

Proyecto de Fin de Carrera

Pizarra Web Compartida

Albert Llop

7 de diciembre de 2008

Índice general

1. Desarrollo: Ruby on Rails	2
1.1. Consideraciones previas	3
1.2. Primer ciclo	5
1.2.1. Análisis de requisitos	5
1.2.2. Diseño	5
1.2.3. Implementación	5
1.3. Segundo Ciclo	10

Capítulo 1

Desarrollo: Ruby on Rails

Se dispone en estos momentos de un motor completo en Javascript que permitiría todo el proceso necesario para los objetivos de este proyecto. Éste, sin embargo, necesita de un fondo que le aporte todo lo que html estático y javascript no es capaz de hacer, como por ejemplo, tratar con la base de datos. Por las razones comentadas en el primer capítulo, se ha considerado que Ruby on Rails sería la solución para implementar este fondo, y en este capítulo se comenta el proceso seguido, resaltando en cada caso las diferencias de utilizar Ruby on Rails frente a soluciones más clásicas como PHP o ASP.

1.1. Consideraciones previas

Antes de nada, para entender como funciona Ruby on Rails, hay que aclarar ciertos conceptos. Primero, el lenguaje en que se está programando es Ruby. Éste es un lenguaje interpretado, que al igual que lenguajes de scripting típicos como Perl, PHP o Python, no es necesario compilar. De hecho, existen implementaciones de interpretadores de Ruby en varios lenguajes, siendo la implementada en C la *oficial*, pero existiendo otras tan dispares como jRuby, una implementación en Java que permite utilizar cualquier librería de Java.

Desde un principio su creador pretendió hacer de ruby un lenguaje muy *natural* a la hora de ser leído, y lleva al extremo el concepto de lenguaje de alto nivel. Entre sus características se encuentra el considerar absolutamente un objeto (incluyendo tipos básicos), permitiendo generar líneas tan autoexplicativas como las siguientes:

```
100.times do
  print "No hablaré en clase".upcase
end
```

Incluso con la naturalidad con la que se se escribe y se lee Ruby, no hace de él un lenguaje simple, y es lo suficientemente robusto y funcional como para permitir escribir cualquier tipo de aplicación. Sin embargo, allá donde ha triunfado más, es en el mundo del desarrollo web, donde cada vez más, se busca la agilidad de desarrollo más que una gran eficiencia del código. Las características de las webs hacen que sea posible lograr una gran eficiencia una vez una web está funcionando, mediante métodos como el cacheo, que hacen que la mejora de eficiencia que se pueda lograr con otros lenguaje de más bajo nivel, sean minúsculas.

Rails es un Framework escrito en ruby que permite un desarrollo web mucho más ágil y sencillo. Se base en hacer fácil el trabajo del programador, y en su famoso *Convention over configuration* (convención antes que configuración). Debido a que la mayoría del tiempo, el desarrollo de webs es un proceso repetitivo, es posible establecer una serie de patrones que asumir, y solo modificar en los casos especiales en que *lo normal* no es suficiente.

Está basado en una arquitectura MVC (Modelo Vista Controlador), permitiendo aplicar el patrón en 3 capas estudiado las diversas asignaturas de Ingeniería del Software, así como la mayoría de buenas prácticas, hasta ahora prácticamente imposibles de aplicar en el mundo web.

Rails (en parte gracias a la sencillez de Ruby), promueve la práctica del desarrollo ágil de software (Agile software development), una metodología de desarrollo que se basa en aligerar el proceso de desarrollo, alejarse de metodologías *burocráticas*, centrándose en conseguir software de alta calidad de forma muy rápida. Estas características son muy apreciadas en el mundo del desarrollo web, puesto que como ya se ha comentado, la eficiencia suele ser secundaria, y los procesos, al ser repetitivos en su mayoría, hacen de otras metodologías demasiado pesadas y lentas.

A lo largo de este capítulo se irán remarcando los puntos por los cuales Ruby es tan adecuado al desarrollo web, porque Rails permite un desarrollo mucho más fácil, y porque una metodología

basada en la escasez de documentación y planificación es posible, y de hecho, beneficiosa, en el contexto de este proyecto (y en el de otros similares de desarrollo de webs).

1.2. Primer ciclo

La metodología de desarrollo ágil defiende un desarrollo iterativo, por ciclos pequeños que generen de forma rápida ciertas funcionalidades. Cada ciclo amplía funcionalidades hasta que al final estén todas incluidas. Cada ciclo debe ser completo, con sus fases de análisis de requisitos, diseño, implementación y revisión. La fase de documentación tiene como resultado este capítulo.

En este primer ciclo, se pretende obtener todas las funcionalidades necesarias para poder crear y editar documentos.

1.2.1. Análisis de requisitos

Gracias a que se ha tratado la implementación del Javascript de forma previa a este ciclo, se conocen ya los requisitos básicos necesarios para poder utilizar el motor de dibujo. De forma detallada, las funcionalidades requeridas para este ciclo son las siguientes:

- Poder crear, editar y borrar documentos, dándoles un título, una descripción y asignándole una cantidad de páginas fija, mediante interacción normal por HTML.
- Tener soporte para documentos, páginas y los elementos contenidos en ellas, de forma que el motor de comunicación en Javascript pueda comunicarse con la aplicación para editar las pizarras.

1.2.2. Diseño

El primer paso natural a la hora de planear un ciclo en Ruby on Rails es planeando la estructura de la aplicación en cuanto a modelos y controladores se refiere. Los modelos serán necesarios para poder guardar los datos en la base de datos, y deben estar relacionados adecuadamente de forma que sea fácil y directo acceder a datos de modelos *cercanos*. También es importante considerar los controladores en los que agrupar las funciones. Sería posible implementar la aplicación entera en un controlador enorme, pero obviamente esto no sería práctico ni productivo.

Modelos

El diagrama de clases es extremadamente sencillo en este ciclo. Solamente son necesarias tres clases, una para representar el documento, otra para las páginas y otra para los elementos. Durante la discusión sobre la implementación del javascript ya se discutió que la descripción de cada elemento se guardaría como una cadena de texto JSON representativa de los elementos que se crean mediante javascript, por lo tanto, un elemento no necesita más variedad de atributos.

Controladores

En estos momentos se considera suficiente agrupar todas las funcionalidades que traten con los documentos dentro de un solo controlador, `documents_controller`.

1.2.3. Implementación

Para inicializar una aplicación en rails, hay que crear dicha aplicación, y configurar la conexión con la base de datos. Desde el principio, se le ha dado el nombre `drawme`.

```
rails drawme
```

Modelos

Este comando generará los archivos que formen el esqueleto de la aplicación. Una vez hecho esto, hay que generar los modelos que se han planeado:

```
script/generate model Document
script/generate model Page
script/generate model Element
```

Estos comandos generarán los archivos dentro de la carpeta `/app/models`, además de las *migraciones* correspondientes para crear las tablas en la base de datos. Una migración es una acción sobre la base de datos, de cualquier tipo. Dentro de la carpeta `/db/migrations` se encuentran, ordenadas de forma temporal mediante un timestamp. Gracias a esto se pueden ir realizando modificaciones sobre la base de datos de forma ordenada, permitiendo así, por ejemplo, que si en un futuro se pretende realizar una modificación, no se tenga que eliminar la base de datos entera y volverla a crear.

A forma de ejemplo, la migración para la tabla del modelo `Document` será así:

```
class CreateDocuments < ActiveRecord::Migration
  def self.up
    create_table :documents do |t|
      t.string :title
      t.text :description
      t.integer :current_page, :default => 1
      t.integer :height, :default => 600
      t.integer :width, :default => 800
      t.timestamps
    end
  end

  def self.down
    drop_table :documents
  end
end
```

Ya en este punto se empieza a notar el concepto de *convention over configuration*, en que se ha creado el modelo `Document`, y la migración generada llama a la tabla de la base de datos `documents`. Ésto siempre es así, un modelo es un nombre en singular, y su tabla es el equivalente en plural. Sabiendo esto, no habrá en ningún momento problemas de nomenclatura, y en caso de que sea necesario llamar a una tabla con un nombre distinto al de su modelo, será necesario añadir una línea en su modelo correspondiente (en este caso `/app/models/document.rb`).

Un documento puede tener múltiples páginas, y es bien sabido que este tipo de relaciones, para ser representadas en una base de datos relacional, hay que añadir una clave externa en la tabla de las páginas. Para hacer esto, en Ruby on Rails se haría lo siguiente:

```
class CreatePages < ActiveRecord::Migration
  def self.up
    create_table :pages do |t|
      t.integer :number, :null => false
      t.references :document
```

```

        t.index :document_id

        t.timestamps
      end
    end

    def self.down
      drop_table :pages
    end
  end
end

```

Esta migración es totalmente autoexplicativa. Por convención, los campos que referencian otra tabla, tienen la nomenclatura `modelo_id`, y se aprovecha en este caso para añadir un índice sobre este campo, para que cuando sea necesario consultar las páginas de un documento, no sea necesario recorrer la tabla entera.

La migración para `Element` es igual a la de `Page`.

```

class CreateElements < ActiveRecord::Migration
  def self.up
    create_table :elements do |t|
      t.text :attr, :null => false
      t.references :page
      t.index :page_id

      t.timestamps
    end
  end

  def self.down
    drop_table :elements
  end
end

```

Para trasladar estas migraciones a la base de datos es necesario ejecutar la línea siguiente en la línea de comandos:

```
rake db:migrate
```

Estas migraciones, no obstante, solamente sirven para tratar con la base de datos. Para que la aplicación sepa que un `Document` tiene múltiples `Pages`, es necesario dejar constancia en los modelos:

```

class Document < ActiveRecord::Base
  has_many :pages, :dependent => :destroy
  has_many :elements, :through => :pages
end

class Page < ActiveRecord::Base
  has_many :elements, :dependent => :destroy
  belongs_to :document
end

```



```
class Element < ActiveRecord::Base
  belongs_to :page
  belongs_to :document, :through => :page
end
```

Con estas líneas, será posible en un futuro, ejecutar instrucciones como las siguientes:

```
document.pages << Page.new # Añadir una página a un documento
page.elements << Element.create(:attr => "...") # Añadir un elemento a una página
document.pages.clear # Borrar todas las páginas de un documento
```

Cualquier operación que trate las relaciones entre estos elementos será posible de forma tan sencilla, siempre tratando con la base de datos de forma transparente.

Controladores y vistas

De forma similar a los modelos, para generar los archivos necesarios para un controlador:

```
script/generate controller Documents
```

El archivo generado, utilizando las convenciones, será `/app/controllers/documents_controller.rb`. Dentro de este archivo, cada función implementada tendrá una traducción directa en las capa de vistas, por lo tanto el proceso de implementar acciones en controladores suele ir al mismo ritmo que se van implementando las vistas.

Debido a que es el primer ciclo, y que de momento aún no se tiene muy claro cómo acabará estructurándose la web, se optará por tener lo que en Rails se llama un *Scaffold*, con unas pocas modificaciones, para poder editar los documentos. Un Scaffold es una envoltura para un modelo, con las cuatro operaciones básicas que se pueden querer realizar: crear, mostrar, modificar y borrar (**create**, **show**, **update** y **destroy**). Debido a la naturaleza de la web, es necesario otras acciones, previas a estas cuatro. Por ejemplo, antes de crear un Documento, es necesario mostrar un formulario en el que rellenar los datos necesarios. Dicha acción se llamaría **new**. Para poder mostrar un elemento en concreto, primero hay que listar los que hay, y dicha acción se llama **index**. Para poder modificar un elemento primero hay que mostrar como está en este instante, además de proporcionar un formulario con el que poder especificar los cambios. Ésta acción se llamaría **edit**. Para eliminar un elemento solo es necesario saber qué elemento eliminar, por lo tanto es posible ejecutar tanto desde **show** como desde **index**.

Éstas cuatro acciones forman el llamado **CRUD** (Create, Retrieve, Update y Delete), y equivalen a las cuatro comentadas, que junto con las otras tres, forman las siete acciones básicas de un controlador de Scaffold.

Puesto que estas acciones son tan comunes, Rails proporciona una forma de generar un Scaffold sencillo de forma totalmente automática, para utilizar como punto de partida.

```
script/generate scaffold Document
```

Gracias a las convenciones de Rails, este Scaffold buscará el modelo **Document** y creará el controlador **documents_controller** con el código adecuado, y las vistas correspondientes.

Este scaffolding, sin embargo, no tiene en cuenta las relaciones entre modelos, ni ningún comportamiento personalizado que se quiera tener. Así, los cambios necesarios para adecuar este scaffold al gusto de la aplicación serían los siguientes:

- En el momento de crear un documento, añadirle un número fijo de páginas vacías, con sus números de páginas correspondientes.
- El atributo `current_page` no debe ser editable, de momento, puesto que se manejará mediante la interfaz javascript.

Con estos sencillos pasos se puede considerar la primera funcionalidad objetivo cumplida. La segunda funcionalidad es permitir tratar con la interfaz, mediante la comunicación con ajax y transportando elementos en formato JSON. Las funciones necesarias para ello se pueden resumir en las siguientes:

`add_element` recibiendo para qué página (y por consiguiente, qué documento), y el objeto en JSON, y retornando el identificador del mismo una vez guardado.

`remove_element` recibiendo el identificador del elemento.

`list_elements` recibiendo la página (y por consiguiente, el documento), devolviendo el listado de elementos en formato JSON.

`change_page` recibiendo la página a la que se quiere cambiar.

Cabe destacar de la implementación de estas funciones, la facilidad que da Rails para tratar con peticiones en Ajax. En el caso de la función `list_elements`, por ejemplo, lo lógico sería realizar una búsqueda en la base de datos, teniendo luego un vector de elementos, que luego se deberían traducir a JSON, y mostrarse en la vista.

```
def list_elements
  elements = Page.find(params[:page_id]).elements
  render :json => elements
end
```

Debido a que JSON se ha convertido en un formato standard en este tipo de acciones, Rails usa las librerías de Ruby para la conversión de objetos a sus equivalentes en JSON, y mediante una sola línea es posible devolver un objeto exactamente equivalente al que se tiene en ese momento en Ruby, pero que javascript podrá entender.

Con esto, se tienen las dos funcionalidades originales, y ya sería posible empezar a dibujar en la interficie en Javascript, con múltiples usuarios al mismo tiempo, todo con persistencia en una base de datos, y todo con una simplicidad extrema.

1.3. Segundo Ciclo

asd