RESTful Rails Development

Ralf Wirdemann
ralf.wirdemann@b-simple.de

Thomas Baustert
thomas.baustert@b-simple.de

translated by **Florian Görsdorf** and **Ed Ruder**

March 26, 2007

Acknowledgments

Many thanks go out to: Astrid Ritscher for her reviews of the first german version of this document; to Adam Groves from Berlin for his final review of the english version of this document.

License

Ralf Wirdemann, Hamburg in March 2007

Contents

Chapter 1

RESTful Rails

HTTP can do more then GET and POST, a fact that is almost forgotten by many web developers these days. But if you consider that browsers only support GET and POST requests, maybe this shouldn't be a surprise.

GET and POST are two of the types of HTTP requests that are often transmitted from clients to servers. The HTTP protocol also defines PUT and DELETE methods which are used to tell the server to create or delete a web resource.

This tutorial's aim is to broaden developers' horizons about the HTTP methods PUT and DELETE. One use of PUT and DELETE together with GET and POST that has become popular recently is referred to by the common term "REST". One of the highlighted new features of Rails 1.2 is its support of REST.

The tutorial starts with a short introduction into the concepts and the background of REST. Building on this, the reasons for developing RESTful Rails applications are explained. Using scaffolding, a detailed development of a REST controller and its model shows us the technical tools that help us with RESTful development. With this technical base in mind, the next chapter shows the general functionality and customization of so-called REST Routing, on which REST functionality is heavily dependent. The chapter *Nested Resources* introduces the reader to the advanced school of REST development and explains how resources can be nested in a parent-child relationship without violating the concept of REST URLs. The tutorial ends with chapters about REST and AJAX, testing of RESTful applications and an introduction to ActiveResource–the client-side part of REST.

Before we start, one last word: this tutorial assumes that you have at least a basic knowledge of Rails development. If this is not the case, please consider working through one of the many Rails tutorials available on the Internet (e.g., [3], [4], [5]), or read one of many books on the subject (e.g., [1] and [2]).

1.1 What is REST?

The term REST was coined by Roy Fielding in his Ph.D. dissertation [6] and means *Representational State Transfer*. REST describes an architecture paradigm for web applications that request and manipulate web resources using the standard HTTP methods GET, POST, PUT and DELETE.

A resource in the context of REST is a URL-addressable entity that offers interaction via HTTP. Resources can be represented in different formats like HTML, XML or RSS, depending on what the client requests. REST URLs are unique. Unlike a traditional Rails application[1], a resource URL does not address a model and its corresponding action; it addresses only the resource itself.

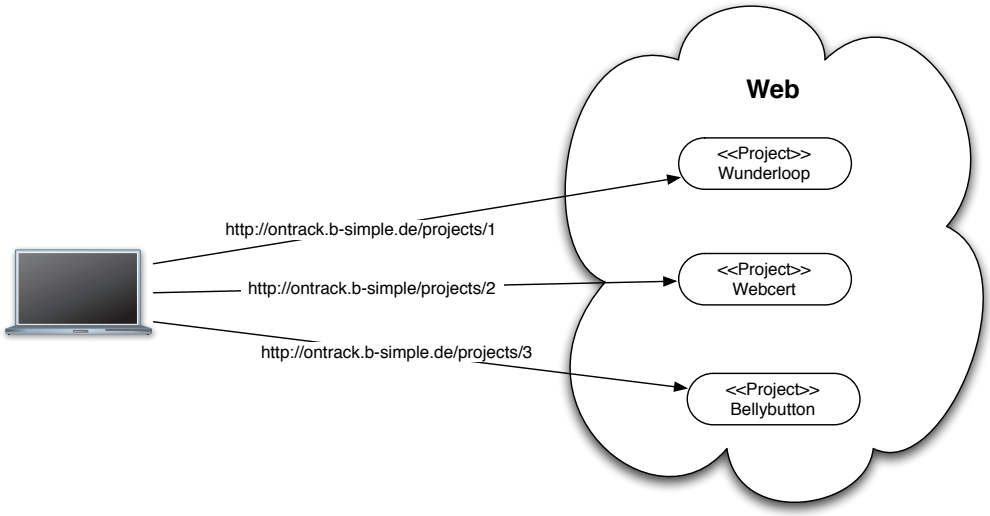

Figure 1.1: Resources on the web and their URLs

All three resources in figure 1.1 are addressed by URLs that are almost identical, followed by the id of the resource. Note that the URL doesn't show what should happen with the resource.

In the context of a Rails application, a resource is a combination of a dedicated controller and a model. So from a technical standpoint, the project resources from figure 1.1 are instances of the ActiveRecord class *Project* in combination with a *ProjectsController* which is responsible for the manipulation of the instances.

[1] If we want to make a distinction between REST- and non-REST-based Rails applications, we use the word traditional. Traditional does not mean old or even bad, it's only used to make a reference to an equivalent non-REST concept. This comparison should help to better explain the new technology.

1.2 Why REST?

This is a good question when you consider that we have been very successfully developing Rails applications for two years now using the proven MVC concept. What REST shows us, however, is that Rails has room for conceptional improvements, as the following feature list of REST-based Applications makes clear:

Clean URLs. REST URLs represent resources and not actions. URLs always have the same format: first comes the controller and then the id of the referenced resource. The requested manipulation is independent of the URL and is expressed with the help of HTTP verbs.

Different Response Formats. REST controllers are developed in a way that actions can easily deliver their results in different response formats. Depending on the requirements of the client, the same action can deliver HTML, XML, RSS, or other data formats–the application becomes able to handle multiple client demands, cleanly and simply.

Less Code. The development of multi-client-capable actions avoids repetitions in the sense of DRY[2] and results in controllers having less code.

CRUD-oriented Controllers. Controllers and resources melt together into one unit–each controller is responsible for the manipulation of one resource type.

Clear Application Design. RESTful development results in a conceptually clear and maintainable application design.

The forthcoming chapters of this tutorial will make the features above clear with the help of several examples.

1.3 What's New?

If you are now thinking that REST-based application design makes all of your previously-gained Rails development experience useless, we can assure you that this is not the case: REST is still MVC-based and from an technical point of view, can be reduced to the following new techniques:

- The usage of *respond_to* in controller code.
- New helper methods for links and forms.
- The usage of URL methods in controller redirects.
- New routes that are generated from the method *resources* in routes.rb.

Once you have an understanding of REST and begin using its techniques, RESTful application design becomes second nature.

[2] Don't repeat yourself

1.4 Preparations

We're going to explain the new REST-specific features of Rails within the context of the example application in our book *Rapid Web Development mit Ruby on Rails* [1], *Ontrack*, a project management application. We will not develop the full application here, but will use the same terminology to create a technical environment for REST concepts.

Let's start with the generation of our Rails project:

```
> rails ontrack
```

Then, create the development and test databases:

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

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

1.4.1 Rails 1.2

We will assume that not all of our readers want to install Rails 1.2 as a system-wide installation, where all of their Rails applications will use Rails 1.2. Instead, we will install Rails 1.2 directly into our project, so that only our example *ontrack* will use the new version. According to the documentation, the tag for the latest stable version of Rails 1.2 is *rel_1-2-2*. This rake command installs the new version only into ontrack:

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

Alternative installation methods and information about newer versions of Rails can be found on the Rails home page *http://www.rubyonrails.org*.

1.5 Resource Scaffolding

REST-oriented Rails development can be easily illustrated with a resource *Project* created from the new scaffold generator *scaffold_resource*. The generator takes a resource name, *project*, as its first parameter, and optionally a list of model attributes and their types. These attributes are needed for the generation of a database migration script and also for the appropriate fields in the created views:

```
> 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
```

In addition to the model, controller, and views, the generator also creates a fully-functional migration script and a new mapping entry *map.resources :projects* in *config/routes.rb*. It is this last entry that is responsible for the RESTful character of our new controller. We won't go too deeply into it just yet–instead, we'll go through all the generated parts, step by step.

1.6 The Model

As we mentioned at the beginning, REST resources are, in the context of Rails, the combination of controller and model. The model is a normal ActiveRecord class that inherits from *ActiveRecord::Base*:

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

So in the case of the model, there is nothing new to learn. But don't forget to generate the database table:

```
> rake db:migrate
```

1.7 The Controller

The generated *ProjectsController* is a CRUD controller that manipulates the resource *Project*. This means that the controller belongs to exactly one resource type and offers a designated action for each of the four CRUD operations[3]:

[3] Additionally, the controller consists of the action *index*, to display a list of all resources of this type, the *new* action, for opening the new form, and the *edit* action, for opening the editing form.

Listing 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
```

If you look at the generated controller you will not find very much new here: there are actions for creating, retrieving, updating and deleting projects. Both controller and actions appear normal at a first glance, but they all have a generated comment showing the relevant request URL including its HTTP verb. These are the REST URLs, and we will take a closer look at them in the next section.

1.7.1 REST URLs

Basic REST URLs aren't composed of a controller name, action name, and optional model id (e.g., */projects/show/1*), as we're used to seeing in traditional Rails applications. Instead, they include only the controller name followed by the id of the resource to manipulate:

```
/projects/1
```

With the loss of the action name, it is no longer obvious what should happen to the indicated resource. Should the above shown URL list or delete the resource? The answer to this question comes from the HTTP method[4] which is used when

[4] In this tutorial, we will use the term HTTP *verb* to describe the HTTP method because it better expresses that the request will result in an *action*.

requesting the URL. The following table lists the four HTTP verbs along with their REST URLs and shows how they correspond to controller actions and traditional Rails URLs:

Table 1.1: HTTP Verbs and REST-URLs

HTTP Verb	REST-URL	Action	URL without 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

The HTTP verb determines which action will be executed. URLs become DRY and address resources instead of actions. The REST URLs are identical for all operations except for POST because when creating a new project an id does not yet exist.

Remark: Entering the URL *http://localhost:3000/projects/1* in the browser always calls *show* because when a browser fetches a URL, it uses the GET verb. Since browsers only use GET and POST HTTP verbs, Rails must perform a bit of sleight-of-hand to enable *destroy* and *update* actions. Rails provides special helpers for the creation of links to delete and update resources: the HTTP DELETE verb is transmitted to the server in a hidden field inside of a POST request (see section 1.8.3) for delete actions, and the PUT verb is similarly sent to update existing resources (see section 1.8.2).

1.7.2 REST Actions Use respond_to

We have seen that REST actions are activated through a combination of a resource URL and an HTTP verb. This results in clean URLs that only address the resource to be manipulated. But what other options can be specified in a REST URL?

A REST action can also respond to different client types with different response formats. Typical client types for a web application are, of course, browser clients, but also, for example, web service clients that often expect server responses in XML, and RSS reader clients, which prefer their responses in RSS or Atom format.

The principal mechanism for the generation of the answer format requested by the client is the method *respond_to* which is generated by the scaffold-generator in the CRUD actions. The following code fragment demonstrates the *show* action as an example usage of *respond_to*:

Listing 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
```

The method *respond_to* gets a block with format-specific instructions. In the example, the block handles two formats: HTML and XML. Depending on the client's requested format, the corresponding instructions are executed. If HTML is the format requested, nothing is specified, which causes the delivery of the default view, *show.rhtml*. If the requested format is XML, the requested resource will be converted into XML and delivered to the client.

The requested format of the response is indicated in one of two ways: it's either put into the Accept field of the HTTP header of the request or it's appended to the request URL.

1.7.3 Accept Field of the HTTP Header

Let's start with the first form, using the HTTP verb in the accept field of the HTTP header. It's very easy to set the HTTP header with the command-line HTTP tool *curl*. But before we demonstrate that, we'll start the web server:

```
> 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) [i686-darwin8.6.1]
[2006-12-30 18:10:50] INFO  WEBrick::HTTPServer#start: pid=4709 port=3000
```

Next, we browse to Ontrack's projects page, *http://localhost:3000/projects*, and create one or more projects (see figure 1.2).

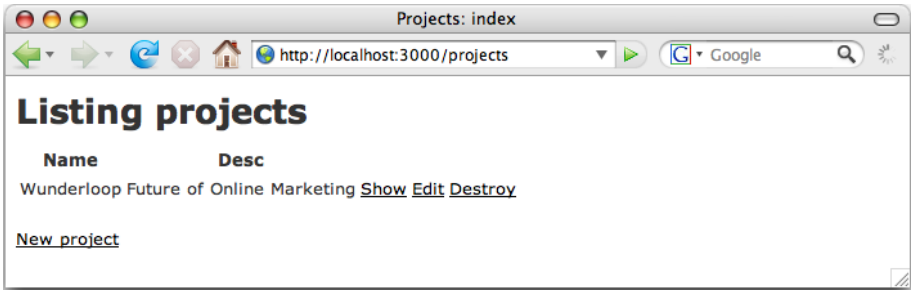

Figure 1.2: Create some projects

This *curl* command requests the project resource 1 in XML format:

```
> 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
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>
```

The Rails dispatcher routes this request to the *show* action. Because of our XML preference in the HTTP Accept field, the method *respond_to* executes the format.xml block and the requested resource is converted into XML and delivered as the response.

curl is not only good for testing of different response formats–it is also good for sending HTTP verbs that are normally not supported by web browsers. For example, the following request deletes the project resource with id 1:

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

This time the request uses the HTTP DELETE verb. The Rails dispatcher evaluates the HTTP verb and routes the request to the *ProjectsController.destroy* method. Note that the URL is identical to the URL used in the last *curl* request. The only difference is the HTTP verb that was used.

1.7.4 Format Specification Via the Request URL

The second way to request the response format is to encode the desired format into the request URL. (If you deleted your last project resource in the previous example, go back to the Ontrack application and create a project now.) To show project 1 and request the response in XML format, use your browser to navigate to the following URL:

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

Attention Mac users: This request is easier to observe in Firefox than in Safari because Safari simply ignores the delivered XML. Firefox formats the XML nicely (see figure 1.3).

So far, we have learned how REST controllers work and what the appropriate request URLs look like. In the following two sections, we will see how to use and generate these URLs in views and controllers.

Figure 1.3: Project Wunderloop in XML

1.8 REST URLs and Views

Views represent the interface between the application and its users. The user interacts with the application using links and buttons. Traditionally, Rails developers generate links using the *link_to* helper. The *link_to* method expects a hash containing the controller and action, along with some optional request parameters:

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

What immediately springs to mind is that this traditional usage of *link_to* doesn't work very well with our new REST philosophy–REST specifies that URLs identify the resource and the action is determined by the HTTP verb of the request. What needs to happen is for links and buttons to deliver the appropriate HTTP verb together with the URL.

Rails provides the solution: it still uses *link_to* to generate the link, but the hash is replaced with a call to a *Path* method. Path methods create link destinations that *link_to* puts into the href attribute of the generated link. As a first example, we create a link to the *show* action of the *ProjectsController*. Instead of specifying a controller, action and project id in a hash, *project_path()* is used:

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

The href attribute of the traditional *link_to* method is composed of a controller, an action, and a project id. Instead, *project_path* constructs a RESTful path consisting only of the controller and the referenced project id. Because links use the HTTP GET verb by default, the Rails dispatcher recognizes that the requested project should be displayed, and the *show* action is executed.

For each resource, Rails automatically generates the seven standard path methods shown in table 1.2.

Each Path method is associated with an HTTP verb, meaning that this is the HTTP method that is sent to the server when clicking on a link or button. Some of the

Table 1.2: Standard Path methods

Path Method	HTTP Verb	Path	Requested Action
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

requests (show, create) are transmitted by default with the right HTTP verb (here, GET and POST). Others (update, delete) need to be treated in a special way (using hidden fields) because, as was already mentioned, browsers don't ever use PUT and DELETE verbs. You will read more about this special treatment and its implementation in section 1.8.3 and section 1.8.2.

A closer look at the table shows that four HTTP verbs are not enough to map all of the CRUD actions. The first two methods, *projects_path* and *project_path(1)*, work well with GET and can be routed directly to their appropriate actions. However, things don't look as bright for *new_project_path* and *edit_project_path*.

1.8.1 New and Edit

A click on the New link is transmitted to the server using the GET verb. The following example shows that the generated path consists of the *ProjectsController* and the action *new* that is to be called:

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

Is this a crack in the REST philosophy? Maybe at a first glance. But if you look closer it becomes clear that *new* is not a REST/CRUD action–it doesn't modify anything in the application's data–it is more of a preparatory action for the creation of a new resource. The real CRUD action *create* is first executed when the *new* form is eventually submitted. The resource id is not present in the URL because there is no resource yet.

It's a similar story for the method *edit_project_path*–it refers to a concrete resource, but it's not a CRUD action. *edit_project_path* is used to prepare the real CRUD action call when *update* is requested. The difference between *edit_project_path* and *new_project_path* is that *edit_project_path* needs the id of the project to be manipulated. Following the REST convention, the id comes after the controller: */projects/1*. However, if this path were submitted to the server with GET, the call would be routed to the *show* action. To distinguish the edit action, *edit_project_path* simply extends the generated href attribute in a special way. This is how the generated HTML link finally looks:

```
link_to "Edit", edit_project_path(project)
```

```
=>
<a href="/projects/1;edit">Edit</a>
```

It's okay for *new* and *edit* to include the action in their URL because neither are real REST/CRUD actions. The same principle is also used for developing other actions that use names other than the standard CRUD names. We will have a look at this in section 1.12.

1.8.2 Path Methods in Forms: Create and Update

Traditional forms are created using the helpers *form_tag* or *form_for* with specifying a submit action:

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

In a typical REST application, the :url hash is set to the return value of a call to a Path method:

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

The New Form

A form gets transmitted to the server with a standard POST. The request of *projects_path* without an id results in the path /projects–when submitted using POST, this results in the execution of the *create* action:

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

The Edit Form

In accordance with the REST philosophy, an update is transmitted via PUT. But as we know, neither PUT or DELETE are supported by web browsers. The solution Rails offers is the usage of a *method* key in the *:html* hash of *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 generates a hidden *_method* field that contains the appropriate HTTP verb *put*. The dispatcher looks at this field and routes the request to the *update* action.

1.8.3 Destroy

Note that the method used for both showing and deleting a project is *project_path*:

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

The only difference is that the destroy link additionally uses the parameter *:method* to name the HTTP method to use (:delete, in this case). Because the web-browser doesn't support the DELETE verb, Rails generates a JavaScript fragment that gets executed when clicking on the link:

```
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>
```

This script creates a form on the fly and ensures that the DELETE HTTP verb gets transmitted to the server in the hidden field *_method*. Again, the Rails dispatcher analyzes the content of this field and sees that the request should be routed onto the action *destroy*.

1.9 URL Methods in the Controller

In the same way that links and submit actions get created in REST views with the help of new helpers, controllers have to take special care when using the new technique when doing redirects. For this, Rails uses URL methods, generating a URL method corresponding to each Path method:

`project_url` for `project_path`

or

`projects_url` for `projects_path`.

In contrast to Path methods, URL methods create fully-qualified URLs including the protocol, host, port and path:

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

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

In the controllers of a REST application, URL methods are used everywhere where the *redirect_to* method gets traditionally handed over a controller/action/parameter hash. So the following:

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

in a REST application becomes:

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

You can find an example for this in the *destroy* action, where *projects_url* is used without any parameters to redirect to a list of all projects after a project has been deleted:

Listing 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 REST Routing

So far, we have explained the REST concept and a bunch of new methods to be used in links, forms and controllers. But we haven't yet explained where these methods come from. Responsibility for the existence of all these methods and the proper handling of REST requests belongs to a new entry in the Rails routing file, *config/routes.rb*:

```
map.resources :projects
```

The entry was generated by the *scaffold_resource* generator. It creates the named routes that are needed for requesting the REST actions of the *ProjectsController*.

In addition, *resources* generates the Path and URL methods for the *Project* resource that we've been experimenting with in this chapter:

```
map.resources :projects
=>
Route         Generated Helper
------------------------------------------------------------
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 Conventions

A necessary consequence of development using REST routes is to comply with the naming conventions for the controller methods that handle the CRUD actions. The following *link_to* call, the resulting HTML, and the Rails dispatcher's behavior illustrate this:

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

Neither the *link_to* call nor the generated HTML include the name of the action to call. The Rails dispatcher knows that the route */projects/:id* must be routed onto the *show* action of the *ProjectsController* if the request was sent via a GET verb. The controller must have an action with the name *show*. The same convention is true for the actions *index*, *update*, *create*, *destroy*, *new* and *edit*–each REST controller must implement these methods.

1.10.2 Customizing

REST routes can be adapted to application-specific requirements with the help of the following options:

- **:controller.** Specifies the controller to use.
- **:path_prefix.** Names the URL prefix, including any necessary variables.
- **:name_prefix.** Names the prefix of the created route helpers.
- **:singular.** The singular form of the resource name.

The following routing entry creates routes for a new resource, *Sprint*. Sprint is a synonym for an iteration and maps onto the ActiveRecord model *Iteration* to be introduced in the next section:

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

The option *path_prefix* is used for the URL format. Each URL starts with */ontrack* followed by a project id. The responsible controller should be *OntrackController*. Therefore the URL

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

gets routed according to the given routing rules to the *index* action of the *OntrackController*. Another example is the URL

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

which gets routed to the *show* action of our *OntrackController*.

While *path_prefix* defines the format of URLs and paths, *name_prefix* makes generated helper-methods start with *ontrack_*. For example:

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

or

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

1.11 Nested Resources

RESTful development gets really interesting when using so-called *nested resources*. Here the importance of clean URLs will become much clearer. Nested resources will further help to clarify REST and should help you to better understand the REST paradigm.

Nested resources are strongly-coupled resources, in the sense of a parent-child relationship. In the context of Rails, we mean models that represent a one-to-many relationship, such as *Projects* and *Iterations* in Ontrack. Nested REST controllers are still responsible for the manipulation of a single model type, but in the case of a child controller they also read the model of the parent resource. This may sound complicated at first, but it will become clear in the course of this section.

The REST approach of Rails reflects the relationship between nested resources in its URLs and maintains the principal of *clean* REST URLs. We'll describe this principal with the example of iterations and projects in Ontrack. It starts with the generation of the new *Iteration* resource and the creation of the appropriate *iterations* database table:

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

Projects have a one-to-many relationship to iterations. This relationship is implemented in the model:

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

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

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

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

In addition to creating model, controller and views, the generator also creates a new routing entry in *config/routes.rb*:

```
map.resources :iterations
```

As with the similar line for *projects*, this statement generates new routes and helpers for the manipulation of the *Iteration* resource. However, iterations only make sense in the context of a concrete project–this is not taken into consideration by the created routes and helpers. For example, the helper *new_iteration_path* creates the path */iterations/new*, which contains no information about the project that the new iteration belongs to.

The point of nested resources is essentially the realization that a child resource (here, an Iteration) doesn't–and can't–exist outside of the context of the parent resource (in this case, a Project) to which it belongs. REST Rails tries to reflect this in the usage of URLs and the controller of the child resource. To get this to work, you need to replace the generated resource entry in *config/routes.rb*:

```
map.resources :iterations
```

with:

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

This entry causes *Iteration* to be a nested resource and generates appropriate routes that allow you to manipulate iterations only in the context of a project. The generated routes have the following format:

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

For example, entering the URL

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

results in the *index* action of the *IterationsController* being executed, getting the id of the project via the request parameter *:project_id*. Note especially how the URL clearly indicates the underlying ActiveRecord association:

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

Nested REST URLs are still clean REST URLs, meaning they address resources and not actions. The fact that a resource is a nested resource is indicated by two REST URLs appearing one after another in one URL. A request for the *show* action should make this clear:

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

1.11.1 Adapting the Controllers

The generated *IterationsController* doesn't automatically know that it is responsible for a nested resource and that with every request it also gets the id of the parent project. For example, the *index* action still loads all saved iterations, even though the calling URL clearly indicates that only the iterations of a specific project should be loaded:

Listing 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
```

We need to rewrite the action so that only the iterations of the chosen project are loaded:

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

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

All the actions of *IterationsController* will only work properly with a */projects/:project_id* prefix–they require that the parent project that defines the context of the iteration actions be present. Since the scaffold_resource generator doesn't create controller code that handles nested resources, not only does *index* need to be rewritten, but the actions *create* (see section 1.11.3) and *update* (see section 1.11.4) need to be adjusted, too.

1.11.2 New Parameters for Path and URL Helpers

The resource entry in *config/routes.rb* for iterations generates not only new routes, but helpers as well. Like the routes, these helpers expect a project id as the first parameter. For example, the path specifying the list of all iterations of a project is generated by the helper *iterations_path*. The names of the nested helpers are identical to the names of non-nested resource helpers. What changes is the number of parameters that the helpers expect. Helpers for nested resources always expect the id of the nesting resource as the first parameter, in this case the project id. Here, for example, the *iterations* link (that shows up in the *index* view of the *ProjectsController*) shows all iterations of the chosen project–the URL for the list of iterations is generated by the helper *iterations_path*, which expects the project id parameter:

```
link_to "Iterations", iterations_path(project)
```

```
=>
<a href="/projects/1/iterations">Iterations</a>
```

For a better understanding, let's look at the link where it actually appears, in the *ProjectsControllers index* view:

Listing 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 %>
...
```

A consequence of the changed parameter list is that not only are some actions in the controller broken, but also many scaffold views for the iterations. For example, the index view contains a table with all iterations, and each iteration has three links:

Listing 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 %>
...
```

All links start out with the id of the respective iteration as the first and only parameter. This doesn't work any longer because the first parameter of an iteration helper should be the project id. The needed change looks like this:

Listing 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 %>
....
```

An alternate way to correct the parameter list of the nested resource helpers is to pass the required ids in a hash:

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

This increases the readability of code when it's not immediately clear what object type the iteration relates to.

1.11.3 Adding New Iterations

Adding new iterations also only works in the context of a previously-chosen project. To easily deal with this, we simply add a *New Iteration* link to the *ProjectsControllers index* view:

Listing 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 %>
...
```

For the path method we use *new_iteration_path* which generates for the project with id 1 the following HTML:

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

The link routes to the *new* action of the *IterationsController*. The action receives the value 1 via the request parameter *project_id,* which is the id of the current project.

The project id is thereby available in the rendered *new* view of the *IterationsController* and can be used there by the helper *iterations_path*, which is responsible for the generation of the new iteration form. The generated form contains a nested route in the *action* attribute, which contains the id of the project in which a new iteration should be created:

Listing 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">
```

The usage of *params[:project_id]* in *iterations_path* is optional because Rails automatically sets the request's *project_id* parameter as the generated *action* attribute. This means that

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

has the same effect.

The REST routing ensures that the form action */projects/1/iterations*, in combination with the HTTP POST verb results in an execution of the *create* action in the *IterationsController*. The HTTP method (method='post') of the generated form tag was created by the helper by default, since no explicit HTTP verb was given and *post* is the default value.

Besides the actual form parameters, the create action gets the project id via the request parameter *project_id*. Therefore you have to change the method so that the newly created iteration is assigned to the right project:

Listing 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 { redirect_to iteration_url(@iteration.project,
9                      @iteration) }
10       format.xml  { head :created, :location =>
11                     iteration_url(@iteration.project, @iteration) }
12     else
13       format.html { render :action => "new" }
14       format.xml  { render :xml => @iteration.errors.to_xml }
15     end
16   end
17 end
```

In line 3, the project is assigned explicitly. We have also extended the helper *iteration_url* with the project id in lines 8 and 11.

To make adding of new iterations really work, you need to extend the *Edit* and *Back* links in the *show* view of the *IterationsController* to include the project id:

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

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

This view is rendered after the creation of a new iteration and would otherwise throw an exception if we didn't pass in the project id.

1.11.4 Editing Existing Iterations

For editing iterations, two changes are necessary in the generated code.

The default *form_for* helper in the *edit* view of the *IterationsController* gets only the iteration id:

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

However, both the project id and the iteration id are needed:

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

A similar change needs to be made in the *update* action that is called from the form– the method *iteration_url* in line 7 is passed only the iteration id after a successful update:

Listing 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
```

As above, Line 7 needs to be changed to:

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

After all these changes have been made, the create and update views and their actions should finally be working. Iterations can now be created and edited. But to be absolutely sure, look closely at the *IterationsController* and its respective views. Check all path and URL helpers, looking for any that don't aren't receiving a project id and change them as we changed the create and update views.

1.12 Defining Your Own Actions

The *resources* entry in the routing file generates named routes and helpers for CRUD actions. But how do we create routes and helpers for non-CRUD actions that also belong to the controller? As an example, let's look at the new *close* action in the *ProjectsController*. This action is used to close a project–i.e., to mark a project as finished.

To begin with, here's the database migration:

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

Listing 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
```

Next, we create a *Close* link in the *ProjectsControllers index* view:

Listing 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 %>
```

Two questions arise when adding this link:

1. Which HTTP method should be used when sending *close*?
2. How does the helper that creates the path to the *close* action get generated?

Because *close* is not a typical CRUD action, Rails does not know which HTTP verb it should use. Close is a specialized action of the project, a kind of an update to the project record, so according to REST, it should be sent using POST. We define the route and the corresponding helper in the routing file *config/routes.rb* with the help of the *member* hash in the *resources* call for *projects*. The hash consists of action-method pairs and specifies which action should or is allowed to be called with which HTTP verb[5].

Allowed values are *:get*, *:put*, *:post*, *:delete* and *:any*. If an action is marked with *:any*, it is allowed to call the action with any HTTP verb. In our example, *close* should be requested via POST, so we have to change the *resources* entry as shown below:

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

After adding this entry, we can use the new helper *close_project_path* when creating the Close link we talked about earlier:

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

However, a Routing Error appears when clicking the resulting link:

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

The route exists but the new *resources* entry allows only requests via HTTP POST. Other HTTP methods (like GET, used in the link above), are denied by Rails. What we need is something similar to the *destroy* link: a helper that generates a form that is sent via POST to the server. Luckily, Rails has such a helper–*button_to* does exactly what we need:

```
<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>
```

The only thing missing is the *close* action in the *ProjectsController*:

Listing 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 }
```

[5] Rails covers the routes with HTTP restrictions, resulting in RoutingError Exceptions if an Action is requested with the wrong HTTP verb.

```
    else
      flash[:notice] = "Error while closing project."
      format.html { redirect_to projects_path }
      format.xml  { head 500 }
    end
  end
end
```

Besides *:member*, the keys *:collection* and *:new* can be specified in the *resources* call. *:collection* is required when the action is performed on a collection of resources of a particular type, rather than a single resource of the type. An example is requesting a project list as an RSS feed:

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

The hash key *:new* is used for actions that work on new resources that are not yet saved:

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

1.12.1 Are We Still DRY?

The last paragraph could be considered to be a violation of the DRY principal–actions are no longer implemented solely in the controller, they are now also named in the routing file.

As an alternative to the RESTful patterns described above, you can also call non-REST actions in the traditional way using the action and the project id:

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

If you haven't deleted the *map.connect ':controller/:action/:id'* call in the routing file, the necessary routes for this should still be defined. However, the old route will be functional only if you haven't changed the *resources* call for *projects* as described above.

1.13 Defining your own Formats

By default, the *respond_to* method only knows the following formats:

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

```
  wants.yaml
end
```

As an extension to this you can register your own formats as MIME types. Let's say you have developed a PIM application and you want to deliver the entered addresses via the *vcard* format[6]. To do that, you first must register the new format in the configuration file, *config/environment.rb*, like this:

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

Now we can extend the *show* action of the *AddressesController* so that it can deliver addresses in the vcard format if the client asks for it.

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

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

The method *to_vcard* is not a standard ActiveRecord method and must be implemented using the vcard specification (RFC2426). If implemented correctly, the following URL should result in an address, delivered in standard vcard XML syntax:

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

1.14 RESTful AJAX

With regard to the development of RESTful AJAX applications, there is not much new to learn. You can use the known remote helpers and give the *:url* parameter the *path* method instead of a controller and action hash. The following code snippet converts the *destroy* link in the *ProjectsControllers index* view into an AJAX link:

```
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>
```

One note: don't forget to include the needed JavaScript libraries if you don't want to waste a quarter of an hour figuring out why the link is not working (as I did). One way to achieve this is to call the *javascript_include_tag* helper in the layout file *projects.rhtml* of the *ProjectsController*:

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

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

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

A click on the link gets routed to the *destroy* action of the *ProjectsController*. From a business logic point of view, the method already does everything right: it deletes the chosen project. What's missing is an additional entry in the *respond_to* block for delivering the client the newly requested format, in this case JavaScript. The following piece of code shows the already-updated *destroy* action:

Listing 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
```

The only change to the old version is the additional *format.js* entry in the *respond_to* block. Because the new entry has no further block of code to execute, Rails acts in the standard manner and delivers an RJS template with the name *destroy.rjs*. It looks like this:

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

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

The template deletes the element with the id *project_ID* from the web browsers DOM tree. To make this work in the *index* view of *ProjectsController* you have to add a unique id to the table rows:

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

```
...
<% for project in @projects %>
  <tr id="project_<%= project.id %>">
```

The minor edit required to update *ProjectsController* to support AJAX demonstrate the benefits of REST applications and the DRY principal. One extra line in the controller makes the same action capable of handling JavaScript/AJAX requests!

The example also demonstrates a general rule of developing RESTful controllers clear: implementing more logic outside of the *respond_to* block leads to fewer repetitions in the resulting code.

1.15 Testing

No matter how exciting RESTful development with Rails is, testing shouldn't be forgotten! That we have already developed too much without running our unit tests at least once becomes clear when we finally run the tests with *rake*[7]:

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

The good news: all unit tests and the *ProjectsControllerTest* functional test are still running. The bad news: all seven *IterationsControllerTest* functional tests are broken.

If all tests of a single test case throw errors, it's a clear sign that something fundamental is wrong. In our case the error is obvious: the test case was generated by the scaffold generator for iterations without a parent project. We extended the iterations to make them belong to a project when we added all the needed functionality in *IterationsController* and now all actions there are expecting the additional request parameter, *:project_id*. To fix this, extend the request hash of all test methods with the parameter *project_id*. As an example, we take the test *test_should_get_edit*:

Listing 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
```

Additionally, the *IterationsControllerTest* must also load the *projects* fixture:

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

After all these changes only two tests should still fail: *test_should_create_iteration* and *test_should_update_iteration*. In both cases, the reason is a wrong *assert_redirected_to* assertion:

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

It's obvious what's going on here: we have changed all redirects in the *IterationsController* so that the first parameter is the project id. The assertion checks only if there is an iterations id in the redirect call. In this case, the controller is right and we have to adopt the test:

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

By the way, the use of path methods in redirect assertions is the only difference between REST functional tests and non-REST functional tests.

[7] Attention: Don't forget to create the test database *ontrack_test* if you haven't already!

1.16 RESTful Clients: ActiveResource

ActiveResource is often mentioned together with REST. ActiveResource is a Rails library for the development of REST-based web service clients. Such a REST-based web service client uses the four typical REST HTTP verbs to communicate with the server.

ActiveResource is not a part of Rails 1.2 but is available via the development trunk and can be installed using *svn*, the Subversion source control application[8]:

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

ActiveResource abstracts client-side web resources as classes that inherit from *ActiveResource::Base*. As an example, we use the existing server-side resource *Project* that we model on the client side as follows:

```
require "activeresource/lib/active_resource"

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

The ActiveResource library is explicitly imported. Additionally, the URL of the service gets specified in the class variable *site*. The class *Project* abstracts the client-side part of the web service so well that the programmer gets the impression he is working with a normal ActiveRecord class. For example, there is a *find* method that requests a resource with the given id from the server:

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

The *find* call executes a REST-conforming GET request:

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

The server delivers the answer in XML. From the XML, the client generates an ActiveResource object *wunderloop* that offers, like an ActiveRecord model, getter and setter methods for all of its attributes. But how does it work with updates?

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

The call to *save* converts the resource into XML and sends it via PUT to the server:

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

Take your web browser and reload the list of projects. The changed project should have a new name.

Creating new resources via ActiveResource is as easy as requesting and updating resources:

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

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

The new project is transmitted to the sever in XML via POST and gets saved into the database:

```
POST /projects.xml
```

Reloading the project list view in the browser shows the newly created project. The last of the four CRUD operations we have to look at is the deletion of projects:

```
bellybutton.destroy
```

The calling of *destroy* gets transmitted via DELETE and results in the deletion of the project on the server:

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

ActiveResource uses all of the four HTTP verbs as appropriate for REST. It offers a very good client-side abstraction of REST resources. Additionally, many other known methods of ActiveRecord work in ActiveResource, such as finding all instances of a resource:

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

We believe that ActiveResource is a very good foundation for the development of loosely-coupled systems in Ruby. It is a good idea to have a look into the trunk and experiment with the basic classes of ActiveResource.

1.17 Finally

You don't have to use REST everywhere. Hybrid solutions are conceivable and can easily be implemented. Typically, you're in the middle of a project when new Rails features appear. It's not a problem to develop single REST-based models and their dedicated controllers to gain some experience. When starting a new application from scratch, think about doing it fully in REST from the beginning. The advantages are clear: a clean architecture, less code and multi-client capability.

Bibliography

[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