegidos
  protected
  def protegida
    print 'Método protegido'
  end
# Todos los métodos que estén después de esta declaración son privados
  private
  def privada
    print 'Método privado'
  end
end

#Herencia
class Dos < Uno
  def to_s
    print 'Esta es la clase 2'
  end
end

c = Dos.new 'Nuevo valor' # Creación de una instancia
c.val # => Nuevo valor
c.val = 'Prueba' # => En el objeto @val = 'Prueba'
print c # => 'Esta es la clase 2'
```

Código 1.7

2 Comenzando Rails (Creación de Carrito)

Rails fue desarrollado por David Heienemeier Hansson para la compañía 37 Signals durante el desarrollo de la aplicación *BaseCamp*, Rails aplica varios patrones para poder hacer la vida mas fácil a los desarrolladores, la primera de las cuales es convención sobre configuración, un ser humano siempre estar mas acostumbrado a las conveciones, por ejemplo estamos acostumbrados que el saludo en la mañana sea “buenos días” y “buenas tardes” en la tarde, otra de los principios aplicados a Rails es el principio DRY (Don't Repeat Yourself) no te repitas, hay muchas tareas que resultan repetitivas en el desarrollo las cuales ya han sido implementadas en Rails y la siguiente es el Modelo Vista Controlador (MVC)

2.1 *Modelo Vista Controlador (MVC)*

Es un patrón de arquitectura de software que separa los datos de una aplicación, en caso de las aplicaciones el Modelo resultaría todas las consultas a la Base de Datos, la Vista la presentación de los datos en HTML y CSS y el Controlador con la lógica que comunica al Modelo y a la Vista, este patrón permite un desarrollo mas coherente y mas ordenado del código, fue inicialmente descrito en 1979 por Trygve Reenskaug

Modelo

Es la representación específica de la información, es la parte con la el sistema opera con los datos y en la cual se realiza todas las consultas y modificaciones a las Base de Datos

Vista

La vista es la parte de presentación de la información, en el caso de las aplicaciones Web es donde la esta el código HTML, y los formularios.

Controlador

Es la parte encargada de realizar la comunicación entre el Modelo y la Vista, el controlador prepara los datos para que puedan ser presentados en el la Vista, además que contiene algo de la lógica del programa.

2.2 *Comenzando con Rails (Creación de Carrito)*

El proyecto de Rails que realizaremos es un pequeño carrito de compras, ***Carrito*** es el nombre de nuestra aplicación, Para poder comenzar con un nuevo proyecto primero definan cual sera su carpeta de trabajo la cual desde ahora será la ruta que usaremos de referencia para todo el tutor, una ruta ejemplo puede ser */home/<nombre>/rails*, y a la cual la llamaremos ruta de trabajo, una ves en la carpeta que hayan seleccionado hagan correr el siguiente comando *rails mi*-proyecto, una vez creado el proyecto ahora es necesario configurar la Base de Datos en caso de que usen SQLite no es necesario configurar nada, pero en caso de que usen MySQL deben ir a su carpeta de trabajo y editar *config/database.yml* con los siguientes datos

```
#Configuración del archivo de conexión a la Base de Datos
development:
  adapter: mysql
  encoding: utf8
  database: enano
  username: demo
  password: demo
  socket: /var/lib/mysql/mysql.sock
```

Código 2.1

Deben configurar correctamente la dirección de su socket en caso de que no les funciones prueben borrando la línea del *socket*.

Estructura de un proyecto Rails

```
|-- app
|-- config
|-- db
```

```
|-- doc
|-- lib
|-- log
|-- public
|-- script
|-- test
|-- tmp
|-- vendor
```

app es el directorio de la aplicación donde se almacena los modelos, controladores y la vistas, *config* es la carpeta donde se realiza las configuraciones como la base de datos y variables de entorno, *db* es donde se almacena la base de datos en caso de que se use SQLite o las migraciones para otras base de datos, *doc* carpeta de documentos, *lib* carpeta donde se pueden crear librerías o clases, *log*, *public* es donde residen los archivos CSS, imágenes, y javascript, en general es donde se guarda todo el contenido público, *script* aquí se almacena los scripts para generación de código y otros, *test* lugar donde se almacenan los scripts de testeo (pruebas), *tmp* temporal, *vendor* es donde se almacenan los plugins.

2.3 *Usando el comando scaffold*

La aplicación que realizaremos será un carrito de compras de a cual pueden econtrar la base de datos en la Ilustración *3*, comenzaremos con usando el scaffold una de las funciones mas rápidas para poder desarrollar aplicaciones, ingresen a su directorio de trabajo y escriban:

```
#Scaffold para crear un usuario
ruby script/generate scaffold usuario nombre:string apellido:string login:string pass:string
```

Código 2.2

Este comando nos genera el Modelo, la Vista y el Controlador, ahora corran el comando:

```
rake db:migrate
```

Este comando hace que los archivos de migración que se encuentran en *db/migrations* generen las tablas en la base de datos, ahora deben iniciar el servidor web: **ruby script/server**
Ingresen ahora a *http://localhost:3000*

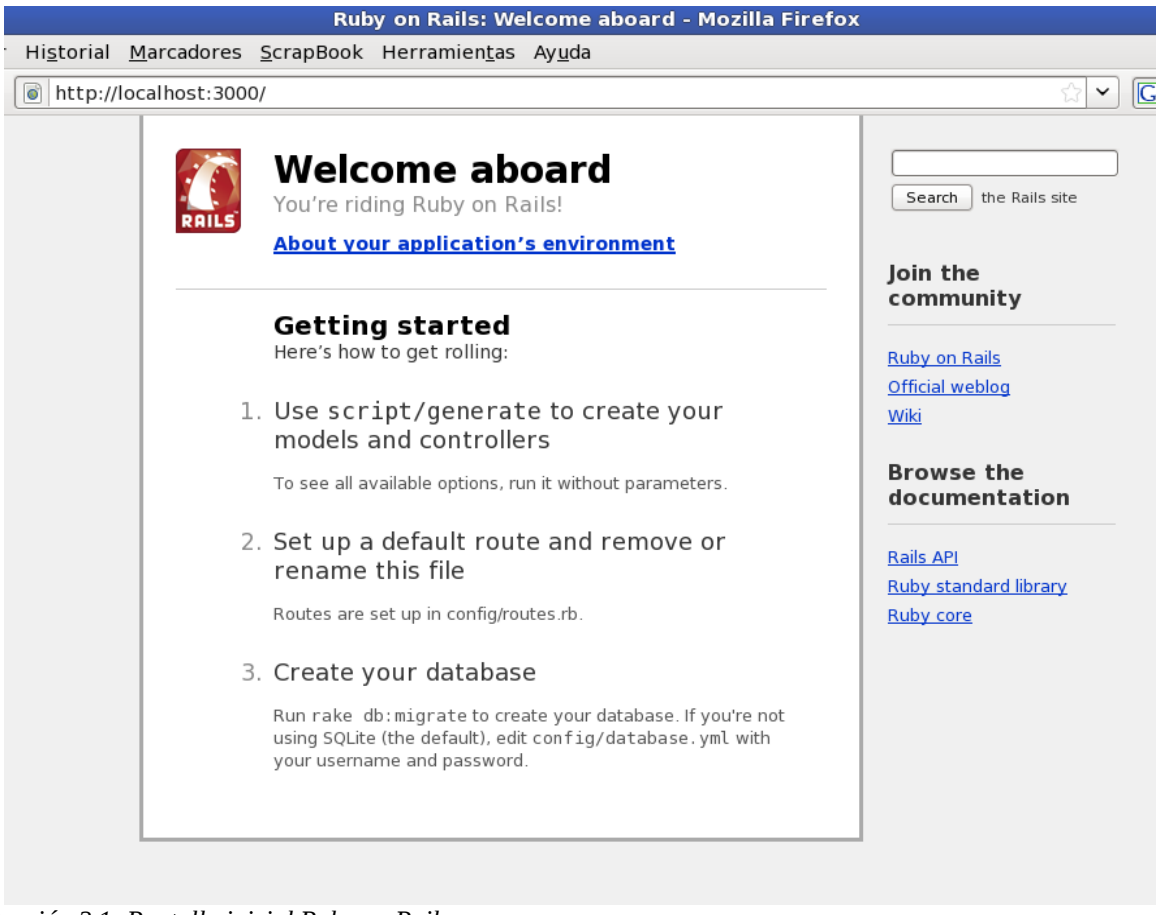


Ilustración 2.1: Pantalla inicial Ruby on Rails

Esta es la pantalla inicial de Rails que presenta un mensaje de saludo ademas si se hace click en *About your application's envroment*, presenta detalles de la instalación de Rails, ahora vayan a la dirección *http://localhost:3000/usuarios*, lo cual les presentara la pantalla de listado de usuarios y a la cual podrán ver, adicionar editar y borrar, este tipo de operaciones es conocida como ABM (altas, bajas, modificaciones) en español y CRUD en ingles (create, read, update, delete)

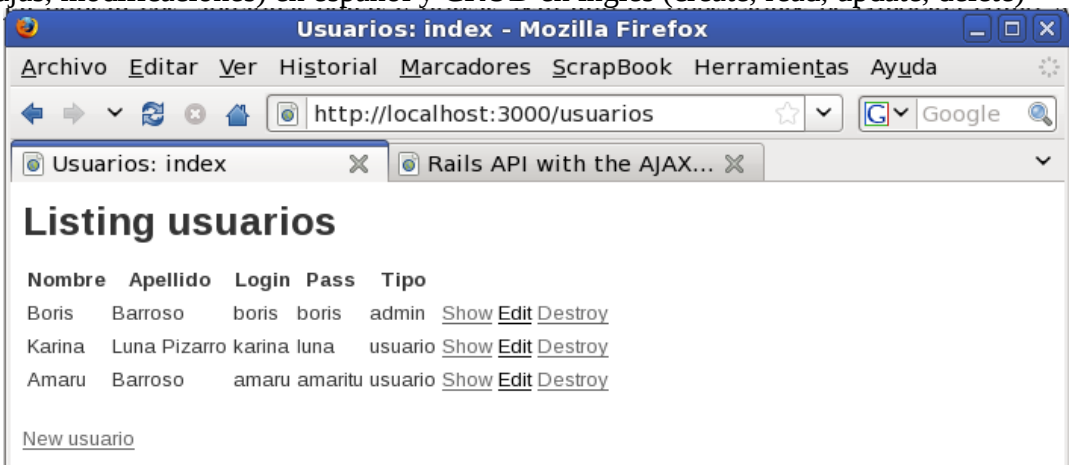


Ilustración 2.2: Listado de usuarios

Ahora explicaremos que es lo que sucedió cuando hicimos el scaffold en el *código 8*, para ello primer veremos la parte de la migración; vayan al directorio *db/migrate*, y encontrarán un archivo que tiene la fecha y hora similar a (*20081105031343_create_usuarios.rb*), abran este archivo.

2.4 *Migraciones*

El archivo de migración contiene la estructura de la tabla de la Base de Datos, cuando ejecutamos el comando *rake db:migrate* hacemos que todas las migraciones se ejecuten para modificar la estructura de la Base de Datos, en caso de que hayan sido ejecutas no borraran sus tablas, esta migración puede ser modificada antes de ejecutar el comando *rake db:migrate* y realizar modificaciones al la tabla, al tener la creación y modificación de nuestra Base de Datos basada en estas migraciones logramos de que la aplicación sea independiente del motor de Base de Datos que se vaya a usar, además que nos sirve de mecanismo de documentación para saber los cambios que se realizaron a la estructura

```
#Migración creada con el scaffold: db/migrate/20081105031343_create_usuarios.rb
class CreateUsuarios < ActiveRecord::Migration # Definición de la clase
  # aqui se inicia con la creación de los campos de la tabla
  def self.up
    create_table :usuarios do |t|
      t.string :nombre
      t.string :apellido
      t.string :login
      t.string :pass
      t.string :tipo # Tipo de usuario cliente o admin
      # Este comando nos crea 2 campos created_at y updated_at que Rails manejado automáticamente por
      # Rails  para guardar la fecha de creación y modificación de un registro
      t.timestamps
    end
  end
  # Sirve para poder eliminar la tabla
  def self.down
    drop_table :usuarios
  end
end
```

Código 2.3

2.5 *Modelo*

Como pueden ver ahora el modelo es muy simple pero haremos modificaciones para poder realizar validaciones y otros procesos, en realidad este es el lugar donde la mayor parte de la lógica de

negocios debe ser implementada.

```
#Modelo: app/models/usuario.rb
class Usuario < ActiveRecord::Base
end
```
Código 2.4

2.6 *Controlador*

```
# Controlador: app/controllers/usuarios_controller.rb
class UsuariosController < ApplicationController
  # Función inicial, listando de Usuarios
  def index
    @usuarios = Usuario.find(:all) # Listar todos los usuarios
    respond_to do |format| # Responder según el formato, en este caso puede ser HTML o XML
      format.html # index.html.erb, en caso de que la respuesta sea HTML debería existir una vista con este nombre
      format.xml  { render :xml => @usuarios } # la variable @usuarios es la que despues se usara en la vista
    end
  end
  # Presenta un registro
  def show
    # Buscar un usuario por su id, params[:id] es creada por un POST 'id' o GET 'id', ej: /?id=dato
    @usuario = Usuario.find(params[:id])
    respond_to do |format|
      format.html # show.html.erb
      format.xml  { render :xml => @usuario }
    end
  end
  # Creación de un nuevo usuario, solo sirve para presentar la vista
  def new
    @usuario = Usuario.new
    respond_to do |format|
      format.html # new.html.erb
      format.xml  { render :xml => @usuario }
    end
  end
  # Edición, sirve para presentar la vista de edición
  def edit
    @usuario = Usuario.find(params[:id])
  end
  # Creación de un usuario cuando se procesa la forma de la vista new
  def create
    @usuario = Usuario.new(params[:usuario])
    respond_to do |format|
      if @usuario.save
        flash[:notice] = 'Usuario was successfully created.' # Crear una notificación
        format.html { redirect_to(@usuario) } # Redireccionar al listado
        format.xml  { render :xml => @usuario, :status => :created, :location => @usuario }
      else
        format.html { render :action => "new" } # Redirección a new en caso de que no se haya salvado correctamente
        format.xml  { render :xml => @usuario.errors, :status => :unprocessable_entity }
      end
    end
  end

  # Se elimina el registro en cuetión
  def destroy
    @usuario = Usuario.find(params[:id])
    @usuario.destroy
    respond_to do |format|
      format.html { redirect_to(usuarios_url) }
      format.xml  { head :ok }
    end
  end
end
```
Código 2.5

Debido a que el archivo del controlador es muy extenso para el documento no se lo presenta en su

totalidad pero ustedes puede verlo en la direción indicada
Como pueden ver se generó una buena cantidad de código para el controlador, Rails contiene una serie de generadores de código que genera el CRUD, esto nos permite crear maquetas de nuestra aplicación las cuales modificamos durante el desarrollo, el primer método o acción (en rails se llama acciones a lugares que pueden ser accedidos por el navegador) es el ***index***, en este método se listan los usuarios, en este caso llamamos a un listado en el modelo haciendo:
@usuarios = Usuario.fins(:all), este comando nos lista todos los usuarios y asigna a la variable ***@usuarios*** la lista para que pueda ser desplegada en la vista, el método ***show***, es el método que presenta el detalle de un solo usuario, el método ***new***, crea primero un usuario en el Modelo ejecutando *@usuario = Usuario .new*, esto nos crea un nuevo registro que aún no ha sido salvado y el cual es usado para poder crear a vista del mismo, en la vista hay un formulario, la vista debería estar en: (*app/views/usuarios/new.html.erb*), cuando se hace ingreso de los datos al formulario y se salva este se dirige al método ***create***, en el cual se almacena los datos del formulario, en caso de que no se puedan salvar los datos por errores se redirecciona a ***new***, pero en caso de que sea salvado correctamente se redirecciona al listado *redirect_to(@usuario)*, *básicamente la edición* ***edit*** sigue el mismo tratamiento solo que en este caso se realiza una búsqueda del registro que se desea editar, *@usuario = Usuario.find(params[:id])*, y se presenta la forma de edición que se direcciona a ***update***, en caso de que se salve correctamente se redirecciona al listado de lo contrario se regresa a la edición, en casó del método ***destroy***, se realiza la eliminación de un registro y no es necesaria una vista solo se redirecciona con el mensaje que se borró correctamente o no.

2.7 Vistas

Cada acción en el controlador debería tener una vista con excepción de algunas como ***destroy***, primero observemos el listado ***index:***

```
#Vista index: app/views/usuarios/index.html.erb
<h1>Listing usuarios</h1>

<table>
  <tr>
    <th>Nombre</th>
    <th>Apellido</th>
    <th>Login</th>
    <th>Pass</th>
  </tr>

<% for usuario in @usuarios %>
  <tr>
    <td><%=h usuario.nombre %></td>
    <td><%=h usuario.apellido %></td>
    <td><%=h usuario.login %></td>
    <td><%=h usuario.pass %></td>
    <td><%= link_to 'Show', usuario %></td>
    <td><%= link_to 'Edit', edit_usuario_path(usuario) %></td>
   <!--
   link_to crea un vínculo a la direción que indiquemos, indicamos diciendo primero
   'Texto del vinculo', :controller => 'usuarios' , :action => 'destroy', :confirm => 'Solicitud de confirmación'
   -->
    <td><%= link_to 'Destroy', usuario, :confirm => 'Are you sure?', :method => :delete %></td>
  </tr>
<% end %>
</table>

<br />

<%= link_to 'New usuario', new_usuario_path %>
```

Código 2.6

Esta es la vista del listado de usuarios que presenta una tabla con títulos de cada uno de los campos, excluyendo el ***id***, y que en las ultimas tres columnas crea un vínculo (link) para poder mostrar las

distintas acciones (index, show, edit, destroy). Comencemos haciendo una modificación en las vistas ***edit.html.erb*** y ***new.html.erb***, abran esos dos archivos y modifiquemos el código en el cual se ingresa el *tipo*, osea el tipo de usuario.

```
#Vista new: app/views/usuarios/new.html.erb
# Deben modificar esto
<p>
   <%= f.label :tipo %><br />
   <%= f.text_field :tipo %>
</p>
# Por esto
<p>
   <%= f.label :tipo %><br />
   <%= select "usuario", "tipo", ["usuario", "admin"].collect{|v| [v.capitalize, v]} %>
 </p>
```

Código 2.7

Hasta aquí terminamos con este extenso capítulo que nos mostró cuan fácil es realizar maquetas de aplicaciones con Rails además de poder aprender como es cada parte de una aplicación, debemos recoradar que esta no es la forma correcta de realizar una aplicación ya que esta muy incompleta, pero se puede utilizar sin modificaciones el código generado en partes donde se se realizan muchas modificaciones en la aplicación que posteriormente puedes ser mejoradas. En el siguiente capítulo veremos como son las migraciones y también como usar e instalar plugins que es una de las mayores fortalezas de Ruby on Rails.

3 Base de Datos y plugins

Hasta ahora hemos creado un scaffold para generar código pero ahora modificaremos el código generado por los generadores de Rails además de que veremos como instalar y usar el plugin Paperclip que facilita mucho el manejo de archivos. Comencemos con la estructura de la base de datos de nuestro programa Carrito

3.1 *Base de datos de la aplicación Carrito*

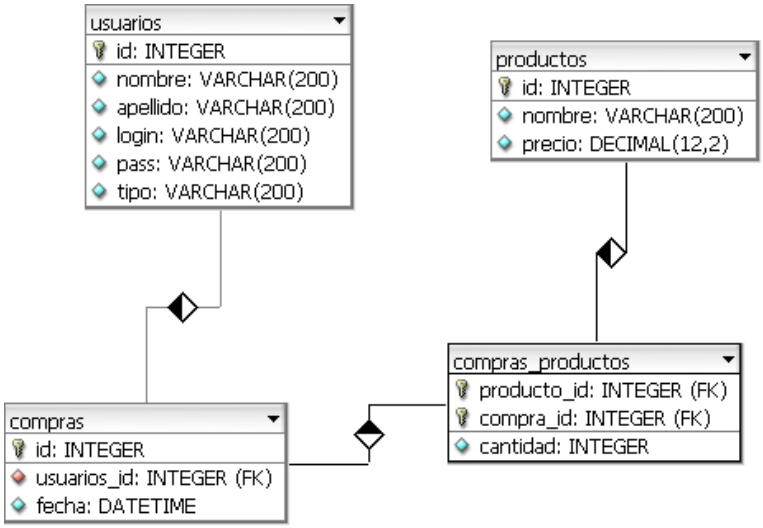


Ilustración 3.1: Base de Datos

En la base de datos podemos ver las tablas *usuarios*, *productos*, *compras* y el detalle de las compras *compras_productos*, para seguir con nuestra aplicación crearemos la tabla de productos para el cual generaremos un scaffold pero después les mostrare como usar un plugin para la poder asociar imagenes al producto, para generar el producto ejecutamos desde nuestra carpeta de trabajo.

```
# Generación de Usuarios
ruby script/generate scaffold producto nombre:string precio:decimal foto_file_name
rake db:migrate
```

Código 3.1

3.2 *Plugins (Paperclip)*

Una de las mayores fortalezas de Ruby on Rails es su comunidad que ha creado una gran cantidad de plugins, los plugins son programas creados por usuarios de Rails que nos permiten realizar cierto tipo de tareas como manejo de archivos, creación de árboles, manejo de tags y otros comencemos instalando el plugin ***paperclip*** para poder manejar las imágenes, para poder instalar ejecuten desde su carpeta de trabajo

```
# Instalando Papaperclip
ruby script/plugin install git://github.com/thoughtbot/paperclip.git
```

Código 3.2

Si no logran instalar descarguen el plugin de http://github.com/thoughtnot/paperclip/tree/master, y deben descomprimir y copiar en la carpeta *vendor/plugins*, osea que la ruta completa seria: *vendor/plugins/paperclip*, ahora debemos ir al modelo osea ***app/models/producto.rb***, y modifiquen el campo de texto de la imagen

```
# Modificación del Modelo app/models/producto.rb
class Producto < ActiveRecord::Base
  has_attached_file :foto,
                :styles => {:medio => '300x300>', :mini => '100x100>'}
end
```
Código 3.2

Lo que estamos indicando es como funcionará el plugin, al adicionar *has_attached_file*, le indicamos de que el modelo puede tener archivos adjuntos de cualquier tipo, pero este plugin nos permite manejar imágenes, en este caso solo usaremos imágenes y el prámetro *:styles* indica las dimensiones a las que redimensonará las imágenes, hemos definido dos tamaños *:medio* y *:mini*, pero es posible crear mas dimensiones, ahora debemos modificar las vistas debido a que Rails solo generó un campo de texto para poder almacenar la imagen, ahora es necesario ademas hacer modificaciones a la forma para poder hacer *uploads*, modifiquemos los las vistas ***new.html.erb*** y ***edit.html.erb*** de la siguiente forma:

```
# Modificación del Modelo app/views/productos/new.html.erb
# Primero modificamos esto
<% form_for(@producto) do |f| %>
# Por esto en ambas vistas, new.html.erb y edit.html.erb
<% form_for(@producto, :html => {:multipart => true}) do |f| %>
# Ahora debemos modificar el campo de la imágen, cambien esto
<p>
   <%= f.label :foto %><br />
   <%= f.text_field :foto %>
</p>
# Por esto
<p>
   <%= f.label :foto %><br />
   <%= f.field_field :foto %>
</p>
```
Código 3.3

Ahora podemos crear productos que ademas tendrán imágenes, tanto una pequeña de 100x100 y una grande de 300x300 para poder mostrar en la página, para poder mostrar las imágenes en los archivos ***indexhtml.erb*** y ***show.html.erb***, solo debemos realizar las siguientes modificaciones:

```
# Modificación de la vista app/views/productos/index.html.erb
# Para el index reemplazamos
<td><%= producto.foto_file_name %></td>
#Por
<td><%= image_tag producto.foto.url(:mini) %></td>
# Y modificamos app/views/productos/show.html.erb
<p>
  <b>Foto:</b>
  <%=h @producto.foto_file_name %>
</p>
# Por esto
<%= image_tag @producto.foto.url :medio %>
```
Código 3.4

Ahora podemos visualizar las imágenes que subimos al servidor, realmente el plugin paperclip nos permite el manejo de imágenes de una forma simple con muy pocos pasos y ahora es posible visualizar las imágenes que se hay subido al servidor, el plugin paperclip crea un directorio primero con el nombre del campo al cual le asignamos el nombre es este caso ***fotos***, dentro de este directorio creará un directorio con el *id* de cada registro con imágenes o archivos adjuntos y en caso de las imágenes crea tres directorios: ***medio***, ***mini*** y ***original***, medio y mini son los tipos de dimensión que definimos en el modelo y original es la carpeta donde se almacena la imagen original.

3.3 *Layouts*

Hasta ahora nuestra aplicación tiene una presentación muy burda la vista por defecto generada por Rails, pero es importante que una aplicación tenga una buena presentación, inclusive si es una pequeña aplicación por que generalmente cuando una aplicación tiene una mala presentación la gente suele pensar que la aplicación es de mala calidad. Primero debemos realizar una modificación en el Modelo de Producto ***app/models/product.rb***, aumenten lo siguiente despues de la definición del plugin paperclip:

```
# Aumentar al modelo app/models/producto.rb
def to_param
   "#{id}-#{nombre}"
end
```

Código 3.5

Lo que estamos haciendo aquí es permitir que los urls que se presenten en la aplicación sean mas claros y mas amigables para SEO (Search Engine Optimization). Una de las cosas mas importates de Rails es que siempre asume la convención sobre la configuración y se da este caso para el layout principal, creen el archivo ***app/views/layouts/application.html.erb***, básicamente esta es una plantilla base que será utilizada a través de la aplicación permitiendo mantener la misma consistencia de la aplicación a través de todo el sitio, para que pueda funcionar correctamente deben borrar las plantillas que generó automáticamente para productos y usuarios, borren los archivos: ***app/views/layouts/productos.html.erb,*** ***app/views/layouts/usuarios.html.erb***, como pueden ver Rails asumía estos archivos como las plantillas, en caso de que deseen darle una apariencia particular a un controlador solo deben crear su archivo en ***app/views/layouts*** y darle el nombre correcto del controlador, por ejemplo si el controlador es productos se crear productos.html.erb, o por ejemplo si tienen el controlador lista_objectos crean lista_objetos.html.erb, ahora acreguemos el siguiente código al archivo ***app/views/layouts/application.html.erb***

```
# app/views/layouts/application.html.erb
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
<title>Carrito de Compras</title>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<meta http-equiv="content-language" content="es">
<meta name="author" content="Boris Barroso">
<%= stylesheet_link_tag 'style.css' %>
</head>
<body>
  <div id="wrap">
    <div id="head">
      <a href="/"><img src="/images/Carrito.png" alt="Carrito" /></a>
    </div>
    <div id="content">
    <!--Contenido-->
    <%= yield %>
    </div>
  </div>
</body>
</html>
```

Código 3.6

Como ven las extensiones .**erb**, en Rails son las extensiones de las vistas, como pueden ver este template o plantilla nos presenta una estructura base a la cual le añadiremos posteriormente una hoja de estilos para darle una mejor apariencia.

3.3 *Mejorando la apariencia de la aplicación*

Ahora le daremos el maquillaje que requiere muestra aplicación y modificaremos las vistas principalmente de productos, comencemos modificando ***app/views/productos/index.html.erb***

```
# app/views/productos/index.html.erb
<h2>Listado de productos</h2>

<% for producto in @productos %>
  <div class="lista_productos">
    <h3><a href="<%= url_for producto %>"><%=h producto.nombre %></a></h3>
    <a href="<%= url_for producto %>"><%= image_tag producto.foto.url(:mini), :alt => producto.nombre
%></a>
    <div class="precio">$us <span><%=h producto.precio %></span></div>
    <div style="clear:both"></div>
  </div>
<% end %>
```

Código 3.7

Ahora adicinemos el css para que la aplicación luzca mejor, primero deben crear el archivo: ***public/stylesheets/style.css*** y añadir el siguiente código:

```
# public/stylesheets/style.css
body{font-family:verdana,arial,helvetica,sans-serif; background-color: #EFEFEF; }
#wrap{ background: #FFF; width: 700px; }
a { color:navy; text-decoration: none;}
a:hover{ text-decoration: underline; }
a img{ border: 0; }
.lista_productos{ padding: 4px; }
.lista_productos img{ float:left;}
.precio{ font-weight:bold; float:left; font-size:1.2em; }
.precio span{ color: red;}
.lista_productos .adicionar span{ float:left }
table { border-collapse: collapse; border:none; }
```

```
.carrito{ margin: 10px;}
.carrito tr{ border-top: 1px solid #AFAFAF; }
td, th{ padding: 4px; }
.carrito th{ padding: 4px 8px }
.carrito input[type=text]{ text-align:right }
```
Código 3.8

Ahora carrito tiene una mejor presentación además adicionamos css extra, revisen la dirección ***http://localhost:3000/productos***, que queremos sea la dirección inicial cuando entremos a nuestra página para ello deben borrar el archivo ***public/index.html***, y ahora deben modificar el archivo ***config/routes.rb***, busquen la línea con **# map.root :controller => "welcome"** y cambien esto por **map.root :cotroller => 'productos', :action => 'index'**, bueno solo la primera ves tal vez sea necesario indicar la acción ***:action***, pero la siguiente vez pueden borrar esta parte, ahora tenemos la lista de productos como la página principal. Modifiquemos también la vista para mostrar el producto ***app/views/productos/show.html.erb***.

```
# app/views/productos/show.html.erb
<h2><%=h @producto.nombre %></h2>
<%= image_tag @producto.foto.url :medio %>
<p class="precio">
  <b>Precio:</b>
  $us <span><%=h @producto.precio %></span>
</p>
<div style="clear:both"></div>
<%= link_to 'Listado de productos', productos_path %>
```
Código 3.9

Ahora tenemos una página Web mas elegante y la cual solo permite presentar información pero que todavía no nos permite seleccionar los productos de nuestro carrito además de poder modificar las cantidades.
Haremos algo mas para terminar, vamos a aumentar una línea en el código en el siguiente archivo para que tengamos una ruta especial para poder realizar la administración de los productos, abran el archivo ***config/routes.rb***, busquen la línea con **map.connect ':controller/:action/:id.:format'** y aumenten la siguiente línea después de esta:

map.admin 'admin', :controller => 'productos', :action => 'admin_index'

deben ir al controlador y aumentar ahora lo siguiente

```
# app/controlers/productos_controller.rb
class  ProductosController < ApplicationController
...
# Indice de administración de productos
   def admin_index
     @productos = Producto.find(:all)
   end
...
end
```
Código 3.10

La vista es la misma que la de de ***app/views/productos/index.html.erb*** solo que debemos adicionar las líneas para poder editar, adicionen lo siguiente

```
# app/views/productos/admin_index.html.erb
# adicionen esto en la tabla
<td><%= link_to 'Borrar', producto, :confirm => "Esta Seguro de Borrar \"#{producto.nombre}?\"", :method
=> :delete %></td>
# adicionen esto al final para adcionar nuevos productos
<%= link_to 'Nuevo producto', new_producto_path %>
```
Código 3.11

Ahora tenemos una vista para la administración, ya veremos como restringir el acceso para esta vista en el siguiente capítulo.

4 Sesiones, relaciones y ActiveRecord

4.1 Sesiones

Una de las partes de prácticamente cualquier aplicación Web son las sesiones que permiten almacenar información temporalmente que después puede ser procesada, en este caso tenemos que poder almacenar los productos que el usuario selecciona del carrito de compras en una sesión para que después la almacenemos en la base de datos, para manejar estos datos es necesario que creemos un controlador, el cual nos permitirá tener las vistas para poder almacenar la información del carrito Comencemos creando la sesión, ejecuten: **ruby script/generate sessions new carrito**, estos nos creara el controlador y las vistas, necesarias que necesitamos para poder conectarnos a la aplicación ademas de adicionar productos al carrito. Comencemos con el login, abran el archivo del controlador que recién creamos: ***app/controllers/sessions_controller.rb***

```
# app/controllers/sessions_controller.rb
# Definición
class SessionsController < ApplicationController
  def new
    # Verifica se el usuario ya se ha logueado
    if session[:usuario_id]
      redirect_to productos_path
    end
  end
  # Creación de sesión de un usuario
  def create
    # Busqueda usando metaprogramación
    @usuario = Usuario.find_by_login_and_pass(params[:login], params[:pass])
    if @usuario
      session[:usuario_id] = @usuario.id
      session[:nombre] = "#{@usuario.nombre} #{@usuario.apellido}"
      redirect_to productos_path
    else
      redirect_to '/sessions/new'
    end
  end
end
```

Código 4.1

El procedimiento **def new** lo que hace es verificar si es que el usuario ya inicio su sesión, en caso de que lo haya hecho lo redirecciona a la página de la lista de productos caso contrario le presentará la vista para que pueda ingresar, cuando el usuario ingresa los datos en el formulario de ingreso estos datos son enviados a la acción **create** del controlador **sessions** y se realiza una búsqueda con el login y password del usuario **Usuario.find_by_login_and_pass**, este método en realidad no existe sino que se crea en base a los campos que tiene la clase usuario, ahora que tenemos la parte del controlador necesitamos la vista, debemos crear el siguiente archivo: ***app/views/sessions/new.html.erb***

```
# app/views/sessions/new.html.erb
<h2>Ingreso de usuarios</h2>
<% form_tag '/sessions/create' do %>
  <p>
    <label for="login">Usuario:</label><br /><%= text_field_tag 'login' %>
  </p>
  <p>
    <label for="pass">Contraseña:</label><br /><%= password_field_tag 'pass' %>
  </p>
  <p><%= submit_tag 'Ingresar' %></p>
<% end %>
```

Código 4.2

Como pueden ver hemos creado una forma con **form_tag** y le pasamos la dirección **/sessions/create**, también hemos creado un campo **text_field_tag 'login'** y **password_field_tag 'pass'** que tendrán por nombres login y pass respectivamente. Ahora pueden ingresar al sistema, pero no hay forma visual de ver si uno esta conectado al sistema o no para eso debemos modificar: ***app/views/layouts/application.html.erb*** modifiquen en las siguientes líneas:

```
# app/views/layouts/application.html.erb
# Vayan a la línea “<div id="head">” y modifiquen de esta forma
<div id="head">
    <a href="/"><img src="/images/Carrito.png" alt="Carrito" /></a>
    <% if session[:usuario_id] %>
    Saludos <%= "#{session[:nombre]} | #{link_to '<img src="/images/exit.png" alt="Salir" /> Salir',
'/sessions/destroy'}" %>
    <% else %>
    <%= link_to 'Ingresar', '/sessions/new' %>
    <% end %>
    <% if session[:productos].length > 0 %>
      <a href="/sessions/carrito"><img src="/images/shopping_cart.png" alt="Carrito" />Ver carrito</a>
    <% end %>
  </div>
```

Código 4.3

Este código nos permite ver si estamos conectados al sistema así como presenta un vinculo del carrito de compras en caso de que hayamos seleccionado algún producto, podemos ver la sesión que se maneja en la vista igual que en el controlador, en cierta parte vemos ***<% if session[:productos].length > 0 %>*** que lo que hace es verificar si hay algún producto en caso de que haya presenta un vínculo para poder ver la lista de productos en el carrito de compras, pero de que nos sirve un vínculo al carrito de compras si es que no podemos adicionar productos a este, para ello deben modificar ***app/views/productos/index.html.erb*** y cambien la vista de la siguiente forma:

```
#  app/views/productos/index.html.erb
<h2>Listado de productos</h2>
<% for producto in @productos %>
  <div class="lista_productos">
    <h3><a href="<%= url_for producto %>"><%=h producto.nombre %></a></h3>
    <a href="<%= url_for producto %>"><%= image_tag producto.foto.url(:mini), :alt => producto.nombre
%></a>
    <div class="precio">$us <span><%=h producto.precio %></span></div>
    <br /><br />
    <div class="adicionar">
      <%= link_to "<span>Adicionar</span> <img src=\"/images/add.png\" alt=\"#{producto.nombre}\" />",
      "sessions/carrito/#{producto.id}"  %>
    </div>
    <div style="clear:both"></div>
  </div>
<% end %>
```

Código 4.4

Como ven hemos adicionado un vínculo con una imagen para que puedan adicionar algún producto a nuestro carrito de compras, este vínculo nos dirige a la dirección ***sessions/carrito/producto_id*** en la vista carrito implementaremos la lógica para poder administrar el carrito, abran el archivo: ***app/controllers/sessions_controller.rb*** y aumenten ahí las siguientes funciones:

```
# app/controllers/sessions_controller.rb
class SessionsController < ApplicationController
  ...
  # Procedimiento que permite adicionar y modificar los productos en el carrito
  def carrito
    # En caso de que se pase un producto que este en el carrito se adiciona suma la cantidad
    if params[:id] && request.get?
      producto_id = params[:id].to_i
      cantidad = session[:productos].key?(producto_id) ? session[:productos][producto_id]+1 : 1
      actualizar_productos producto_id, cantidad
    elsif request.post? # En caso de Post iterar a traves de todos los items
      params[:productos].each { |k, v| actualizar_productos k, v }
    end

    @productos = Producto.find(:all, :conditions => { :id => session[:productos].keys }) || []
  end

  # Borra un producto del carrito
  def borrar_producto
    id = params[:id].to_i
    session[:productos].delete(id)
    redirect_to '/sessions/carrito/'
  end

  protected
  # Actualiza o crear un item en la sesión
  def actualizar_productos producto_id, cantidad = 1

    producto_id = producto_id.to_i
    cantidad = cantidad.to_i
    if session[:productos].key?(producto_id)
      session[:productos][producto_id] = cantidad
    else
      session[:productos][producto_id] = 1;
    end
    # Eliminar en caso de que la cantidad sea cero
    session[:productos].delete(producto_id) if session[:productos][producto_id] <= 0
  end
  ...
end
```

Código 4.5

El método **def carrito** lo primero que hace es verificar que tipo de método se esta usando para pasar los parametros o variables en caso de que sea el método GET se copia a **producto_id** el parámetro **params[:id].to_i** y lo convertimos en un entero con **to_i** ya que las variables de **params** son de tipo String, y luego enviamos el id con la cantidad para que puedan ser actualizados los productos almacenados en al sesión con el método **def actualizar_productos**, en este método se verifica la cantidad del producto y se verifica si es que ya esta el producto en el Hash en caso de que exista el mismo producto solo se modifica la cantidad sino se adiciona el producto a la sesión **session[:productos]** este método no retorna ningun valor solo realiza modificaciones a la sesión, sigamos con la segunda parte del método **def carrito** cuando el método por el cual se pasa los parámetros es POST entonces se debe iterar a través de todos los productos que hayan sido seleccionados en el carrito y actualizar las cantidades, el método borrar_producto elimina el producto seleccionado del carrito y redirecciona a la lista de productos del carrito. Finalmente tenemos que crear la vista del carrito para que todo pueda funcionar correctamente, crean el archivo ***app/views/sessions/carrito.html.erb***

```
# app/views/sessions/carrito.html.erb
<h2>Productos seleccionados</h2>

<%= link_to 'Listar Productos', productos_path %>
<% form_tag '/sessions/carrito' do %>
  <table class="carrito">
    <tr>
      <th>Imágen</th>
      <th>Producto</th>
      <th>Precio</th>
      <th>Cantidad</th>
      <th>Subtotal</th>
      <th>Borrar</th>
    </tr>
  <% total = 0 %>
  <% @productos.each do |producto| %>
    <tr>
      <td><%= image_tag producto.foto.url(:mini), :alt => producto.nombre %></td>
      <td><%= producto.nombre %></td>
      <td align="right"><%= producto.precio %></td>
      <td><%= text_field_tag "productos[#{producto.id}]", session[:productos][producto.id], :size => 3  %>
</td>
      <td align="right"><%= session[:productos][producto.id] * producto.precio %></td>
      <td><small><%= link_to 'Borrar', "/sessions/borrar_producto/#{producto.id}" %></small></td>
      <% total += session[:productos][producto.id] * producto.precio %>
    </tr>
  <% end %>
    <tr style="background: #FFB32F">
      <td align="right" colspan="4"><strong>Total :</strong></td>
      <td align="right"><strong><%= total %></strong></td>
      <td> </td>
    </tr>
  </table>
  <%= submit_tag "Actualizar Carrito" %>
<% end %>
```

Código 4.6

Está es una forma completa debido a que tiene el detalle de los productos así como las cantidades y el total de la compra también permite la edición de las cantidades de los productos como su borrado. La siguiente imagen presenta el carrito de compras

Bueno hasta ahora podemos administrar nuestros productos, crear usuarios y muchas otras cosas pero aún no hemos registrado en la base de datos las compras del carrito, osea lo que generalmente en ingles le llaman el check out, pero para esto debemos relacionar los distintos modelos que hemos creado.

4.2 *Relaciones*

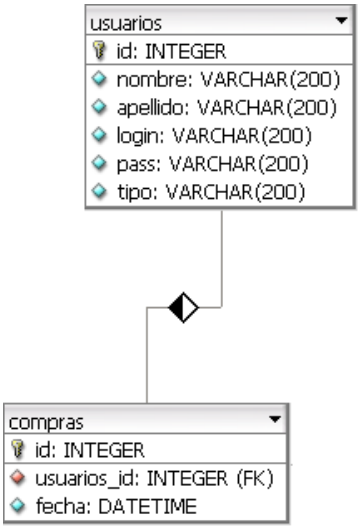
Las bases de datos tienen relaciones y los modelos de igual forma, como hemos podido ver los modelos son una representación de una tabla y mediante los cuales realizamos las consultas y modificaciones, pero los modelos proveen una capa Orientada a Objetos para poder representar el modelo relacional, en las Bases de Datos tenemos tres tipos de relaciones: 1 a varios y de varios a varios, en rails tenemos varios tipos de relaciones

- **has_many**: Este tipo de relaciones se da cuando un modelo esta realacionado con varios modelos de otro modelo, por ejemplo en nuestra base de datos un **usuario has_many compras**, la forma des escribir correctamente en el modelo sería

```
# app/models/usuario.rb
class usuario
  has_many :compras
end
```

Código 4.7

Como pueden ver usamos la forma plural para este caso **:compras** ya veremos y entenderán mejor cuando usar la forma plural y la singular para cada tipo de relación



- **has_one:** Es muy similar a has_many pero se usa la forma singular del modelo con el cual se relaciona en este caso lo existe un solo elemento con el cual se relaciona, la gráfica de la base de datos sería como la Ilustración 4.2 solo que el modelo solo permite que tenga una sola relación, el modelo se definiría de la siguiente forma

```
# app/models/usuario.rb
class usuario
  has_one :compra
end
```
Código 4.8

- **belongs_to:** Ahora esta relación nos define la dependencia de un modelo, en este caso el modelo compra depende de usuario

```
# app/models/compra.rb
class compra
  belongs_to :usuario
end
```
Código 4.9

- **has_and_belongs_to_many:** Este tipo de relación nos permite relacionar de varios a varios, utilizando una tabla intermedia para logra esto, y usamos la forma plural para poder relacionar los dos modelos relacionados en este caso (**producto, compra**)

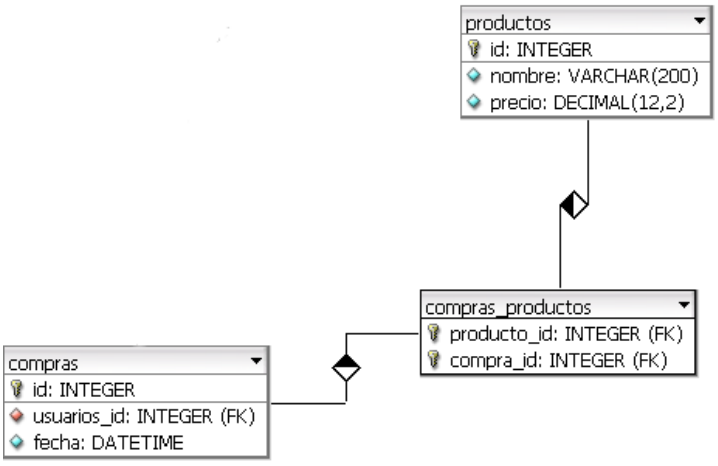


Ilustración 4.3: has_and_belongs_to_many

```
# app/models/compra.rb
class compra
  has_and_belongs_to_many :productos
end

# app/models/compra.rb
class producto
  has_and_belongs_to_many :compras
end
```
Código 4.10

- **has_many :through:** En algunos casos existen relaciones muy similares a la relación **has_and_belongs_to_many** osea de *n a m* pero que requieren almacenar otros datos aparte de la relación en la tabla intermedia, es cuando usamos este tipo de relación para la cual debemos crear el modelo para la tabla intermedia, como pueden ver ActiveRecord se encarga de pluralizar los modelos y nosotros en este caso hemos definido la clase compras_producto de forma singular, osea active record solo ve el final y le aumenta una ***s*** y en algunas casos la forma plural del modelo por ejemplo cancion => canciones, pero no lo hace con palabras en español sino que en ingles, osea que muchas veces tenermos que definir la forma plural de un modelo, pero eso no lo veremos ahora

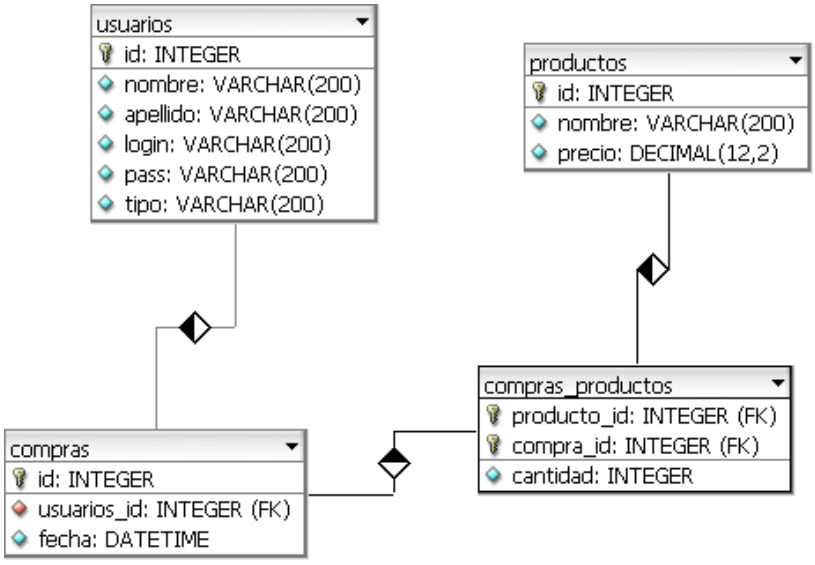


Ilustración 4.3: has_many :trough

Como pueden ver en el caso de carrito necesitamos guardar en la tabla compras_productos la cantidad de un producto, y tambien podriamos guardar otros datos, como el precio en caso de que se modifiquen los precios en periodos de tiempo o haya una oferta

```
# app/models/compra.rb
class compra
  has_many :compras_productos
  has_many :productos, :through => compras_productos
end

# app/models/producto.rb
class producto
  has_many :compras_productos
  has_many :compras, :through => compras_productos
end

# app/models/compras_producto.rb
class compras_producto
  belongs_to :producto
  belongs_to :compras
end
```
Código 4.11

- **has_one :through;** Muy similar a **has_many :trough** solo que usamos la forma singular del modelo

```
# app/models/compra.rb
class compra
  has_one :compras_producto
  has_one :producto, :trough => compras_producto
end

# app/models/producto.rb
class producto
  has_one :compras_producto
  has_one :compra, :trough => compras_producto
end

# app/models/compras_producto.rb
class compras_producto
  belongs_to :compras
end
```

Código 4.11

Bueno una ves que explicamos los distintos tipos de relaciones que existen ahora debemos usar una de ellas para poder relacionar nuestros modelos en carrito., primero crearemos un modelo ejecuten el siguiente comando desde su carpeta de trabajo

```
ruby script/generate model compras_producto producto_id:integer compra_id:integer
cantidad:integer
```

Código 4.13

Como pueden ver aquí también podemos ver otras de las convenciones que usa Rails para las llaves foráneas, se usa la forma singular del modelo seguido de ***_id***, es de esta forma que la base de datos se estructura en una aplicación Rails y en general este patrón es ampliamente aceptado en muchos otros frameworks de otros lenguajes. Una ves generado el modelo ejecuten

rake db:migrate

Esto generara la tabla *compras_productos*, ahora usemos las relaciones para poder hacer la salida y registro de nuestra compra. Lo primero que debemos hacer para que esto funcione es aumentar un vinculo que nos permita realizar la salida vayan a ***app/views/sessions/carrtio.html.erb*** y modifiquen

```
#  app/views/sessions/carrtio.html.erb
#Aumenten la siguiente línea al principio
<% if session[:productos] && session[:productos].length > 0 && session[:usuario_id] %>
  <br /><strong><%= link_to 'Registrar Salida <img src="/images/next.png" />', '/compras/new' %></strong>
<% end %>
```

Código 4.14

Lo cual nos presenta un vínculo para poder realizar la salida o checkuot, ahora debemos generar el modelo compra, ejecuten

```
ruby scipt/generate scaffold compra usuario_id:integer fecha:datetime
```

Código 4.15

Esto nos generará el controlador, el modelo y una vista para poder almacenar la compra, en este caso nos generará el código para realizar altas, bajas y modificaciones, solo necesitamos el de creación, el index comencemos modificando el modelo de compras, productos y compras_producto, no necesitamos ninguna vista y controlador para el modelo compras_producto, entonces todo el proceso de almacenar el detalle del carrito de compras lo realizamos en el modelo ***compra***,

```
ruby scipt/generate model compras_producto compra_id:integer producto_id:integer cantidad:integer
```

Código 4.16

Ahora necesitamos modificar los modelos para que pueda existir las relaciones entre ellos

```
# app/models/producto.rb
class Producto < ActiveRecord::Base
  has_many :compras_productos
  has_many :compras, :through => :compras_productos
end
```
Código 4.17

```
# app/models/compra.rb
class Compra < ActiveRecord::Base
  has_many :compras_productos
  has_many :productos, :through => :compras_productos, :source => :compra
  belongs_to :usuario
end
```
Código 4.18

Como pueden ver en caso de este modelo hemos adicionado **:source => :compra**, esto es debido al nombre del modelo que usamos para poder relacionar entre compras y productos, como se dan cuenta el modelo **compras_producto** usa a producto en su forma singular mientras que compra esta en su forma plural

```
# app/models/compras_producto.rb
class ComprasProducto < ActiveRecord::Base
  belongs_to :compra
  belongs_to :producto
end
```
Código 4.19

```
# app/models/usuario.rb
class Usuario < ActiveRecord::Base
  has_many :compras
end
```
Código 4.20

4.3 La consola

Una de las grandes ventajas que tenemos en ruby es que se puede acceder a una consola en la cual se puede ejecutar el código, en este caso vamos a ingresar a la consola de nuestro proyecto que contiene las clases de Ruby asi como algunas más de Rails, para poder entrar a la consola ejecuten desde su carpeta de trabajo

ruby script/console

Esto nos cargará el ambiente en el cual estamos trabajando y podremos experimentar en la consola comandos, ahora ejecuten

```
>>p = Producto.find(:all) # Retorna una lista de productos
>>p.[0] #Presenta el primer productos del array
>>p.[0].nombre # Muestra el nombre del producto
```
Código 4.21

Como pueden ver podemos realizar las consultas directamente en la consola, lo cual nos permite experimentar con cosas sencillas, tanto de Ruby como de Rails