Ruby on Rails: Porque hay mejores maneras de cortarse las venas que escribiendo PHP

Gunnar Wolf — gwolf@debian.org
http://www.gwolf.org/soft/rails

Instituto de Investigaciones Económicas, UNAM
Desarrollador del proyecto Debian

Encuentro Nacional de Linux y Software Libre
Octubre 19, 2007

Bien. Veo que tengo tu atención

Bien. Veo que ya tengo tu atención.

No, pese a lo que el título te de a pensar, esta no es una plática orientada a hablar mal de PHP.

Esta es una plática orientada a hablar *bien* de las buenas prácticas de programación hacia las cuales nos *orilla* el framework Ruby on Rails — Y como parte de ello, desmenuzaremos parte de las razones de por qué una tan gran parte del código escrito en PHP resulta ser una sopa inmantenible.

Pero no adelantemos vísperas. Vamos a lo que esta presentación *sí* es.

Temas

1. Definiciones
2. Ruby: Un lenguaje pragmático
3. Rails: Un framework para el desarrollo Web
4. Ok... ¿y los demás?

¿Qué es Ruby on Rails?

- Un framework para el desarrollo de aplicaciones Web
- Basado en el patrón MVC
- Orientado a aplicaciones modeladas sobre una base de datos relacional
- Utilizando el lenguaje Ruby
- Fuertemente *opinioneado*
- ...Una de tantas palabrejas de moda

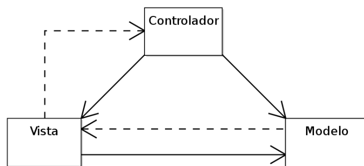
¿Framework?

- Muchos comenzamos a escribir aplicaciones Web procesando, generando e integrando cada uno de los componentes de nuestro sistema
- Un *framework* nos evita tener que encargarnos de las partes más repetitivas del desarrollo, de la integración de nuestro sistema
- Y al hacerlo, se asegura de que sigamos la lógica general de *sus* desarrolladores

El patrón MVC

- Patrón *Modelo, Vista, Controlador* (Trygve Reenskaug, 1979)
- Patrón de arquitectura de sistemas que separa los componentes de un sistema en tres roles claramente separados — Modelo, Vista y Controlador
- Este patrón *no se limita a sistemas Web* (fue descrito formalmente por primera vez en 1979), aunque estos típicamente se ajustan a él a la perfección
- No hace falta trabajar dentro de un framework para escribir sistemas MVC - Pero ayuda a mantener la separación y el flujo de información presentes.

El patrón MVC



http://es.wikipedia.org/wiki/Modelo_Vista_Controlador

Modelo La representación de los datos y las relaciones entre ellos (llamado también *lógica de negocios*)

Vista La interfaz al usuario

Controlador Quien recibe los eventos solicitados a través de la vista, los *empuja* al modelo, y genera/refresca a la vista

Sobre una base de datos relacional

- Los modelos en Rails (casi siempre) utilizan una base de datos relacional como medio de *persistencia*
- Una parte fundamental de Rails es su ORM (*mapeo objeto-relacional*), `ActiveRecord`; nos permite tratar directamente a cada uno de los registros como objetos de Ruby. Trabajando con Rails, es muy poco frecuente escribir SQL (aunque siempre es importante poder hacerlo).
- `ActiveRecord` es un ORM muy inteligente, permitiéndonos expresar fácilmente relaciones y validaciones complejas
- Rails asume una base de datos *tonta*, al estilo MySQL — Busca que las validaciones y restricciones radiquen en los modelos, no en la base. Es posible utilizar bases de datos más potentes y aprovechar sus ventajas — aunque a costa de repetirnos (ver más adelante: `DRY`)

Fuertemente *opinioneado*

Los desarrolladores de Rails decidieron impulsar toda una *cultura* de desarrollo de software, a través de diversos principios de desarrollo (que detallaremos un poco más adelante), entre ellos:

- Metodologías de desarrollo ágil
- Convención sobre configuración
- DRY: ¡No te repitas!
- Andamiaje
- Generadores
- Desarrollo basado en pruebas

Frameworks: ¿Buenos? ¿Malos? ¿Feos?

- Muchas veces, nos cuesta cambiar de nuestra forma tradicional de desarrollo para ajustarnos al estilo de pensamiento de un framework. Sin embargo, muy probablemente lo terminaremos agradeciendo
- Por definición, comenzar a trabajar con un framework implica aprender un *muy extenso* API. Vale la pena invertir el tiempo, para poder realmente aprovechar las ventajas (y comprender la idiosincracia) de un framework
- A diferencia de un lenguaje (que no debe tomarnos como tope más que un par de días aprender), aprender a usar un framework *de modo óptimo* nos puede tomar varios meses.

Frameworks: ¿Buenos? ¿Malos? ¿Feos?

- Particularmente el framework Ruby on Rails es muy extenso, inclusive comparando con otros de propósitos similares
- La documentación disponible en línea (`http://api.rubyonrails.org`), típicamente instalada además junto con Rails, es muy extensa y amigable
- Hay excelentes recursos, tanto en línea como en forma de libro impreso.
- Lo repito: Comenzar a emplear un framework puede ser un paso difícil para quien está acostumbrado a trabajar *a mano alzada*... Pero las ventajas comienzan a aparecer casi de inmediato.

Temas

1. Definiciones
2. Ruby: Un lenguaje pragmático
3. Rails: Un framework para el desarrollo Web
4. Ok... ¿y los demás?

¿Ruby? ¿Qué es eso?

- Un lenguaje de programación
- Limpia y completamente orientado a objetos
- Dinámicamente extensible
- Con una fuerte dosis de programación funcional
- Libre de paradigmas
- Con fuerte influencia de Perl, Smalltalk, Python, Lisp, Dylan y CLU (`http://es.wikipedia.org/wiki/Ruby`)
- Creado por Yukihiro *Matz* Matsumoto, a partir de 1993
- Enfatiza en la comodidad al *programador* más que a la máquina

Limpia y completamente orientado a objetos

- Lenguajes como Java *dicen ser* el epítome del purismo de la programación orientada a objetos — Pero no lo son. Constantemente manipulamos *cosas* que no son objetos (por ejemplo, los mismos números)
- En Ruby, *todo es un objeto*, y toda operación que hagamos sobre un objeto será llamando a uno de sus métodos
- Desde los operadores base del lenguaje (`+`, `-`, `[]`, `[]=`, etc.), todo está implementado como métodos
- ...Y todos estos métodos admiten, claro, que modifiquemos su comportamiento

Limpia y completamente orientado a objetos

- Ruby implementa únicamente herencia simple
- Para lograr comportamientos similares al de la herencia múltiple, tenemos el mecanismo de *mixins*, que nos permite mezclar módulos dentro de la definición de una clase
- El estilo de orientación a objetos de Ruby es conocida como *Duck Typing*:
  Si tiene cara de pato, camina como pato y hace "cuac" como un pato, ¡es un pato!

Dinámicamente extensible

- Una clase en Ruby no es algo cerrado, terminado
- Ni siquiera las clases del sistema, que pueden ser ampliadas en cualquier momento
- Hay unas simples convenciones para los nombres de los métodos
  - Un método terminado en `?` entrega resultado booleano
  - Un método terminado en `!` es destructivo (altera el contenido original del objeto, perdiendo lo que tuviera antes)
  - Un método que inicia con `to_` entrega la transformación del contenido del objeto a otra clase (por ejemplo, `to_s` entrega una cadena, `to_i` un entero). Muchas clases base ofrecen estos métodos, pero siempre podemos extenderlos (*incluso en la clase original*)

Limpia y completamente orientado a objetos

Example (Modificando una clase *en vivo*)

```
irb(main):001:0> class Fixnum
irb(main):002:1> def lastDigit; self %10; end
irb(main):003:1> def isEven?; self %2 == 0; end
irb(main):004:1> end
=> nil
irb(main):005:0> 105.lastDigit
=> 5
irb(main):006:0> 105.isEven?
=> false
irb(main):007:0> 106.isEven?
=> true
```

Con una fuerte dosis de programación funcional

- Una parte importante de la *cultura* de Ruby es aprovechar la programación funcional a través de funciones anónimas, *closures* y continuaciones
- Toda expresión entrega un resultado; el resultado de una función es el último valor evaluado en ella (no requiere un `return` explícito, aunque sí lo admite)
- Fuerte soporte a la introspección y metaprogramación

Con una fuerte dosis de programación funcional

La sintaxis funcional se integra de manera muy natural con la sintaxis orientada a objetos del lenguaje, a través de los *bloques*

Example (Uso de programación funcional en Ruby)

```
irb(main):005:0> class Fixnum
irb(main):006:1> def countDown; return 0 if self<=0;
[self, *(self-1).countDown]; end
irb(main):007:1> def isEven?; self%2 == 0; end
irb(main):008:1> end
=> nil
irb(main):009:0> 10.countDown.select {|num|
num.isEven?}.map {|num| num.to_f}
=> [10.0, 8.0, 6.0, 4.0, 2.0, 0.0]
```

Libre de paradigmas

- El que todo sea un objeto no nos obliga a mantenerlo en mente constantemente
- Podemos escribir código completamente procedimental, funcional, o no estructurado
- ...Aunque a fin de cuentas, todo termina siendo convertido a objetos al ser evaluado por la máquina virtual

Enfatiza en la comodidad al *programador* más que a la máquina

Yukihiro Matsumoto

Often people, especially computer engineers, focus on the machines. They think, "By doing this, the machine will run faster. By doing this, the machine will run more effectively. By doing this, the machine will something something something." They are focusing on machines. But in fact we need to focus on humans, on how humans care about doing programming or operating the application of the machines. We are the masters. They are the slaves.

- El *principio de la menor sorpresa*
- Lenguaje muy extenso, con una curva de aprendizaje larga, paulatina, nunca abrupta

Y un larguísimo etcétera

- Ruby es un lenguaje sorprendentemente fácil de aprender — ¡y de aprender con gusto!
- Tiene aún varias desventajas
  - Está aún en un proceso de cambio muy fuerte
  - Muchos de los módulos (*gemas*) disponibles no están correctamente documentados
  - La velocidad de ejecución y la utilización de memoria son claramente inferiores a los lenguajes con que más frecuentemente se le compara (Perl, Python, PHP)
- ¡Pero definitivamente es una valiosa herramienta para cargar en nuestro arsenal!

Temas

1. Definiciones
2. Ruby: Un lenguaje pragmático
3. Rails: Un framework para el desarrollo Web
4. Ok... ¿y los demás?

¿Qué es Rails?

- Un framework para el desarrollo de aplicaciones Web basado en Ruby
- Creado por David Heinemeier Hansson, liberado en abril del 2004
- La versión 1.0 fue liberada a principios del 2006
- Ha detonado toda una revolución en la creación de frameworks similares para otros lenguajes

Convención sobre configuración

- La mayor parte de los frameworks Web nos hacen tedioso el inicio de un proyecto por la gran cantidad de detalles de configuración que hay que indicarle
- Rails está basado en el principio de *convención sobre configuración*: Hay una serie de convenciones que podemos seguir de modo que no tenemos que explicitarlos
  - Algunos tipos de datos, campos sugeridos (`id`, `created_at` y `updated_at`, etc.)
  - Cómo se llaman las tablas y las clases, así como las tablas de relación (y qué va en singular, qué va en plural — y cómo se determina cómo singularizar/pluralizar)
  - Cómo se organizan los archivos en nuestro árbol
  - ...

DRY: ¡No te repitas!

El código no repetido no sólo es una pérdida de tiempo una vez, sino que muchas

- Dar mantenimiento a pedazos repetidos de código es cada vez más pesado
- Requiere que todos los desarrolladores del proyecto conozcan todos los detalles del proyecto *para no romperlo* al reimplementar alguna parte
- Separar nuestro código en bibliotecas, o utilizar *ayudantes* en las diferentes pedazos del código, nos permite aplicarlo en diferentes proyectos

Migraciones, ActiveRecord y SQL

- Una de las principales fuentes de problemas en muchos proyectos es que los programadores pueden ser excelentes en un lenguaje y no tanto en otro...
- Los ORMs como ActiveRecord *esconden* el SQL, presentándonos una interfaz directa en nuestro lenguaje de aplicación... Pero no resuelven cómo crear la estructura de datos
- Rails incluye el concepto de *migraciones*, que nos permiten definir o modificar la estructura de nuestra BD desde Ruby, y además, hacerlo de manera independiente del motor de BD que elijamos
- Las migraciones nos permiten además manejar versiones de todo el entorno de nuestra aplicación, simplificándonos en gran medida el despliegue a servidores de producción

Generadores

- La estructura básica del código, aunque corta, tiende a ser repetitiva — ¡No te repitas!
- Con el modelo común de Rails (y en general, bajo MVC), un archivo rara vez es creado sólo
  - Un modelo implica una migración y pruebas de unidad
  - Un controlador implica pruebas de funcionalidad, vistas y ayudantes
- Los *generadores* nos ahorran esta talacha

Andamiaje

- Pero más allá - La mayor parte de la funcionalidad de una aplicación Web clásica se resume en las operaciones *CRUD* (Create, Replace, Update, Delete) sobre nuestros modelos
- El *andamiaje* (*scaffolding*) nos permite crear la estructura global que permite estas tareas
- El andamiaje no sirve para todo. Es un acercamiento burdo pero funcional, que nos permite hacer las primeras entregas a un cliente, presentando principalmente la estructura global de nuestra información (esto es, la correspondencia entre modelos)
- El andamiaje **no está pensado para ser utilizado en producción**... Pero reduce dramáticamente nuestros tiempos de desarrollo

Desarrollo basado en pruebas

A lo largo del ciclo de desarrollo, es común que se rompan pedazos de funcionalidad, o se introduzcan bugs, por los cambios que vayamos realizando.
Rails facilita la creación de pruebas que ayuden a asegurar que cada aspecto del sistema se mantenga operativo en todo momento.

Pruebas de unidad
: Aquellas que se aplican a relaciones entre modelos, así como sus validaciones

Pruebas de funcionalidad
: Aquellas que verifican los efectos de la llamada directa a un método de un controlador

Pruebas de interacción
: Aquellas que emulan el comportamiento de un usuario — Verifica la salida que recibe el usuario

Las facilidades de análisis que ofrecen estas tres plataformas a sus respectivos niveles son sencillamente impresionantes.

Extenso esquema de templates

Rails incluye varios esquemas para crear templates, dependiendo del tipo de contenido que requiramos generar:

- La mayor parte de templates para HTML consisten en archivos `.rhtml`, que incluyen fragmentos de Ruby a mezclados en el código con una sintaxis comparable con la de PHP, Emberl, ASP o JSP
- Los templates para generación de XML (`.rxml`) son muy sencillos, básicamente delínean la estructura del docuemnto a generar, y son poblados con los elementos que instancie el controlador
- Tenemos la opción (como veremos más adelante, rara vez hace falta) de generar contenido Javascript directamente utilizando templates `.rjs`

Integración transparente hacia AJAX

- Hoy, todo mundo quiere que sus aplicaciones Web sean *altamente interactivas*, que estén cargadas de AJAX
- Eso está muy bien... Pero requiere conocer bien a Javascript... O más bien, a las diferentes versiones de Javascript, con sus incompatibilidades y bugs
- Rails nuevamente nos invita a no escribir más que en Ruby: Incluye las funciones necesarias para generar todo tipo de funcionalidad AJAX (desde comunicación asíncrona hasta efectos gráficos) sin tocar una línea de Javascript
- Esta funcionalidad descansa principalmente en las bibliotecas libres de Javascript *Prototype* y *Scriptaculous*, ampliamente probadas

Temas

1. Definiciones
2. Ruby: Un lenguaje pragmático
3. Rails: Un framework para el desarrollo Web
4. Ok... ¿y los demás?

No, no olvido el título de la presentación

- Para llegar a esta sección, era necesario explicar la estructura y las ventajas de trabajar con un framework completo como Rails
- ...Y ayuda mucho utilizar un lenguaje apto para la aplicación, como Ruby
- Ahora sí, vamos al punto medular: ¿Qué tiene de malo PHP? ¿Por qué equiparo su uso con *cortarse las venas?*

PHP no es inherentemente malo o inferior

- Hay *excelente* código escrito en PHP
- PHP en sus últimas versiones ofrece facilidades muy completas para programar del modo necesario para diseñar *muy bien* un sistema
- ...Sin embargo, la forma en que típicamente se enseña PHP, y la cultura ya dominante en su comunidad, han llevado a una cantidad inaudita de código inmantenible
- Y no deja de ser cierto: El mismo Rasmus Lerdorf reconoce que PHP *no fue pensado* como un lenguaje de propósito general; él buscaba crear tan sólo un sistema poderoso de templates
- Lo que señalo a continuación es tan sólo lo que he observado al intentar dar mantenimiento a código de PHP de todo tipo

Mezcla de código con presentación

- Históricamente, PHP fue presentado como *un lenguaje fácil de aprender* para la creación de sitios Web dinámicos
- Históricamente, PHP no impulsó prácticas de programación seguras o extensibles
- ...Y es de entenderse — PHP es un *lenguaje de templates* que se popularizó sin control. No fue concebido como un lenguaje de propósito general, aunque haya evolucionado hacia allá
- Incluso con las últimas versiones, es demasiado tentador comenzar a hacer un sistema sin planeación alguna — Sobre los mismísimos templates.

Organización de sistemas basada en *caminitos*, no agrupada lógicamente

- Típicamente, cada una de las acciones, cada uno de los componentes que recibirán una solicitud del cliente, es un archivo independiente — ...Y esos archivos son, justamente, el template (o la *vista*) en cuestión
- Eso lleva a que organicemos nuestro sistema en base a una serie de *caminitos* de interacción, lo que los convierte en un montón de archivitos difíciles de mantener coherentemente
- Nuevamente, un *buen* programador es capaz de hacer maravillas. Un programador novato o un aprendiz... Es capaz de escribir su propia carta suicida

¿Realmente queremos que el diseñador y el programador trabajen tan juntos?

- Una de las grendes promesas cuando se popularizaron los templates era que facilitarían la integración del trabajo de programadores y diseñadores Web
- Esto se cumple muy marginalmente. Pocos diseñadores y pocas herramientas de diseño de sitios realmente respetan las etiquetas con código
- Muchas estructuras (ciclos, bloques condicionales, etc.) son difíciles de entender, incluso para la mejor de estas herramientas
- Resulta mucho más adecuado que el diseñador monte los *layouts* generales y determine las demás estructuras utilizando CSS
- Sí... Diseñar sistemas Web requiere de diseñadores más capacitados que diseñar sitios.

Una gran cantidad de bibliotecas... Exponiendo interfaces anticuadas

- Casi todos los lenguajes ofrecen y empleam una biblioteca común para conectarse a cualquier motor de bases de datos. PHP también... Pero poca gente los usa. ¿Por qué? No lo sé. Muchos desarrolladores siguen usando las clases independientes para cada RDBMS
- El modelo de orientación a objetos de PHP5 es completo y limpio. Hay muchísimos módulos, de uso extendido, que siguen requiriendo las prácticas más anticuadas y menos recomendadas
- Nuevamente, parte importante del problema es la cantidad de código antiguo de donde la gente aprende, de donde los programadores nuevos toman ejemplo
- Si vas a escribir (y especialmente si vas a publicar) PHP... Hazlo sembrando un buen ejemplo

No soy quién para decir qué usar y qué no

- Soy, a fin de cuentas, sólo un programador más. Y ni siquiera uno muy bueno.
- Estoy seguro que hay muy buen código —en todo lenguaje— esperando ser escrito
- Ni siquiera seguir las mejores prácticas es garantía de escribir código de calidad

La conclusión, por fin...

Cuando escribas (o hagas) cualquier cosa, por menor que sea piensa en el ejemplo que estás poniendo a quien aprenda de tí. Nunca sabemos quién nos está observando, y qué está aprendiendo de nosotros.

¿Dudas? ¿Comentarios? ¿Cebollazos?

¡Gracias!

Gunnar Wolf — `gwolf@gwolf.org`