Desarrollo REST con Rails

Ralf Wirdemann
ralf.wirdemann@b-simple.de

Thomas Baustert
thomas.baustert@b-simple.de

traducción de **Juan Lupion**
juan.lupion@the-cocktail.com

17 de junio de 2007

2

ListingListado

Agradecimientos

Damos las gracias a Astrid Ritscher por las revisiones que hizo de la primera versión de este documento y a Adam Groves de Berlín por la revisión final de la versión en inglés de este documento.

Licencia

Ralf Wirdemann, Hamburgo en Febrero de 2007

Índice general

1. Rails y REST **1**

1.1. ¿Qué es REST? 2
1.2. ¿Por qué usar REST? 3
1.3. ¿Y qué hay de nuevo? 3
1.4. Preparativos 4
1.4.1. Rails 1.2 4
1.5. Andamiaje del Recurso 4
1.6. El modelo 5
1.7. El controlador 6
1.7.1. URLs REST 7
1.7.2. Las acciones REST utilizan respond_to 8
1.7.3. Campo Accept de la cabecera HTTP 9
1.7.4. Especificación del formato por la URL de la petición . . 10
1.8. URLs y vistas REST 11
1.8.1. New y Edit 13
1.8.2. Métodos de Path en los formularios: Create y Update . 13
1.8.3. Destroy 14
1.9. Métodos de URL en el controlador 15
1.10. Rutas REST 16
1.10.1. Convenciones 16
1.10.2. Ajustes 17
1.11. Recursos anidados 18
1.11.1. Adaptación de los controladores 20
1.11.2. Nuevos parámetros en los helpers de path y URL 20
1.11.3. Creando nuevas iteraciones 23
1.11.4. Edición de las iteraciones 25
1.12. Definición de otras acciones 26
1.12.1. ¿Seguro que no nos repetimos? 29
1.13. Definición de nuestros propios formatos 29
1.14. REST y AJAX 30
1.15. Pruebas del código 31
1.16. Clientes REST: ActiveResource 32
1.17. Conclusiones 34

Bibliografía **35**

Capítulo 1

Rails y REST

Es un hecho olvidado por muchos desarrolladores web de hoy que el protocolo HTTP puede hacer algo más que GETs y POSTs. Sin embargo, esto no resulta tan sorprendente si consideramos que los navegadores tan sólo soportan esas dos peticiones. GET y POST son los tipos de peticiones HTTP que se suelen transmitir del cliente a servidor, pero el protocolo HTTP también define los métodos PUT y DELETE que, en teoría, se deberían usar para crear o borrar un recurso en la web. En este tutorial ampliaremos nuestros horizontes y profundizaremos en los métodos PUT y DELETE.

Recientemente se ha hecho popular el término REST, que combina PUT y DELETE junto con GET y POST. Una de las nuevas características de Rails 1.2 es que soporta REST.

El tutorial comienza con una breve introducción de los conceptos y el ámbito de REST. A continuación se explicarán las razones que justifican el desarrollo de aplicaciones REST con Rails: usando andamiajes, mostraremos el desarrollo detallado de un controlador REST nos mostrará las herramientas técnicas que nos ayudarán en el desarrollo REST. Con esta base técnica en mente el siguiente capítulo mostrará la funcionalidad general y las modificaciones en las rutas de las cuales depende en gran medida la funcionalidad REST. El capítulo *Recursos anidados* presenta al lector como se pueden anidar los recursos en una relación de parentesco sin violar los conceptos de las URLs REST. Por ultimo, el tutorial termina con capítulos sobre REST y AJAX, la automatización de pruebas de aplicación REST y una breve introducción a ActiveResource, la parte de cliente de REST.

Antes de empezar, conviene aclarar que este tutorial asume que el lector tiene al menos un conocimiento básico del desarrollo con Rails. Si no es este el caso es mejor visitar alguno de los muchos tutoriales de Rails disponibles en la Red. (e.g., [3], [4], [5]) o leer algunos de los muchos libros en la materia que, por el momento, sólo se encuentran en inglés (p.e. [1] y [2]).

1.1. ¿Qué es REST?

El término REST apareció por primera vez en la tesis doctoral de Roy Fielding [6] y significa *Representational State Transfer*[1]. REST describe todo un paradigma de arquitectura para aplicaciones que solicitan y manipulan recursos en la web utilizando los métodos estándar de HTTP: GET, POST, PUT y DELETE.

En REST todo recurso es una entidad direccionable mediante una URL única con la que se puede interactuar a través del protocolo HTTP. Dichos recursos pueden representarse en diversos formatos como HTML, XML o RSS, según lo solicite el cliente. Al contrario que en las aplicaciones Rails tradicionales [2], la URL de un recurso no hace referencia a un modelo y su acción correspondiente; tan sólo hace referencia al propio recurso.

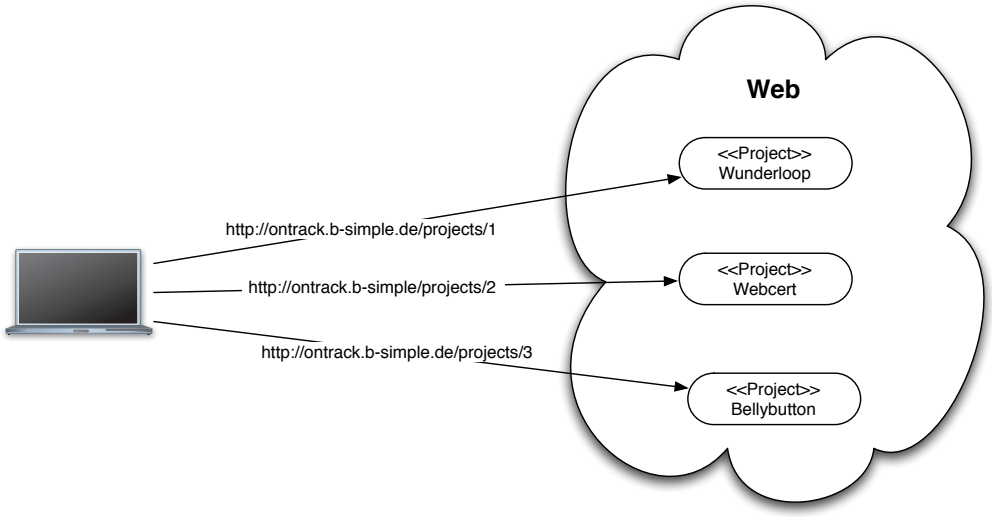

Figura 1.1: Recursos en la web y sus URLs

Los tres recursos de la figura 1.1 están representados por URLs casi idénticas seguidas por el identificador del recurso. Obsérvese que la URL no muestra qué se desea hacer con el recurso.

Un recurso, en el contexto de una aplicación Rails, es una combinación de un controlador dedicado y un modelo. Así, desde un punto de vista técnico, los recursos proyecto de la figura 1.1 son instancias de la clase ActiveRecord *Project* combinadas con un controlador *ProjectsController* que es reponsable de manipular dichas instancias.

[1] Transferencia de estado representacional

[2] Cuando queramos hacer una distinción entre aplicaciones Rails basadas o no en REST utilizaremos la palabra tradicional. Tradicional en este caso no quiere decir obsoleta o incorrecta, simplemente se utiliza para hacer referencia a un concepto no de REST. De esta manera debería ser más fácil explicar esta nueva tecnología.

1.2. ¿Por qué usar REST?

Esta es una cuestión interesante, si tenemos en cuenta que se vienen desarrollando con éxito aplicaciones Rails desde hace dos años utilizando el paradigma MVC. A continuación mostramos una lista de características de las aplicaciones REST, que servirá para mostrar las mejoras conceptuales que pueden hacerse en Rails.

URLs limpias. En REST las URLs representan recursos y no acciones, por lo tanto siempre tienen el mismo formato: primero aparece el controlador y luego el identificador de recurso. La manipulación requerida es independiente de la URL y se expresa con la ayuda de los verbos HTTP.

Formatos de respuesta variados. Los controladores REST están escritos de manera que las acciones pueden devolver sus resultados fácilmente en diferentes formatos de respuesta. Una misma acción puede entregar HTML, XML, RSS o cualquier otro formato de datos según los requisitos de la aplicación cliente de manera simple y sencilla.

Menos código. El desarrollar acciones únicas capaces de soportar múltiples clientes distintos evita repeticiones en el sentido DRY[3] y da como resultado que los controladores tengan menos código.

Controladores orientados a CRUD. Los controladores y los recursos se funden en una única cosa - cada controlador tiene como responsabilidad manipular un único tipo de recurso.

Diseño limpio de la aplicación. El desarrollo REST produce un diseño de aplicación conceptualmente claro y más fácil de mantener.

En los próximos capítulos de este tutorial veremos las características anteriores con la ayuda de varios ejemplos.

1.3. ¿Y qué hay de nuevo?

Aunque podría parecer que el diseño de aplicaciones basado en REST hace inútil toda la experiencia que hayamos atesorado en el desarrollo con Rails podemos asegurar que no es el caso: REST sigue estando basado en MVC y desde un punto de vista técnico se puede resumir en las siguientes técnicas nuevas:

- El uso de *respond_to* en el código del controlador.
- Nuevos métodos helper para enlaces y formularios.
- El uso de métodos de URL en las redirecciones desde el controlador.
- Nuevas rutas, generadas por el método *resources* en routes.rb.

Una vez que entendamos y comencemos a usar estas técnicas el diseño de aplicaciones REST se convertirá en algo natural.

[3] Don't repeat yourself

1.4. Preparativos

A continuación explicaremos las nuevas funcionalidades específicas de REST que incorpora Rails en el contexto de la aplicación de ejemplo de nuestro libro *Rapid Web Development mit Ruby on Rails* [1], *Ontrack*, una aplicación de gestión de proyectos. Por supuesto no vamos a desarrollar aquí toda la aplicación sino que utilizaremos la misma terminología para crear un entorno técnico sobre el que mostrar los conceptos REST.

Empezaremos generando nuestro proyecto Rails:

```
> rails ontrack
```

A continuación crearemos las bases de datos de test y desarrollo:

```
> mysql -u rails -p
Enter password: *****

mysql> create database ontrack_development;
mysql> create database ontrack_test;
mysql> quit
```

1.4.1. Rails 1.2

Asumiremos que no todos nuestros lectores querrán realizar una instalación de Rails 1.2 a nivel de sistema, dado que entonces todas sus aplicaciones utilizarán Rails 1.2 En su lugar vamos a instalar Rails 1.2 directamente sobre nuestro proyecto de manera que sólo nuestra aplicación de ejemplo *ontrack* utilizará esta nueva versión. La etiqueta de la última versión estable de Rails 1.2 según la documentación es *rel_1-2-2*. El siguiente comando rake instala la nueva versión sólo en nuestro proyecto:

```
> cd ontrack
> rake rails:freeze:edge TAG=rel_1-2-2
```

En la página principal de Rails *http://www.rubyonrails.org* pueden encontrarse otros métodos alternativos de instalación, así como información acerca de las nuevas versiones que puedan aparecer.

1.5. Andamiaje del Recurso

Empezaremos por crear el recurso *Project* con el nuevo generador *scaffold_resource*. Este generador toma un nombre de recurso (en nuestro caso *project*) como primer parámetro y, opcionalmente, una lista de atributos del modelo y sus tipos. Estos atributos se usarán para la generación de un script de migración de la base datos y para definir los campos apropiados en las vistas generadas:

```
> cd ontrack
> ruby script/generate scaffold_resource project \
>      name:string desc:text
exists  app/models/
    exists  app/controllers/
    exists  app/helpers/
    create  app/views/projects
    exists  test/functional/
    exists  test/unit/
    create  app/views/projects/index.rhtml
    create  app/views/projects/show.rhtml
    create  app/views/projects/new.rhtml
    create  app/views/projects/edit.rhtml
    create  app/views/layouts/projects.rhtml
    create  public/stylesheets/scaffold.css
    create  app/models/project.rb
    create  app/controllers/projects_controller.rb
    create  test/functional/projects_controller_test.rb
    create  app/helpers/projects_helper.rb
    create  test/unit/project_test.rb
    create  test/fixtures/projects.yml
    create  db/migrate
    create  db/migrate/001_create_projects.rb
     route  map.resources :projects
```

Además del modelo, el controlador y las vistas, el generador también crea un script de migración completamente funcional, así como una nueva entrada *map.resources :projects* en *config/routes.rb*, que será responsable del carácter REST de nuestro nuevo controlador. Pero antes de profundizar en esto haremos en primer lugar un recorrido paso a paso por todo lo que ha sido generado para nosotros.

1.6. El modelo

Tal y como mencionamos al principio los recursos REST son, en el contexto de Rails, la combinación de un controlador y un modelo. El modelo es una clase normal ActiveRecord que hereda de *ActiveRecord::Base*:

```
class Project < ActiveRecord::Base
end
```

Así que en el caso del modelo no hay novedades. Pero no debemos olvidarnos de generar la tabla de la base de datos:

```
> rake db:migrate
```

1.7. El controlador

ProjectsController es un controlador CRUD que manipula el recurso *Project*, lo que quiere decir que maneja exactamente un tipo de recurso y ofrece una acción específica para cada una de las cuatro operaciones CRUD [4]:

[4] El controlador tambien ofrece las acciones *index* para mostrar un listado de todos los recursos de este tipo,*new* para abrir el formulario de creación de nuevo recurso y *edit* para mostrar el formulario de edición.

Listado 1.1: ontrack/app/controllers/projects_controller.rb

```
class ProjectsController < ApplicationController
  # GET /projects
  # GET /projects.xml
  def index...

  # GET /projects/1
  # GET /projects/1.xml
  def show...

  # GET /projects/new
  def new...

  # GET /projects/1;edit
  def edit...

  # POST /projects
  # POST /projects.xml
  def create...

  # PUT /projects/1
  # PUT /projects/1.xml
  def update...
  end

  # DELETE /projects/1
  # DELETE /projects/1.xml
  def destroy...
end
```

No hay muchas novedades por aquí: en el controlador generado automáticamente hay acciones para la creación, recuperación, modificación y borrado de proyectos. Tanto el controlador como las acciones parecen normales a primera vista, pero todos tienen un comentario mostrando la petición URL a la que responderán con su verbo HTTP incluido. Se trata de URLs REST y las inspeccionaremos más a fondo en la siguiente sección.

1.7.1. URLs REST

Las URLs básicas en REST no se componen de nombre de controlador, acción e identificador del modelo (por ejemplo, */projects/show/1*) como estamos acostumbrados a ver en las aplicaciones tradicionales Rails. En lugar de esto tan sólo incluyen el nombre del controlador seguido del identificador del recurso a manipular:

```
/projects/1
```

Al haber desaparecido el nombre de la acción ya no resulta obvio determinar qué va a ocurrir con el recurso indicado. Por ejemplo, la URL mostrada anteriormente ¿debería mostrar o borrar el recurso? La respuesta a esta cues-

tión viene dada por el método HTTP [5] que se utilizará al solicitar la URL. La siguiente tabla presenta los cuatro verbos HTTP, sus URLs REST y cómo se corresponderían con acciones de controlador en las URLs tradicionales de Rails:

Cuadro 1.1: Verbos HTTP y URLs REST

Verbo HTTP	**URL REST**	**Acción**	**URL sin REST**
GET	/projects/1	show	GET /projects/show/1
DELETE	/projects/1	destroy	GET /projects/destroy/1
PUT	/projects/1	update	POST /projects/update/1
POST	/projects	create	POST /projects/create

Las URLs REST son idénticas para todas los verbos excepto en el caso de POST porque cuando creamos un nuevo proyecto aún no existe su identificador. Por lo general, la acción a ejecutar viene determinada por el verbo HTTP y las URLs hacen referencia a recursos en lugar de acciones.

Nota: Si introducimos la URL *http://localhost:3000/projects/1* en el navegador siempre se producirá una llamada a *show* porque cuando un navegador abre una dirección siempre utiliza el verbo GET. Dado que los navegadores sólo utilizan los verbos GET y POST, Rails tiene que hacer algunas triquiñuelas para habilitar las acciones *destroy* y *destroy*, en este caso proporcionando helpers especiales para la creación de links para borrar y crear recursos, de forma que el verbo DELETE se transmite al servidor en un campo oculto dentro de una petición POST (como se verá en la sección 1.8.3). El verbo PUT se envía de la misma forma para la creación de nuevos recursos (véase a sección 1.8.2).

1.7.2. Las acciones REST utilizan respond_to

Ya hemos visto que las acciones REST se activan mediante una combinación de URL de recurso y un verbo HTTP dando como resultado URLs limpias que sólo direccionan el recurso a manipular. Pero, ¿qué más se puede especificar en una URL REST?

Las acciones REST también pueden responder a diferentes tipos de clientes que esperan diferentes formatos de respuesta. Por supuesto el cliente más habitual para una aplicación web será una navegador pero también puede haber, por ejemplo, clientes de servicios web que suelen esperar la respuesta en XML o agregadores de feeds que prefieren el resultado en RSS o Atom.

El mecanismo principal para la generación del formato de respuesta solicitado por el cliente es el método *respond_to* que el generador de andamiajes coloca en las acciones CRUD. El siguiente fragmento de código muestra la acción *show* como ejemplo de uso de *respond_to*:

[5] en este tutorial utilizaremos el término *verbo* HTTP, porque expresa mejor que la petición dará como resultado una *acción*.

Listado 1.2: ontrack/app/controllers/projects_controller.rb

```
# GET /projects/1
# GET /projects/1.xml
def show
   @project = Project.find(params[:id])
   respond_to do |format|
      format.html # show.rhtml
      format.xml { render :xml => @project.to_xml }
   end
end
```

El método *respond_to* recibe un bloque con instrucciones específicas para cada formato. En nuestro ejemplo el bloque maneja dos formatos: HTML y XML, y según el formato solicitado por el cliente se ejecutan las instrucciones correspondientes. Si es HTML no ocurre nada especial, lo que produce como resultado la generación de la vista por defecto, *show.rhtml*, y si por el contrario el formato es XML el recurso será devuelto como XML.

El formato de la respuesta puede indicarse de dos maneras: o bien se coloca en el campo accept de la cabecera HTTP de la petición o se añade a la URL de la petición.

1.7.3. Campo Accept de la cabecera HTTP

Empezaremos viendo la produceimera técnica: utilizar el verbo HTTP en el campo accept de la cabecera. Es muy sencillo cambiar la cabecera HTTP con la herramienta de línea de comandos *curl* Pero antes de comprobarlo tenemos que arrancar el servidor web:

```
> ruby script/server webrick
=> Booting WEBrick...
=> Rails application started on http://0.0.0.0:3000
=> Ctrl-C to shutdown server; call with --help for options
[2006-12-30 18:10:50] INFO  WEBrick 1.3.1
[2006-12-30 18:10:50] INFO  ruby 1.8.4 (2005-12-24)
[2006-12-30 18:10:50] INFO  WEBrick::HTTPServer#start: port=3000
```

A continuación abriremos la página de proyectos de Ontrack, abriendo la siguiente URL: *http://localhost:3000/projects*, y crearemos uno o más proyectos (véase la figura 1.2).

Este invocación de curl solicita el recurso proyecto 1 en formato XML:

```
> curl -H "Accept: application/xml" \
    -i -X GET http://localhost:3000/projects/1
=>
HTTP/1.1 200 OK
Connection: close
Date: Sat, 30 Dec 2006 17:31:50 GMT
Set-Cookie: _session_id=4545eabd9d1bebde367ecbadf015bcc2; path=/
Status: 200 OK
Cache-Control: no-cache
Server: Mongrel 0.3.13.4
```

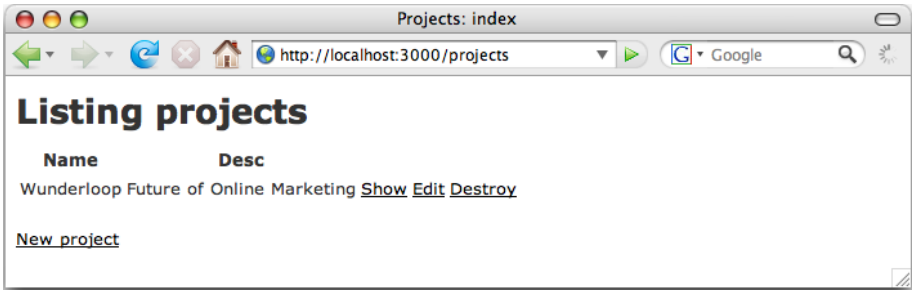

Figura 1.2: Crear algunos proyectos

```
Content-Type: application/xml; charset=utf-8
Content-Length: 160

<?xml version="1.0" encoding="UTF-8"?>
<project>
  <desc>Future of Online Marketing</desc>
  <id type="integer">1</id>
  <name>Wunderloop</name>
</project>
```

Rails enruta esta petición a la acción *show*. Como hemos preferido XML en el campo Accept de la cabecera HTTP nuestro método *respond_to* ejecuta el bloque correspondiente y el recurso accedido se convierte a XML en la respuesta.

De hecho curl no sólo nos sirve para probar los diferentes formatos de respuesta - también es muy útil para enviar verbos HTTP que no son soportados por los navegadores. La siguiente petición, por ejemplo, borrará el proyecto número 1:

```
> curl -X DELETE http://localhost:3000/projects/1
=>
<html>
<body>
You are being <a href="http://localhost:3000/projects">
   redirected</a>.
</body>
</html>
```

Esta vez la petición utiliza el verbo DELETE. El dispatcher de Rails evalúa el verbo y encamina la petición al método *ProjectsController.destroy*. Obsérvese que la URL es idéntica a la que se ha usado en la anterior petición con curl, siendo la única diferencia que esta vez se ha usado otro verbo HTTP.

1.7.4. Especificación del formato por la URL de la petición

La segunda manera de pedir un formato específico de respuesta es codificándolo en la URL. (Si borramos el último recurso de proyecto en el ejemplo previo tendremos que ir a la aplicación Ontrack y volver a crear otro ahora)

Para mostrar el proyecto 1 y para solicitar la respuesta en formato XML basta con abrir la siguiente dirección en el navegador:

```
http://localhost:3000/projects/1.xml
```

Figura 1.3: El proyecto Wunderloop en XML

Atención, usuarios de Mac: Esta petición es más fácil de comprobar en Firefox que en Safari porque este navegador simplemente ignora el XML devuelto. Firefox, por el contrario, formatea el XML (ver la figura 1.3).

Hasta ahora hemos aprendido cómo funcionan los controladores REST y qué aspecto tienen las URLs. En las dos próximas secciones veremos cómo usar y generar estas URLs en las vistas y controladores.

1.8. URLs y vistas REST

Las vistas representan la interfaz entre la aplicación y sus usuarios, que interactúan con ella utilizando enlaces y botones. Tradicionalmente en Rails se generaban los enlaces utilizando el helper *link_to*, que esperaba un hash conteniendo el controlado y la acción así como parámetros opcionales para la petición:

```
link_to :controller => "projects",
        :action => "show", :id => project
=>
<a href="/projects/show/1">Show</a>
```

Se ve inmediatamente que este uso tradicional de *link_to* no encaja muy bien con la nueva filosofía REST porque en ella se especifica que las URLs identifican al recurso y la acción se determina con el verbo HTTP utilizado. Lo que necesitamos es que los enlaces y botones pasen el verbo HTTP apropiado junto con la URL.

Rails propone la siguiente solución: se sigue usando el método *link_to* para generar el enlace, pero el hash es sustituido por una llamada a un metodo de *Path*. Los métodos de path crean enlaces de destino que *link_to* colocará en el atributo href del enlace generado. Como primer ejemplo crearemos un enlace a la acción *show* del controlador *ProjectsController*. En vez de especificar el controlador, acción e id de proyecto en un hash utilizaremos *project_path()*:

```
link_to "Show", project_path(project)
=>
<a href="/projects/1">Show</a>
```

El atributo href del método tradicional *link_to* está compuesto de un controlador, una acción y un identificador de proyecto. En su lugar, *project_path* construye un path REST que consiste sólo en el controlador y el identificador del proyecto. Como los enlaces usan el verbo HTTP GET por defecto, cuando el usuario haga clic en el enlace Rails reconoce que el proyecto en cuestión debería ser mostrado y se ejecutará la acción *show*.

Para cada tipo recurso en nuestra aplicación Rails generará automáticamente los siete métodos de path estándar que se muestran en la tabla 1.2.

Cuadro 1.2: Métodos de path

Método de path	**Verbo**	**Path**	**Acción solicitada**
projects_path	GET	/projects	index
project_path(1)	GET	/projects/1	show
new_project_path	GET	/projects/new	new
edit_project_path(1)	GET	/projects/1;edit	edit
projects_path	POST	/projects	create
project_path(1)	PUT	/projects/1	update
project_path(1)	DELETE	/projects/1	destroy

Cada método de path irá asociado con un verbo HTTP, lo que significa que este es el método HTTP que se enviará al servidor cuando se pulse sobre el enlace o botón. Algunas de las peticiones (show, update) serán transmitidas por defecto con el verbo HTTP adecuado (es decir, GET o POST). Otras (create, delete) tienen que ser tratadas de forma especial (utilizando campos ocultos) porque, como ya se ha mencionado, el navegador no utiliza los verbos PUT y DELETE. Aprenderemos más acerca de estos casos especiales y su implementación en las secciones 1.8.3 y 1.8.2.

Examinando la tabla puede verse que los cuatro verbos HTTP no son suficientes como para mapear todas las acciones CRUD. Los dos primeros métodos, *projects_path* y *project_path(1)*, funcionan bien con GET y pueden ser mapeados directamente a sus acciones apropiadas. Sin embargo, las cosas no están tan claras para *new_project_path* y *edit_project_path*.

1.8.1. New y Edit

Un clic sobre el enlace New se transmite al servidor usando el método GET. El siguiente ejemplo muestra que el path generado consiste en el controlador *ProjectsController* y la acción *new* que es la que se va a invocar:

```
link_to "New", new_project_path
=>
<a href="/projects/new">New</a>
```

¿Constitutye esto una fisura en la filosofía REST? A primera vista, tal vez. Pero, en perspectiva, resulta claro que *new* no es una acción REST/CRUD - no modifica ninguno de los datos de la aplicación, sino que más bien se trata de un preparativo para la creación de un nuevo recurso. La auténtica acción CRUD es *create*, que se ejecuta cuando se envía el formulario *new*. El identificador de recurso no aparece en la URL puesto que el recurso todavía no existe.

Lo mismo ocurre con el método *edit_project_path* - hace referencia a un recurso concreto, pero no es una acción CRUD. *edit_project_path* se emplea para preparar la verdadera llamada CRUD que se producirá cuando se envíe el método *update*. La diferencia entre *edit_project_path* y *new_project_path* es que *edit_project_path* necesita el id de proyecto a manipular. Siguiendo la convención REST, el id aparece después del controlador: */projects/1*. Sin embargo, si se enviase este path al servidor con un GET la llamada sería despachada a la acción *show* action. Para distinguir la acción edit, *edit_project_path* simplemente extiende el atributo href generado de una manera especial. Este es el aspecto del enlace HTML finalmente generado:

```
link_to "Edit", edit_project_path(project)
=>
<a href="/projects/1;edit">Edit</a>
```

No hay ningún problema en que *new* y *edit* incluyan la acción en la URL porque no son realmente acciones REST/CRUD de verdad. El mismo principio se utiliza para escribir acciones que utilicen otros nombres que no sean los nombres CRUD estándar (como veremos en la sección1.12).

1.8.2. Métodos de Path en los formularios: Create y Update

Tradicionalmente, los formularios se creaban usando los helpers *form_tag* o *form_for*, especificando la acción a invocar cuando se pulsa el botón de envío:

```
<% form_for :project, @project,
            :url => { :action => "create" } do |f| %>
...
<% end %>
```

En una aplicación REST el hash :url se rellena con el valor devuelto por alguno de los métodos de path:

- *project_path* for the *new* form
- *project_path(:id)* for the *edit* form

El nuevo formulario

El formulario se transmite al servidor mediante POST. La petición de *project_path* sin ningún id da como resultado el path /projects que cuando se envía usando POST provoca la ejecución de la acción *create*:

```
form_for(:project, :url => projects_path) do |f| ...
=>
<form action="/projects" method="post">
```

El formulario Edit

De acuerdo con la filosofía REST la modificación de un recurso se realiza mediante PUT pero, como ya sabemos, los navegadores no soportan ni PUT ni DELETE. La solución que ofrece Rails es utilizar una clave *method* en el hash *:html* de *form_for*:

```
form_for(:project, :url => project_path(@project),
         :html => { :method => :put }) do |f| ...
=>
<form action="/projects/1" method="post">
<div style="margin:0;padding:0">
  <input name="_method" type="hidden" value="put" />
</div>
```

Rails genera un campo oculto *_method* que contendrá el verbo HTTP apropiado (*put*). El dispatcher examinará este campo y pasará la petición a la acción *update*.

1.8.3. Destroy

Obsérverse que el método usado en los enlaces tanto para mostrar como para borrar un proyecto es *project_path*:

```
link_to "Show", project_path(project)
link_to "Destroy", project_path(project), :method => :delete
```

La única diferencia es que el enlace a destroy utiliza el parámetro *:method* para especificar el método HTTP a usar (en este caso :delete). Debido a que el navegador no soporta el verbo DELETE, Rails generará el siguiente fragmento de código Javascript que se ejecutará al hacer clic en el enlace:

```
link_to "Destroy", project_path(project), :method => :delete
=>
<a href="/projects/1"
```

```
onclick="var f = document.createElement('form');
f.style.display = 'none'; this.parentNode.appendChild(f);
f.method = 'POST'; f.action = this.href;
var m = document.createElement('input');
m.setAttribute('type', 'hidden');
m.setAttribute('name', '_method');
m.setAttribute('value', 'delete');
f.appendChild(m);
f.submit();
return false;">Destroy</a>
```

Este pequeño script crea un formulario al vuelo y garantiza que se envía el método DELETE en el campo oculto *_method*. Nuevamente, el dispatcher de Rails comprueba el contenido de este campo y sabe que la petición se debe encaminar hacia la acción *destroy*.

1.9. Métodos de URL en el controlador

Igual que los enlaces y acciones de envío de las vistas REST se crean con la ayuda de nuevos helpers, los controladores tienen que tener también cuidado cuando hagan redirecciones. Rails define para esto métodos de URL, existiendo un método URL para cada método de path:

`project_url` for `project_path`

o

`projects_url` for `projects_path`.

Al contrario que los métodos de path los métodos de URL crean URLs completas incluyendo el protocolo, el host, el puerto y la dirección:

```
project_url(1)
=>
"http://localhost:3000/projects/1"

projects_url
=>
"http://localhost:3000/projects"
```

Los métodos de URL se usan dentro de los controladores de una aplicación REST en los mismos lugares donde tradicionalmente se usaba *redirect_to* pasándole un hash con controlador/acción/parámetros. Por tanto, lo siguiente:

```
redirect_to :controller => "projects", :action => "show",
            :id => @project.id
```

en una aplicación REST se convierte en:

```
redirect_to project_url(@project)
```

Un ejemplo de esto lo podemos encontrar en el método *destroy*, donde se utiliza *projects_url* sin parámetro alguno para redirigir al listado de todos los proyectos después de que se haya borrado uno:

Listado 1.3: ontrack/app/controllers/projects_controller.rb

```
def destroy
  @project = Project.find(params[:id])
  @project.destroy

  respond_to do |format|
    format.html { redirect_to projects_url }
    format.xml  { head :ok }
  end
end
```

1.10. Rutas REST

Hasta ahora llevamos explicado el concepto REST así como unos cuantos métodos nuevos que hay que usar en los enlaces, formularios y controladores. Sin embargo, aún no hemos explicado de dónde surgen todos estos métodos: de su existencia y de la gestión adecuada de las peticiones REST es responsable una nueva entrada en el fichero de rutas de Rails, *config/routes.rb*:

```
map.resources :projects
```

Esta entrada ha sido puesta ahí por el generador *scaffold_resource*, y crea las rutas necesarias para solicitar las acciones REST de *ProjectsController*.

Además, *resources* genera los métodos de path y URL para el recurso *Project* con los que ya hemos estado experimentando en este capítulo:

```
map.resources :projects
=>
Ruta           Helper generado
------------------------------------------------------------
projects       projects_url, projects_path
project        project_url(id), project_path(id)
new_project    new_project_url, new_project_path
edit_project   edit_project_url(id), edit_project_path(id)
```

1.10.1. Convenciones

Como consecuencia de desarrollar empleando rutas REST, tenemos que cumplir con las convenciones de nombrado de las acciones CRUD. La siguiente llamada a *link_to* y el HTML generado lo muestran:

```
link_to "Show", project_path(project)
=>
<a href="/projects/1">Show</a>
```

Ni la llamada a *link_to* ni el código HTML generado incluyen el nombre de la acción que hay que invocar. El dispatcher de Rails sabe que la ruta */projects/:id* debe enviarse a la acción *show* de *ProjectsController* si la petición fue

enviada con el verbo GET, lo que implica que el controlador debe tener una acción con el nombre *show*. La misma convención se cumple para las acciones *index*, *update*, *create*, *destroy*, *new* y *edit* - todo controlador REST debe implementar estos métodos.

1.10.2. Ajustes

Las rutas REST pueden adaptarse a los requisitos específicos de nuestra aplicación con ayuda de las siguientes opciones:

- **:controller.** El controlador a usar
- **:path_prefix.** Prefijo de la URL (incluyendo cualquier variable que sea necesaria)
- **:name_prefix.** Prefijo de los helpers de ruta creado
- **:singular.** Forma singular del nombre de recurso

La siguiente entrada crea rutas para un nuevo recurso *Sprint*. Un sprint, en el contexto de gestión ágil de proyectos, es sinónimo de iteración y se mapea al modelo ActiveRecord *Iteration* (que presentaremos en el siguiente apartado):

```
map.resources :sprints,
              :controller => "ontrack",
              :path_prefix => "/ontrack/:project_id",
              :name_prefix => "ontrack_"
```

La opción *path_prefix* se utiliza para dar formato de la URL, en nuestro ejemplo, cada URL comienza con */ontrack* seguida del identificador de proyecto. También especificamos que el controlador responsable será *OntrackController*. Así, la URL

```
http://localhost:3000/ontrack/1/sprints
```

se encamina de acuerdo con el mapa anterior a la acción *index* de *OntrackController*. Otro ejemplo es la URL

```
http://localhost:3000/ontrack/1/sprints/1
```

que se encamina hacia la acción *show* del controlador *OntrackController*.

De la misma manera que *path_prefix* define el formato de la URL, *name_prefix* hace que los métodos helper generados empiecen por *ontrack_*. Por ejemplo:

```
ontrack_sprints_path(1)
=>
/ontrack/1/sprints
```

o

```
ontrack_edit_sprint_path(1, 1)
=>
/ontrack/1/sprints/1;edit
```

1.11. Recursos anidados

El desarrollo REST se pone realmente interesante cuando se emplean los denominados *recursos anidados*, donde se hace patente la importancia de las URLs limpias. Además los recursos anidados nos ayudarán a entender mejor los conceptos del paradigma REST.

Los recursos anidados son recursos fuertemente acoplados en el sentido de que muestran una relación de padres e hijos. En el contexto de Rails esto quiere decir que se trata de modelos que representan relaciones uno-a-muchos, tales como *Projects* y *Iterations* en Ontrack. Los controladores REST anidados siguen siendo responsables de manipular un único tipo de recurso, pero en el caso del controlador del hijo también leen el modelo del recurso padre.

El enfoque REST de Rails refleja la relación entre recursos anidados en sus URLs y mantiene el principio de URLs *limpias*. Describiremos este principio con el ejemplo de las iteraciones y proyectos en Ontrack. Por supuesto, hemos de empezar con la generación del recurso *Iteration* y la creación de la tabla *iterations* en la base de datos:

```
> ruby script/generate scaffold_resource iteration name:string \
       start:date end:date project_id:integer
> rake db:migrate
```

Los proyectos tienen una relación uno-a-muchos con respecto a las iteraciones. Esta relación se implementa en el modelo:

Listado 1.4: ontrack/app/models/project.rb

```
class Project < ActiveRecord::Base
  has_many :iterations
end
```

Listado 1.5: ontrack/app/models/iteration.rb

```
class Iteration < ActiveRecord::Base
  belongs_to :project
end
```

Además de crear modelo, controlador y vistas, el generador también crea una nueva ruta en *config/routes.rb*:

```
map.resources :iterations
```

Igual que con la linea para *projects*, esta sentencia genera las nuevas rutas y helpers para la manipulación del recurso *Iteration*. No obstante las iteraciones sólo tienen sentido en el contexto de un proyecto concreto - y esto es algo que las rutas y helpers recién creados no tienen en cuenta. Por ejemplo, el helper *new_iteration_path* devolverá el path */iterations/new*, que no aporta información sobre el proyecto al que pertenece la nueva iteración.

La idea de los recursos anidados consiste, en esencia, en tener en cuenta que un recurso hijo (en este caso, una iteración) no puede existir fuera del contexto de algún recurso padre al que pertenece (en este caso, un proyecto). Rails trata de reflejar esto en el uso de URLs y el controlador del recurso hijo. Y para que todo esto funcione se necesita cambiar la entrada generada del recurso en *config/routes.rb*:

```
map.resources :iterations
```

por:

```
map.resources :projects do |projects|
  projects.resources :iterations
end
```

Esta línea indica que *Iteration* es un recurso anidado y genera las rutas apropiadas que permiten manipular iteraciones únicamente en el contexto de un proyecto. Las rutas generadas tienen el siguiente formato:

```
/project/:project_id/iterations
/project/:project_id/iterations/:id
```

Por ejemplo, la URL

```
http://localhost:3000/projects/1/iterations
```

hace que se ejecute la acción *index* del controlador *IterationsController*, obteniendo el identificador del proyecto por medio del parámetro de la petición *:project_id*. Nótese cómo la URL claramente indica la asociación subyacente:

```
/projects/1/iterations <=> Project.find(1).iterations
```

Las URLs REST anidadas siguen siendo URLs limpias: hacen referencia a recursos, no acciones. El hecho de que un recurso esté anidado viene indicado por dos URLs REST que aparecen una detrás de otra en una única URL. Por ejemplo he aquí una petición a la acción *show* de una iteración concreta:

```
http://localhost:3000/projects/1/iterations/1
```

1.11.1. Adaptación de los controladores

El controlador generado *IterationsController* no sabe automáticamente que es responsable de un recurso anidado y que con cada petición también debe averiguar el identificador del proyecto padre. Por ejemplo, la acción *index* sigue cargando todas las iteraciones almacenadas aún cuando la URL claramente indica que sólo se deberían cargar todas las iteraciones de un proyecto específico.

Listado 1.6: ontrack/app/controllers/iterations_controller.rb

```
def index
  @iterations = Iteration.find(:all)

  respond_to do |format|
    format.html # index.rhtml
    format.xml  { render :xml => @iterations.to_xml }
  end
end
```

Es necesario cambiar la acción para que sólo se carguen las iteraciones del proyecto correspondiente:

Listado 1.7: ontrack/app/controllers/iterations_controller.rb

```
def index
  project = Project.find(params[:project_id])
  @iterations = project.iterations.find(:all)
  ...
end
```

Dado que toda acción del controlador *IterationsController* sólo puede funcionar adecuadamente con un prefijo */projects/:project_id*: es necesario que quede definido el proyecto padre que define el ámbito de la iteración o iteraciones que queremos manipular. Pero esto quiere decir también que hay que reescribir no sólo la acción *index* sino también las acciones *create* (véanse las secciones *create* 1.11.3) y *update* (sección 1.11.4).

1.11.2. Nuevos parámetros en los helpers de path y URL

La entrada del recurso en *config/routes.rb* para las iteraciones no sólo genera nuevas rutas sino también nuevos helpers. Y como en el caso de las rutas estos helpers esperan un identificador de proyecto como primer parámetro. Por ejemplo el path que denota el listado de todas las iteraciones de un proyecto se genera con el helper *iterations_path*. Los nombres de los helpers para recursos anidados son idénticos a los nombres de los helpers para los recursos no anidados, lo único que cambia es el número de parámetros que requieren los helpers. Los helpers de los recursos anidados siempre esperarán el identificador del recurso contenedor como primer parámetro, en este caso el identificador del proyecto. Aquí, por ejemplo, el enlace *iterations* (que aparece en la vista

index de *ProjectsController*) muestra todas las iteraciones del proyecto escogido. Aquí la URL para el listado de iteraciones se genera por el helper *iterations_path*, que espera el parámetro con el identificador del proyecto:

```
link_to "Iterations", iterations_path(project)
=>
<a href="/projects/1/iterations">Iterations</a>
```

Para entenderlo mejor miraremos el enlace donde realmente se usa, en la vista *ProjectsController*:

Listado 1.8: ontrack/app/views/projects/index.rhtml

```
...
<% for project in @projects %>
 <tr>
  <td><%=h project.name %></td>
  <td><%=h project.desc %></td>
  <td><%= link_to "Iterations", iterations_path(project) %></td>
  <td><%= link_to "Show", project_path(project) %></td>
  <td><%= link_to "Edit", edit_project_path(project) %></td>
  <td><%= link_to "Destroy", project_path(project),
       :confirm => "Are you sure?", :method => :delete %></td>
 </tr>
<% end %>
...
```

Consecuencia del nuevo parámetro no es que sólo se han roto algunas acciones del controlador, sino también muchas de las vistas generadas automáticamente para las iteraciones. Por ejemplo, la vista index contiene una tabla con todas las iteraciones y cada iteración tiene tres enlaces:

Listado 1.9: ontrack/app/views/iterations/index.rhtml

```
...
<% for iteration in @iterations %>
<tr>
 <td><%=h iteration.name %></td>
 <td><%=h iteration.start %></td>
 <td><%=h iteration.end %></td>
 <td><%= link_to "Show", iteration_path(iteration) %></td>
 <td><%= link_to "Edit", edit_iteration_path(iteration) %></td>
 <td><%= link_to "Destroy", iteration_path(iteration),
    :confirm => "Are you sure?",
    :method => :delete %></td>
  </tr>
<% end %>
...
```

Todos los enlaces comienzan con el id de la iteración como primer y único parámetro. Esto ya no funciona porque el primer parámetro de un helper para las iteraciones debería ser el identificador de proyecto. El cambio necesario tiene el siguiente aspecto:

Listado 1.10: ontrack/app/views/projects/index.rhtml

```
...
<% for iteration in @iterations %>
<tr>
 <td><%=h iteration.name %></td>
 <td><%=h iteration.start %></td>
 <td><%=h iteration.end %></td>
 <td><%= link_to "Show",
       iteration_path(iteration.project, iteration) %></td>
 <td><%= link_to "Edit",
       edit_iteration_path(iteration.project, iteration) %></td>
```

```
 <td><%= link_to "Destroy",
       iteration_path(iteration.project, iteration),
       :confirm => "Are you sure?",
       :method => :delete %></td>
</tr>
<% end %>
...
```

Una manera alternativa de corregir la lista de parámetros de los helpers de los recursos anidados es pasar los identificadores necesarios como un hash:

```
iteration_path(:project_id => iteration.project, :id => iteration)
```

lo cual hace el código más legible cuando no es evidente con qué tipo de objeto está relacionada la iteración.

1.11.3. Creando nuevas iteraciones

Otra acción que sólo funciona en el contexto de un proyecto previamente escogido es la de añadir nuevas iteraciones. Para gestionar esto, solamente tenemos que añadir un enlace *New Iteration* a la vista *index* de *ProjectsController*:

Listado 1.11: ontrack/app/views/projects/index.rhtml

```
...
<% for project in @projects %>
<tr>
 <td><%=h project.name %></td>
 <td><%=h project.desc %></td>
 <td><%= link_to "Iterations", iterations_path(project) %></td>
 <td><%= link_to "Show", project_path(project) %></td>
 <td><%= link_to "Edit", edit_project_path(project) %></td>
 <td><%= link_to "Destroy", project_path(project),
         :confirm => "Are you sure?",
         :method => :delete %></td>
 <td><%= link_to "New Iteration",
         new_iteration_path(project) %></td>
</tr>
<% end %>
...
```

Para el método de path utilizaremos *new_iteration_path* que genera para el proyecto con id 1 el siguiente HTML:

```
link_to "New Iteration", new_iteration_path(project)
=>
<a href="/projects/1/iterations/new">New iteration</a>
```

El enlace lleva a la acción *new* de *IterationsController*. La acción recibe el valor 1 por medio del parámetro *project_id*, que es el identificador del proyecto

actual. De esta manera el identificador del proyecto está disponible en la vista *new* de *IterationsController* y puede ser usado allí por el helper *iterations_path* responsable de la creación del formulario para la nueva iteración. El formulario generado contiene una ruta anidada en el atributo *action* que contiene el id del proyecto para el que se debería crear la nueva iteración:

Listado 1.12: ontrack/app/views/iterations/new.rhtml

```
<% form_for(
          :iteration,
      :url => iterations_path(params[:project_id])) do |f| %>
...
<% end %>
=>
<form action="/projects/1/iterations" method="post">
```

El uso de *params[:project_id]* en *iterations_path* es opcional porque Rails automáticamente pone el parámetro *project_id* como el atributo *action*, lo que quiere decir que

```
form_for(:iteration, :url => iterations_path)
```

tiene el mismo efecto.

El encaminamiento REST asegura que la acción del formulario */projects/1/iterations*, en combinación del verbo HTTP POST resulta en la ejecución de la acción *create* action de *IterationsController*. El método HTTP (method='post') de la etiqueta form fue creado por el helper por defecto, dado que no se especificó ningún verbo HTTP de manera explícita y *post* es el valor por defecto.

Además de los parámetros del formulario la acción create obtiene el identificador del proyecto por medio del parámetro *project_id* de la petición, por consiguiente hay que cambiar el método de forma que la iteración recién creada se asigne al proyecto correcto:

Listado 1.13: ontrack/app/controllers/iterations_controller.rb

```
1  def create
2    @iteration = Iteration.new(params[:iteration])
3    @iteration.project = Project.find(params[:project_id])
4
5    respond_to do |format|
6     if @iteration.save
7      flash[:notice] = "Iteration was successfully created."
8      format.html {
9       redirect_to iteration_url(@iteration.project,
10                                      @iteration)}
11     format.xml  {
12          head :created,
13               :location => iteration_url(@iteration.project,
14                                                  @iteration)}
15    else
16     format.html { render :action => "new" }
```

```
17      format.xml  { render :xml => @iteration.errors.to_xml }
18    end
19  end
20 end
```

En la linea 3 se asigna el proyecto explícitamente. También se le ha añadido el identificador de proyecto al helper *iteration_url* en las líneas 8 y 11.

Para que el añadir nuevas iteraciones funcione hay que extender los enlace *Edit* y *Back* de la vista *show* del *IterationsController* para incluir el identificador del proyecto.

Listado 1.14: ontrack/app/views/iterations/show.rhtml

```
...
<%= link_to "Edit", edit_iteration_path(@iteration.project,
                                        @iteration) %>

<%= link_to "Back", iterations_path(@iteration.project) %>
```

Esta vista se genera después de la creación de una nueva iteración y lanzaría una excepción si no le pasásemos el identificador del proyecto.

1.11.4. Edición de las iteraciones

Para editar iteraciones, hay que hacer dos cambios sobre el código generado.

El helper *form_for* en la vista *edit* del controlador *IterationsController* sólo recibe el id de la iteración:

```
form_for(:iteration,
         :url => iteration_path(@iteration),
         :html => { :method => :put }) do |f|
```

sin embargo, hacen falta tanto el identificador de proyecto como el de la iteración:

```
form_for(
 :iteration,
 :url => iteration_path(params[:project_id], @iteration),
 :html => { :method => :put }) do |f|
```

Es necesario un cambio similar en la acción *update* que se llama desde el formulario – el método *iteration_url* de la línea 7 sólo recibe el identificador de la iteración:

Listado 1.15: ontrack/app/controllers/iterations_controller.rb

```
1 def update
2   @iteration = Iteration.find(params[:id])
3
4   respond_to do |format|
```

```
5       if @iteration.update_attributes(params[:iteration])
6         flash[:notice] = "Iteration was successfully updated."
7         format.html { redirect_to iteration_url(@iteration) }
8         format.xml  { head :ok }
9       else
10        format.html { render :action => "edit" }
11        format.xml  { render :xml => @iteration.errors.to_xml }
12      end
13    end
14  end
```

Igual que antes, hay que cambiar la línea 7:

```
format.html { redirect_to iteration_url(@iteration.project,
                                        @iteration) }
```

Tras todos estos cambios las vistas de create y update y sus acciones deberían por fin funcionar. Pueden crearse y editarse las iteraciones pero para estar absolutamente seguros hemos de examinar *IterationsController* y sus vistas respectivas comprobando todos los helpers de path y URL buscando los que no reciban un identificador de proyecto y cambiándolo como hemos hecho con las vistas anteriores.

1.12. Definición de otras acciones

La entrada *resources* del fichero de rutas genera rutas y helpers para las acciones CRUD. ¿Pero cómo se pueden craer rutas y helpers para otras acciones que también pertenezcan al controlador? Por ejemplo veamos la nueva acción *close* de *ProjectsController*. Esta acción se va a utilizar para cerrar un proyecto, es decir, para marcarlo como finalizado.

Empezaremos on la migración de la base de datos:

```
> ruby script/generate migration add_closed_to_projects
    exists db/migrate
    create db/migrate/003_add_closed_to_projects.rb
```

Listado 1.16: ontrack/db/migrate/003_add_closed_to_projects.rb

```
class AddClosedToProjects < ActiveRecord::Migration
  def self.up
    add_column :projects, :closed, :boolean, :default => false
  end

  def self.down
    remove_column :projects, :closed
  end
end
```

```
rake db:migrate
```

A continuación crearemos un enlace *close* en la vista *index* de *ProjectsController*:

Listado 1.17: ontrack/app/views/projects/index.rhtml

```
<% for project in @projects %>
  <tr id="project_<%= project.id %>">
    <td><%=h project.name %></td>
    <td><%= link_to "Show", project_path(project) %></td>
    ...
    <td><%= link_to "Close", <WHICH_HELPER?> %></td>
  </tr>
<% end %>
```

Surgen dos cuestiones a la hora de añadir este enlace:

1. ¿Qué método HTTP debería usarse para enviar la acción *close*?

2. ¿Cómo se puede generar un helper que cree la ruta a la acción *close*?

Dado que *close* no es una acción CRUD estándar, Rails por defecto no sabe qué verbo HTTP debería usarse. Close es una acción especializada del proyecto, una especie de actualización del registro de un proyecto, así que conforme a los principios REST debería ser enviada con POST.

Definiremos la ruta y el helper correspondiente en *config/routes.rb* con la ayuda del hash *member* en la llamada a *resources* para *projects*. Este hash consiste en pares método-acción y especifica qué acción debe o puede llamarse con cada verbo HTTP [6].

Los valores permitidos son *:get*, *:put*, *:post*, *:delete* y *:any*. Si una acción aparece marcada con *:any*, significa que puede llamarse a la acción con cualquier verbo HTTP. En nuestro ejemplo *close* debería solicitarse via POST, así que cambiaremos la entrada en *resources* tal y como se muestra a continuación:

```
map.resources :projects, :member => { :close => :post }
```

Tras escribir esta entrada ya podemos usar el nuevo helper *close_project_path* para crear el enlace a Close que mencionábamos antes:

```
<td><%= link_to "Close", close_project_path(project) %></td>
```

Sin embargo cuando pulsemos el enlace obtendremos un error:

```
no route found to match "/projects/1;close" with {:method=>:get}
```

[6] Rails protege estas rutas con restricciones HTTP, de forma que si se solicita una acción con el verbo incorrecto se lanza una excepción RoutingError.

La ruta existe, pero la nueva entrada en *resources* sólo permite las peticiones via POST. Rails rechaza los otros verbos HTTP (como GET, que es el que se ha usado en el enlace anterior). Lo que necesitamos es algo parecido al enlace *destroy*: un helper que genere un formulario que se envíe vía POST al servidor. Por suerte, Rails ofrece el helper *button_to* para hacer exactamente esto.

```
<td><%= button_to "Close", close_project_path(project) %></td>
=>
<td>
  <form method="post" action="/projects/1;close"
        class="button-to">
    <div><input type="submit" value="Close" /></div>
  </form>
</td>
```

Lo único que falta es la acción *close* en *ProjectsController*:

Listado 1.18: ontrack/app/controllers/projects_controller.rb

```
def close
  respond_to do |format|
    if Project.find(params[:id]).update_attribute(:closed, true)
      flash[:notice] = "Project was successfully closed."
      format.html { redirect_to projects_path }
      format.xml  { head :ok }
    else
      flash[:notice] = "Error while closing project."
      format.html { redirect_to projects_path }
      format.xml  { head 500 }
    end
  end
end
```

En la llamada a *resources* además de *:member* pueden especificarse las claves *:collection* y *:new*. *:collection* es necesaria cuando, en lugar de sobre un único recurso, la acción sea realizada una colección de recursos de ese tipo particular. Un ejemplo es la solicitud de un listado de proyectos como un feed RSS:

```
map.resources :projects, :collection => { :rss => :get }
--> GET /projects;rss (maps onto the #rss action)
```

La clave *:new* se utiliza con acciones que actúan sobre recursos nuevos que aún no han sido almacenados:

```
map.resources :projects, :new => { :validate => :post }
--> POST /projects/new;validate (maps onto the #validate action)
```

1.12.1. ¿Seguro que no nos repetimos?

El párrafo anterior suena como una violación del principio de no repetición: las acciones ya no están solamente implementadas en el controlador, sino que también aparecen en el fichero de rutas. Como alternativa a los patrones REST descritos arriba se pueden invocar las acciones no REST de la manera tradicional utilizando la acción y el identificador de proyecto:

```
<%= link_to "Close", :action => "close", :id => project %>
```

Las rutas necesarias para esto deberían seguir estando definidas siempre y cuando no se haya borrado la entrada *map.connect ':controller/:action/:id'* en el fichero de rutas. Pero la ruta antigua sólo funcionará si no se ha modificado la llamada a *resources* para *projects* como se describió anteriormente.

1.13. Definición de nuestros propios formatos

El método *respond_to* por defecto sólo reconoce los siguientes formatos:

```
respond_to do |wants|
  wants.text
  wants.html
  wants.js
  wants.ics
  wants.xml
  wants.rss
  wants.atom
  wants.yaml
end
```

Para extender esto podemos registrar nuestros propios formatos como tipos MIME. Supongamos que hemos desarrollado una aplicación PIM y queremos devolver las direcciones por medio del microformato vcard format[7]. Para hacerlo primero tenemos que registrar el nuevo formato en fichero de configuración *config/environment.rb*:

```
Mime::Type.register "application/vcard", :vcard
```

Ahora podemos ampliar la acción *show* del controlador *AddressesController* de forma que pueda devolver direcciones el formato vcard si el cliente lo solicita:

```
def show
  @address = Address.find(params[:id])

  respond_to do |format|
    format.vcard { render :xml => @address.to_vcard }
    ...
```

[7] http://microformats.org/wiki/hcard

```
  end
end
```

Por supuesto, el método *to_vcard* no es un método estándar de ActiveRecord y deberíamos implementarlo cumpliendo la especificación vcard (RFC2426) Si se implementa correctamente, la llamada a la siguiente URL debería devolver una dirección en formato XML conforme a vcard:

```
http://localhost:3000/addresses/1.vcard
```

1.14. REST y AJAX

En cuanto al desarrollo de aplicaciones AJAX basadas en REST no hay muchas novedades. Se pueden usar los helpers asíncronos ya conocidos y pasar al parámetro *:url* el método de path en lugar de un hash con controlador y acción. El siguiente fragmento de código convierte el enlace *destroy* de la vista *index* en *ProjectsController* en un enlace AJAX:

```
link_to_remote "Destroy", :url => project_path(project),
                          :method => :delete
=>
<a href="#" onclick="new Ajax.Request("/projects/1",
  {asynchronous:true, evalScripts:true, method:"delete"});
  return false;">Async Destroy</a>
```

Consejo: inclúyanse las librerías Javascript necesarias si no se desea pasar un cuarto de hora averiguando por qué no funciona el enlace. Una forma de hacerlo es llamar a *javascript_include_tag* en el fichero del layout *projects.rhtml* del proyecto *ProjectsController*:

Listado 1.19: ontrack/app/views/layouts/projects.rhtml

```
<head>
  <%= javascript_include_tag :defaults %>
  ...
</head>
```

El enlace lleva a la acción *destroy* de *ProjectsController*. Desde el punto de vista de la lógica de negocio el método ya lo hace todo correctamente: elimina el proyecto escogido. Lo que falta es una entrada adicional en el bloque *respond_to* para devolver al cliente el nuevo formato solicitado, en este caso Javascript. El siguiente fragmento de código muestra la nueva versión de *destroy*:

Listado 1.20: ontrack/app/controllers/projects_controller.rb

```
def destroy
   @project = Project.find(params[:id])
```

```
    @project.destroy

    respond_to do |format|
      format.html { redirect_to projects_url }
      format.js   # default template destroy.rjs
      format.xml  { head :ok }
    end
  end
```

El único cambio es la nueva entrada *format.js* en el bloque *respond_to*. Dado que la nueva entrada no tiene bloque asociado para ejecutar Rails actúa de la manera estándar y procesa una plantilla RJS de nombre *destroy.rjs*:

Listado 1.21: ontrack/app/views/projects/destroy.rjs

```
page.remove "project_#{@project.id}"
```

La plantilla anterior elimina el elemento con el id *project_ID* del árbol DOM del navegador. Para que esto funcione en la vista *index* de *ProjectsController* hay que añadir un identificador único a cada fila:

Listado 1.22: ontrack/app/views/projects/index.rhtml

```
id="project_<%= project.id %>">
```

Esta extensión de *ProjectsController* sirve como ejemplo de cómo las aplicaciones REST cumplen con el principio DRY y requieren menor cantidad de código. ¡Una única línea en el controlador hace que la misma acción sea capaz de manejar peticiones Javascript!

Además el ejemplo muestra una regla general para el desarrollo de controladores REST: implementar la lógica fuera del bloque *respond_to* lleva a menos repeticiones en el código resultante.

1.15. Pruebas del código

No importa cómo de excitante sea el desarrollo REST con Rails, las pruebas de código deberían dejarse de lado los tests. Que hemos estado desarrollando demasiado sin correr nuestros test unitarios será patente cuando finalmente ejecutemos los tests con *rake*[8]:

```
> rake
...
Started
EEEEEEE.......
```

[8]Atención, si aún no se ha hecho habría que crear la base de datos *ontrack_test* para pruebas

Las buenas noticias son que todas las pruebas unitarias y la prueba funcional *ProjectsControllerTest* siguen funcionando. Las malas noticias: las sietes pruebas de *IterationsControllerTest* están rotas.

Si todos los tests de un único caso de prueba fallan es un claro síntoma de que algo fundamental anda mal. En nuestro caso el error es obvio: el caso de prueba fue creado por el generador de andamiaje para iteraciones sin un proyecto padre. Extendimos las iteraciones para hacerlas pertenecer a un proyecto cuando añadimos toda la funcionalidad en *IterationsController* y ahora todas las acciones esperan el parámetro añadido *:project_id*. Para arreglarlo extenderemos el hash de la petición en todos los métodos de prueba con el parámetro *project_id*. Por ejemplo, veamos el test *test_should_get_edit*:

Listado 1.23: ontrack/test/functional/iterations_controller_test.rb

```
def test_should_get_edit
  get :edit, :id => 1, :project_id => projects(:one)
  assert_response :success
end
```

Adicionalmente, el controlador *IterationsControllerTest* debe también crear los datos de test de *projects*:

```
fixtures :iterations, :projects
```

Tras estos cambios sólo nos quedarán por pasar los tests *test_should_create_iteration* y *test_should_update_iteration*. En ambos casos la razón es que hay un *assert_redirected_to* equivocado:

```
assert_redirected_to iteration_path(assigns(:iteration))
```

Lo que pasa es obvio: hemos cambiado todas las redirecciones en *IterationsController* de forma que el primer parámetro es el identificador de proyecto. El aserto sólo comprueba si hay un identificador de iteración en la redirección. En este caso, el controlador está bien y tenemos que adaptar el test:

```
assert_redirected_to iteration_path(projects(:one),
                                    assigns(:iteration))
```

Por cierto: el uso de métodos de path en los asertos de redirección es la única diferencia entre las pruebas funcionales REST y las pruebas funcionales tradicionales.

1.16. Clientes REST: ActiveResource

Suele mencionarse con frecuencia ActiveResource junto con REST. ActiveResource es una librería Rails para el desarrollo de clientes de servicio web basados en REST, tales clientes utilizan los cuatro verbos HTTP típicos para comunicarse con el servidor.

ActiveResource no es parte de Rails 1.2 pero está disponible en la rama de desarrollo y puede instalarse usando svn[9]:

```
> cd ontrack/vendor
> mv rails rails-1.2
> svn co http://dev.rubyonrails.org/svn/rails/trunk rails
```

ActiveResource abstrae los recursos web desde el punto de vista del cliente como clases que heredan de *ActiveResource::Base*. Por ejemplo modelaremos el recurso *Project* (con el que trabajabamos en el lado del servidor anteriormente) como sigue:

```
require "activeresource/lib/active_resource"

class Project < ActiveResource::Base
  self.site = "http://localhost:3000"
end
```

Se importa de manera explícita la librería ActiveResource. Adicionalmente se especifica la URL del servicio en la variable de clase *site*. La clase *Project* abstrae la parte de cliente del servicio tan bien que el programador tiene la impresión que está trabajando con una clase ActiveRecord normal. Por ejemplo este método *find* solicita un recurso con un identificador dado al servidor:

```
wunderloop = Project.find 1
puts wunderloop.name
```

La llamada a *find* ejecuta una petición REST con el verbo GET:

```
GET /projects/1.xml
```

El servidor devuelve la respuesta en XML. El cliente genera, a partir del XML, un objeto ActiveResource *wunderloop* que ofrece como un modelo ActiveRecord métodos para acceder a todos sus atributos. Pero, ¿cómo se hace para modificar un recurso?

```
wunderloop.name = "Wunderloop Connect"
wunderloop.save
```

La llamada a *save* convierte el recurso a XML y lo envía por medio de PUT al servidor:

```
PUT /projects/1.xml
```

Si cogemos el navegador web y recargamos el listado de proyectos, veremos que el proyecto modificado debería aparecer con su nuevo nombre.

Crear nuevos recursos es tan fácil como recuperarlos y modificarlos:

[9] http://subversion.tigris.org/

```
bellybutton = Project.new(:name => "Bellybutton")
bellybutton.save
```

El nuevo proyecto se transmite al servidor en XML por medio de POST y se almacena en la base de datos:

```
POST /projects.xml
```

Si ve vuelve a cargar el listado de proyectos en el navegador aparecerá el proyecto recién creado. La última de las cuatro operaciones CRUD que tenemos que repasar es el borrado de proyectos:

```
bellybutton.destroy
```

La invocación de *destroy* se transmite con DELETE y hace que se borre el proyecto en el servidor:

```
DELETE /projects/2.xml
```

ActiveResource utiliza los cuatro verbos HTTP al estilo REST, ofreciendo una abstracción muy buena de los recursos REST en el lado del cliente. Además muchos de los métodos ya conocidos de ActiveRecord funcionan con ActiveResource, tales como encontrar las instancias de un recurso:

```
Project.find(:all).each do |p|
  puts p.name
end
```

Creemos que ActiveResource proporciona una muy buena base para el desarrollo de sistemas débilmente acoplados en Ruby. Es una buena idea echar un vistazo en la rama trunk y experimentar con las clases básicas de ActiveResource.

1.17. Conclusiones

No hay por qué aplicar REST a rajatabla: las soluciones híbridas son más fáciles de implementar, y por lo general uno está en mitad de un proyecto cuando aparecen nuevas funcionalidades en Rails. En este caso no es ningún problema desarrollar modelos REST sencillos y sus controladores para adquirir experiencia. Por el contrario, si se empieza una aplicación desde cero es mejor plantearse desarrollarla en torno a REST desde el principio. Las ventajas son claras: una arquitectura limpia, menos código y capacidad multicliente.

Bibliografía

[1] *Ralf Wirdemann, Thomas Baustert*: Rapid Web Development mit Ruby on Rails, 2. Auflage, Hanser, 2007

[2] *Dave Thomas, David Heinemeier Hansson: Agile Web Development with Rails*, Second Edition, Pragmatic Bookshelf, 2006

[3] *Curt Hibbs*: Rolling with Ruby on Rails – Part 1, http://www.onlamp.com/pub/a/onlamp/2005/01/20/rails.html

[4] *Curt Hibbs*: Rolling with Ruby on Rails – Part 2, http://www.onlamp.com/pub/a/onlamp/2005/03/03/rails.html

[5] *Amy Hoy*: Really Getting Started in Rails, http://www.slash7.com/articles/2005/01/24/really-getting-started-in-rails.html

[6] *Roy Fielding*: Architectural Styles and the Design of Network-based Software Architectures, http://www.ics.uci.edu/ fielding/pubs/dissertation/top.htm