Universidad Católica

Teoría y aplicaciones de la informática 2

Desarrollo de aplicaciones web con Ruby on Rails

Humberto J. Aquino

2005

Introducción

Rails es un framework web escrito enteramente en el lenguaje de programación Ruby. Se lo llama full-stack debido a que posee todas las capas de una aplicación completa, entre las que se encuentran:

- Capa de persistencia de datos (ActiveRecord)
- Capa de aplicación (ActionPack)
- Capa de Vista (ActionPack)
- Capa de mail (ActionMailer)
- Capa de Web Services (ActionWebService)

Una de las grandes ventajas de que sea un framework full-stack es que se necesita menos código en comparación a la mayoría de los demás frameworks(especialmente los de java) debido a que no se debe desperdiciar tiempo en configuraciones de archivos XML 'interminables'.

Rails sigue el principio Don't Repeat Yourself (No lo repitas). El poder que ofrece Rails y su base Ruby permiten versatilidades únicas en un ambiente altamente dinámico.

Rails utiliza el famoso design pattern llamado MVC(Model-View-Controller) para separar concerns en estas 3 grandes categorías.
Luego veremos que Rails establece ciertas convenciones para que el desarrollo de la aplicación sea mas ágil y rápida, así como escalable y fácilmente entendible.

Rails se basa en el manifiesto ágil el cual en resumen cita:
- Individuos e interacción en vez de procesos y herramientas.
- Software en funcionamiento sobre documentación comprensiva.
- Colaboración del cliente en vez de contrato de negociación.
- Respuesta al cambio sobre seguir un plan.

Rails integra el framework Runit y ofrece la posibilidad de testear por capa la aplicación. Además se puede testear el performance y realizar Mock Tests.
Todo esto es parcialmente realizable gracias al build automatation tool(BAT) de Ruby llamado Rake (Similar a Ant, NAnt, Make) aunque no se establecen las reglas con algún lenguaje distinto al de programación, sino con Ruby en si, lo que da todo el poder del lenguaje al BAT.
También posee script generadores de código para facilitar la construcción de modelos, vistas y controladores, así como scaffolds(esqueletos) que permiten realizar operaciones CRUD(Create, Retrive, Update, Delete) sobre el modelo.
Da estadísticas del sitio e integra un logueador llamado log4r análogo al famoso logueador de Java log4j.
Permite realizar operaciones transaccionales que siguen el principio ACID(Atomic, Consistency, Isolation, Durable).
Permite enviar y recibir mail de una manera limpia gracias al modulo ActionMailer.
Integra el sistema de documentación Rdoc con el cual se puede generar documentación del proyecto.
Posee mecanismos de cacheo que permiten mejorar el performance.
Y por ultimo, integra técnica llamada AJAX (Asynchronous JavaScript and XML) para permitir interacciones dinámicas con el cliente web, similares a las proveidas en aplicaciones como gmail.

Rails posee 3 modos en los cuales puede correr, los cuales son:

- Producción: Es el ambiente utilizado cuando se utiliza en el mundo real la aplicación. Su performance incrementa y para que los cambios realizados a la aplicación sean vistos se debe hacer un reload de la aplicación.
- Desarrollo: Es el ambiente por defecto. Se loguean todas las actividades en el directorio logs, al realizarse un cambio en la aplicación este se nota al instante y las excepciones son mostradas cuando

ocurren.
- Testeo: Se utiliza al correr los test cases.

Contenido

Rails establece una estructura de directorio. Esto se puede generar ejecutando el comando:

rails mi_aplicacion

En el directorio **app** se encuentran estos directorios:

- apis Carpeta de librerías propias de la aplicación. En caso de ser librerías a utilizar o que ya se están utilizando en otros proyectos, se debería de guardar en la carpeta libs Si son librerías third-party se guardan en la carpeta vendor.
- controllers Se encuentran las clases que heredan de ApplicationController que manejan los requests y los responces a traves de acciones definidas como métodos.
- helpers Aquí se guardan las clases 'ayudadoras' que se utilizan en toda la aplicación o en parte de ella. Son códigos simples que no pertenecen a una librería en si. En otro caso se deben de

guardar en la carpeta lib.

- models Las clases modelo se guardan en esta carpeta. Son clases que heredan de ActiveRecord::Base para poseer mecanismos automáticos de persistencia de datos.
- Views Se encuentran los archivos .rhtml y .rxml que representan las vistas.
  - Layout Son .rhtml que se utilizan en muchas vistas.

El directorio **components** posee componentes reutilizables que se utilizan en la implementación.
El directorio **config** contiene los siguientes archivos:

- database.yml configuración de la conexión a la base de datos de producción, desarrollo y testeo.
  Ejemplo de este archivo de configuración:

```
development:
  adapter: mysql
  database: dev_app
  host: localhost
  username: webuser
  password: xxxxxx

test:
  adapter: mysql
  database: test_app
  host: localhost
  username: webuser
  password: xxxxxx

production:
  adapter: mysql
  database: prod_app
  host: localhost
  username: webuser
  password: xxxxxx
```

- environment.rb configuración de las variables de ambientes.
- environments directorio que contiene variables de ambiente propias del modo runtime, sea este de testeo, desarrollo o producción.
- routes.rb configuración del enrutamiento.

El directorio **db** posee los schemas de la base de datos. Este se utiliza para cargar la base de datos con los meta datos necesarios para que la aplicación inicialice el RDBMS.

El directorio **doc** contiene el archivo README_FOR_APP en el cual se describe información de la aplicación que sera adherida a la documentación generada por el comando 'rake appdoc'.

El directorio **lib** contiene clases y módulos utilizados como librerías en la aplicación.

El directorio **log** contiene los siguientes archivos respectivos al logueo dependiendo del modo en que se encuentra corriendo la aplicación:

- development.log
- production.log
- server.log
- test.log

El directorio **public** contiene:

- 404.html Esta pagina se utiliza durante la producción cuando hay un request invalido, en caso de no sobreescribir las reglas de ruteo.
- dispatch.cgi
- dispatch.rb
- images Directorio en el cual se guardan las imágenes de la aplicación.
- Javascripts Directorio con los archivos .js
- 500.html Esta pagina se utiliza cuando existe un error en la aplicación en el ambiente de producción.
- dispatch.fcgi
- favicon.ico
- index.html Pagina de inicio.
- stylesheets Carpeta con los .css utilizados en la aplicación.

El directorio scripts contiene los siguientes scripts:

- benchmaker Obtiene las mediciones de performance de uno o mas métodos en la aplicación.
- breakpointer Es un cliente que permite interactuar con la aplicación durante su ejecución.
- console permite utilizar irb para interactuar con los métodos de la aplicación.
- destroy borra los archivos generados por generate.
- generate es el generador de código. Puede crear controladores, mailers, modelos, scaffolds, web services, modulos de logueo, entre otros. Esta lista va creciendo gracias al aporte de la comunidad.
- profiler crea un perfil de tiempo de ejecución de partes del código de la aplicación.
- runner ejecuta un método de la aplicación fuera del contexto de la web.
- server Servidor de prueba basado en WEBrick. Se utiliza normalmente cuando se desarrolla.

El directorio **test** contiene:

- fixtures posee registros cargables en la base de datos en formato YALM. Se utiliza para cargar los registros en la base de datos al realizar los test units.
- functional se utiliza para realizar los tests a los controladores
- mocks se utiliza par emular la interacción con otros sistemas o con el usuario.
- test_helper.rb Módulos, clases y métodos utilizados en el testeo para ayudar a realizarlos de manera fácil.
- unit Carpeta con los test cases correspondientes al modelo.

El directorio **vendor** posee librerías externas al proyecto que no fueron escritas por el desarrollador de la aplicación rails.

El archivo RakeFile posee reglas, las cuales pueden ser vistas al escribir el comando 'rake --tasks'. La salida del mismo es la siguiente:

```
rake apidoc                      # Build the apidoc HTML Files
rake appdoc                      # Build the appdoc HTML Files
rake clear_logs                  # Clears all *.log files in log/
rake clobber_apidoc              # Remove rdoc products
rake clobber_appdoc              # Remove rdoc products
rake clone_structure_to_test     # Recreate the test databases from the
```

```
                                # development structure
rake db_structure_dump               # Dump the database structure to a SQL file
rake default                         # Run all the tests on a fresh test database
rake doc                             # Generate API documentatio, show coding
                                     # stats
rake environment                     # Require application environment.
rake purge_test_database             # Empty the test database
rake reapidoc                        # Force a rebuild of the RDOC files
rake reappdoc                        # Force a rebuild of the RDOC files
rake recent                          # Run tests for
                                     # recentclone_structure_to_test
rake stats                           # Report code statistics (KLOCs, etc) from
                                     # the application
rake test_functional                 # Run tests for test_functional
rake test_units                           # Run tests for test_units
```

Convenciones de Rails

Rails posee ciertas convenciones cuando se trata de clases, Módulos, y métodos.
Las clases empiezan con una letra capital para inicio de palabra, similar a los nombres de clases establecidos en la especificacion JavaBeans.
Ej:

```
class Auto; end
class AutoRojoDeJuguete; end
class PersonaInteligente; end
module AlgunModulo; end
```

Los nombres de los métodos se separan con el caracter de linea baja '_'.
Ej:

```
def metodo_numero_uno; end
def destroy_index; end
def redirect_to; end
```

Los nombres de variables también se escriben con la misma regla de los métodos.

Además se tienen convenciones para cada elemento del patrón MVC.

Convención para el modelo:

Tabla: producto_caros
Clase: ProductoCaro
Archivo: app/models/producto_caro.rb

Convención para el controlador:
URL: http://.../personas/new
Clase: PersonasController
Archivo: app/controllers/personas_controller.rb
Método: new()
Layout: app/views/layout/personas.rhtml

Convención para la vista:
URL: http://.../personas/new
Archivo: app/views/personas/new.rhtml (o .rxml)

Modulo: modulo PersonasHelper
Archivo: app/helpers/personas _helper.rb

Las tablas son nombradas de acuerdo al plural del nombre de la clase. Aunque esto puede ser establecido conforme se necesite. Con Rails uno siempre puede eludir las convenciones. Esto es bueno para bases de datos legacy. Hay que
tener en cuenta que rails utiliza un mecanismo de racionalización pero en el idioma ingles. Esto se extenderá a otros idiomas mas adelante.

También, cada tabla debe de poseer un campo llamado id(tiene que ser en minúsculas) del tipo primario y marcado con autoincremento en el caso de MySQL. Esto se puede cambiar en caso de tener una BD legacy pero si se esta diseñando una nueva aplicación es recomendable hacer esto. Las tablas que emulan la relación many-to-many no necesitan tener el campo id.

ActiveRecord

Este modulo se utiliza para dar persistencia automática y semi transparente a las clases modelo. Esto se realiza heredando de ActiveRecord::Base. Gracias a esto, la clase heredadora, de acuerdo a su nombre y al archivo database.yml, rails sabe que tabla le corresponde a la clase y que atributos posee la misma.

Se heredan métodos y se generan otros:

- Por cada atributo de la tabla que mapea a una clase del modelo se obtienen los siguientes métodos, por ejemplo si se tiene la columna nombre, se crean los siguientes métodos:
  - nombre #Método getter que devuelve un objeto Ruby de un tipo equivalente establecido en la tabla 1.
  - nombre=(value) #Método setter
  - nombre_before_type_cast #accede al atributo nombre antes de realizar el type cast. Se puede utilizar eso para controlar patrones en la cadena.
  - save #Hace persistente al objeto y sus objetos dependientes dependiendo de las relaciones que posea. Retorna nil en caso de no poder realizar la acción.
  - save! #Realiza lo mismo que save pero lanza una excepción en caso de no poder realizar la acción.
- Genera los siguientes métodos de clase:
  - find #Devuelve un array con objetos dependiendo de los parámetros.
    Ej:
    - Cliente.find(:all, :conditions => [“nombre = ? and apellido = ?”, nombre, apellido])
      #devuelve todos los clientes con el respectivo nombre y apellido.
    - Cliente.find(3) #devuelve el cliente con id numero 3.
  - columns #devuelve una array de objetos Column.
  - attributes #devuelve un hash con los nombres de los atributos y sus respectivos valores.
  - count #cuanta la cantidad de elementos dependiendo de la condición pasada como parámetro.

La lista es aun mayor aunque por simplicidad, solo se nombraron los anteriores. También se tienen métodos para actualizar el modelo, crear y persistir en un paso, destruir objetos, entre otros.

Las relaciones entre clases del modelo son:

- one_to_one especifica una relación de uno a uno con otra clase.
- has_many especifica que se poseen cero o mas elementos de otra o la misma clase.

  Ej:

```
class Persona < ActiveRecord::Base
        has_many :autos
end
class Auto < ActiveRecord::Base
        belongs_to :persona
end
```

Cada persona posee cero o mas autos y cada auto pertenece a solo una persona. Se crean métodos como:

```
p = Persona.new("yo")
a = Auto.new("Ferrari")
p.autos << a
p.autos[0]                  # devuelve el ferrari
p.autos.size        # devuelve 1
a.persona                   #devuelve el objeto persona "yo"
```

- has_one: especifica que tiene cero o un elemento de otra o la misma clase.

  Ej:

```
class Perro < ActiveRecord::Base
        has_one :collar
end
class Collar < ActiveRecord::Base
        belongs_to :perro
end
```

Automáticamente se crean métodos para setear desde un objeto perro un objeto collar y viceversa.

- belongs_to: Especifica que una clase pertenece a otra.
- has_and_belongs_to_many: Establece una relación de muchos a muchos entre dos clases.

Ej:

```
class Alumno < ActiveRecord::Base
        has_and_belongs_to_many :clases
end
class Clase < ActiveRecord::Base
        has_and_belongs_to_many :alumnos
end
```

- Para relaciones de herencia se realiza de manera normal aunque se debe crear una tabla con el nombre de la clase que hereda directamente de ActiveRecord::Base y agregar todos los campos de todas las subclases a esta tabla además de un campo llamado type de tipo varchar de cantidad igual al mayor nombre de alguna clase heredadora.

  Ej:

```
class Persona < ActiveRecord::Base
end

class Alumno < Persona
end
```

```
class AlumnoBocho < Alumno
end
```

Entonces se crea una tabla llamada personas con todos los campos de Persona, alumno y alumno bocho además del campo type que en este caso tendrá por lo mínimo 11 caracteres.

También se pueden crear relaciones sintéticas que se generan con consultas sql.

Validators
Muchas personas se quejan que la imposición del campo id como primary key no es bueno, aunque se ha visto que en la practica es mejor dejar las validaciones propias de la lógica de negocios en el modelo y solo utilizar la base de datos como albergue de datos. Permite que la aplicación sea escalable y permite una mejor separation of concerns.
Los validators son:

- validate_on_create
- validate_on_update
- validates_acceptance_of
- validates_associated
- validates_confirmation_of
- validates_each
- validates_format_of
- validates_inclusion_of
- validates_length_of
- validates_numericality_of
- validates_presence_of
- validates_size_of
- validates_uniqueness_of

Todos son validadores que se utilizan usualmente aunque en caso de necesitar tener algún validador sofisticado, se puede sobrecargar el Método validate. T
Todas las validaciones se encuentran en el modelo y son heredables. Son validaciones que corresponden a la lógica de negocios propia de la aplicación y la clase. Se utilizan debido a que están bien integradas a la vista, que saca información de las validaciones para informar al usuario de sus fallas al llenar formularios por ejemplo.

Callbacks
Se utilizan para manejar el ciclo de vida de los objetos. Esto permite realizar ciertas tareas antes o después de ciertas acciones.

- after_create
- after_destroy
- after_save
- after_update
- after_validation
- after_validation_on_create
- after_validation_on_update
- before_create
- before_destroy
- before_save
- before_update

- before_validation
- before_validation_on_create
- before_validation_on_update

ActionPack

Posee dos modulos: ActionController y ActionView.
El controlador se encarga de pasar el request a la acción adecuada así como cargar los parámetros.
Ej:

http://localhost/micontrollers/accion1/algo=3

Al realizar este request se llama a la acción accion1 del controlador MicontrollerController.

```
class MicontrolleraController < ApplicationController
    def accion1
        eltres = @params[:algo]
        #eltres posee el valor del parámetro algo que es 3
    end
end
```

Básicamente se pueden crear controladores y acciones en los mismos. Y redireccionar requests entre controllers y acciones. En el controlador se cargan las variables de atributo que se usaran en las vistas para mostrar los resultados.
Ej

```
    def accion1
        @user = User.find @params[:id]
    end
```

@user podrá ser usado en el templare acction1.rhtml para para sacar información del usuario y mostrarla. Todo esto es gracias a que se paso como parámetro el id del usuario. Durante el desarrollo de la aplicación se ve una pagina con la excepción en caso de que el usuario no sea encontrado. Pero en la producción esto generalmente se redireccionara a una pagina 404.html.

Rails También posee mecanismos stateful a traves de sesiones. Cualquier clase se puede guardar en las sesiones gracias al mecanismo de serializacion de Ruby. El único requisito es que la clase del objeto que se guardara en la sesión debe de ser especificada en el controlador donde se utilizara la clase.
Ej:

```
    PersonasController < ApplicationController
        model :perro
    end
```

Aquí perro es usado en la sesión cuando se encuentre en el controlador personas.

ActionView

Contiene clases ayudadoras y se encarga del maneado de templates.

Ej:

    En el archivo accion1.rhtml tenemos

```
<html>
	<head></head>
	<body>
		Hola <%= @user.nombre %>. Como te va?<br/>
		La hora es <%= Time.now %><br/>
		Voy a imprimir 10 numeros<br/>
		<% 10.times do %>
			<%= rand 10 %>
		<% end %>
	</body>
</html>
```

En este caso se interpreta el el código entre <%= %> y se lo reemplaza con su equivalente de tipo cadena llamando al Método to_s propio de cada objeto.
Si se encuentra entre <% %> entonces solo se interpreta el código.

ActionView posee muchos helpers para formatear la salida, crear formularios, crear vínculos con o sin imágenes, campos de fecha, check boxes, radio buttons, entre otros. El mismo se encarga de manejar el html de cada uno de los elementos citados y cargarlo a los objetos o llamar al Método adecuado según se desee.

También posee layouts que son porciones de rhtml que se pueden compartir entre controladores o utilizarlos de manera única por controlador.
Existen los partials que son porciones de rhtml que se pueden insertar en cualquier rhtml para hacer el código de la vista mas modular y no estar escribiendo el mismo rhtml.
Permite crear componentes los cuales poseen funcionalidad y son reutilizables fuera del contexto de la aplicación, siempre que sean bien diseñados.

Ajax

Ajax es una técnica que extiende el modelo tradicional de aplicación web que permite que partes del html sean actualizadas en paralelo, sin tener que recargar la pagina de manera completa, siempre que sea esto lo que se quiere.
Mayormente se utiliza la librería orientada a objetos prototype para utilizar ajax en Rails.
En realidad son requests xml que se envían al servidor para que cierta acción se encargue de actualizar parte de la pagina con la información adecuada.
Esto permite tener aplicaciones mas eficientes y fluidas. Lo bueno de Rails es que inclusive este tipo de técnicas pueden ser testeadas.
Muchas personas llaman a esta técnica WEB 2.0 debido a los grandes avances en la usabilidad e impresión gráfica que dan al utilizar ajax.
Rails posee helpers que facilitan la programación de vistas con ajax.
Se pueden actualizar periódicamente secciones de paginas o interactuar con el browser.

ActionMailer

Es un componente que permite a la aplicación enviar y recibir e-mails. También se puede testear el envío y recepción correctos de mail sin tener configurado un sendmail o un servidor pop3 o imap.
Se pueden usar templates o generar dinámicamente e-mails. Todo esto es disparado cuando se ejecuta

un acción, siempre que esto sea especificado o pre configurado.

ActionWebServices

Maneja soporte de lado servido para protocolos SOAP y XML-RPC.
Gracias a esto se pueden crear interfaces para dar servicios a otros sitios escritos en otros lenguajes de programación que sigan los mismo estándares. De esta manera se puede interactuar fácilmente con la aplicación.
Y como todo modulo de rails, se puede testear de fácil manera.

Seguridad

Rails posee mecanismos para prevenir el famoso ataque “SQL Injection”. Este problema se origina cuando cuando se pasan directamente los valores de los parámetros a las consultas sql. Por suerte, ActiveRecord posee un mecanismo que permite evitar este tipo de ataques.

Ej:

```
nombre = @params[:nombre]
@clientes = Clientes.find(:all, :condition => “nombre = '#{nombre}' ”)
```

Para este caso se pasa el parámetro nombre directamente a la consulta. Esto permite que el atacante ponga como nombre comandos sql que se ejecutaran. Es por esto que Rails aconseja que lo mismo se realice de la siguiente manera.

```
nombre = @params[:nombre]
@clientes = Clientes.find(:all, [:condition => “nombre = ? ”, nombre])
```

De esta manera controla la cadena nombre para invalidar cualquier comando sql potencialmente peligroso.

También posee un metodo que evita el ataque Cross-scripting CSS/XSS.

Ej:

```
<%= @actividad.comentario %>
```

Al ejecutarse el método comentario de la variable @actividad, el mismo retorna la cadena que se haya guardado en algún momento. Si el usuario malicioso ingreso un código javascript o algún otro, el mismo se ejecutara. Es por esto que Rails posee el método h() que reemplaza <, >, entre otros caracteres por sus equivalentes gráficos los cuales no son ejecutables.

```
<%= h(@actividad.comentario) %>
```

Conclusión

Rails es un framework de alto nivel que posee características de sus predecesores Java. Su curva de aprendizaje es pequeña y permite incrementar la productividad de los programadores y diseñadores. Al comienzo cuesta acostumbrarse al modo en el cual trabaja pero con el tiempo uno se da cuenta de las buenas decisiones de diseño implementadas en el framework. El mismo es ágil y a la vez divertido. Es un entorno completo y profesional así como reliable. No se lo puede comparar con aplicaciones realizadas con php o asp ya que seria como comparar una ferrari con un triciclo. Rails utiliza varias metodologías para que el desarrollo sea de alta calidad y escalable.
Una contra es su corto periodo de vida, aunque la aceptación en la comunidad de desarrolladores fue

prácticamente instantánea.

Bibliografía

Ruby on Rails http://www.rubyonrails.org
Ruby http://www.ruby-lang.org
Rubygems http://rubygems.org/
Rdoc http://rdoc.sourceforge.net/
Rake http://rake.rubyforge.org/
http://www.martinfowler.com/articles/rake.html
log4r http://log4r.sourceforge.net/
Ajax http://www.adaptivepath.com/publications/essays/archives/000385.php
Agil Manifesto http://agilemanifesto.org