

REST & Rails

Web Services for the Rails World

Rein Henrichs <http://reinh.com>

```
self.werewolf? # => false
```

REST & Rails

- What Is REST?
- Make Rails RESTful
- Eating RESTfully
- RESTful Tips & Tricks
- RESTful Resources

What is REST?

Representational

State

Transfer

Of Resources

Architectural Styles and the Design of Network-based Software Architectures

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

[Roy Thomas Fielding](#)

2000

Dissertation Committee:
Professor Richard N. Taylor, Chair
Professor Mark S. Ackerman
Professor David S. Rosenblum

Resources

- Sources of specific information
- Identified by a global identifier (URI)
- Hyperlinked to other resources
- Examples:
 - del.icio.us tags, links
 - Amazon S3 buckets
 - flickr photos
 - blog posts
 - twitters
 - (Nearly) everything else on the internet

Resources

- Things
- With names
- That often relate to other things
- On the internet

Representations

- Formats
 - HTML, XML, JSON, JPG, SVG, etc.
- Content Types (MIME Types)
 - text/html, application/xml, etc.

State

- Resource state, not application state
- Often database backed (CRUD)

Transfer

- Use **HTTP Verbs**
 - (Constraints are liberating)
- Design a **Uniform Interface**
 - Simple, Visible, Reusable
- Stateless
- Manipulate resources by transferring representations

Transfer

RESTless

RESTful

VERB

HREF

VERB

URI

POST /users/create

POST /users

GET /users/1

GET /users/1

POST /users/1/update

PUT /users/1

???? /users/1/delete

DELETE /users/1

Make Rails RESTful

- Design your Resources
- Create their Representations
- Expose them
- Profit!

Design Your Resources

- Identify your resources
- Design a URI scheme

Entity	URI
User	<code>/users/quentin</code>
Photo	<code>/photos/grand-canyon</code>
Comment	<code>/photos/grand-canyon/comments/1</code>

- In rails: `map.resources`

map.resources

maps URIs to controllers

maps HTTP verbs to actions

map.resources :users

Resource	Verb	Action
/users	GET	index
/users	POST	create
/users/1	GET	show
/users/1	PUT	update
/users/1	DELETE	destroy

Design Your Resources

Caveat Implementor

HTML and XHTML currently only support two HTTP verbs: GET and POST. In order to get around this limitation, Rails uses overloaded POST requests with a `_method` parameter to simulate PUT and DELETE requests. Rails handles this transparently through its view helpers and resource routing.

Create Representations

Which formats will you provide?

- User

- HTML
- XML
- JSON
- vCard?

- Photo

- HTML
- XML
- JPG
- Sizes?

- Comment

- XML
- JSON

Expose Resources

- Determine the requested resource
- Determine the requested representation
- Manipulate the resource's state based on the request method and parameters
- Deliver an acceptable representation of the resource's (new) state

Expose Resources

Determine the requested resource

- Resources are identified by URIs
 - “Uniform Resource Indicator”
- Rails maps URIs to controllers and actions
 - `map.resources :users`

Expose Resources

Determine the requested representation

- “Accept” header
- `params[:format]`
 - `users/1.xml`

Expose Resources

Manipulate the resource's state (Rails 101)

- Create the resource

```
@user = User.new(params[:user])  
@user.save
```

- Retrieve the resource

```
@user = User.find(params[:id])
```

- Update the resource

```
@user.update_attributes(params[:user])
```

- Delete the resource

```
@user.destroy
```

Expose Resources

Deliver a representation

- Based on Accept header

```
def index
  @users = User.find(:all)

  respond_to do |format|
    format.html # text/html, application/xhtml+xml
    format.xml  # application/xml
    format.js   # text/javascript
  end
end
```

Expose Resources

Deliver a representation

- Based on format

```
def index
  @users = User.find(:all)

  respond_to do |format|
    format.html # index
    format.xml  # index.xml
    format.js   # index.js
  end
end
```

Expose Resources

Deliver a representation

- Using templates

```
def index
  @users = User.find(:all)

  respond_to do |format|
    format.html # index.html.erb
    format.xml  # index.xml.builder or index.xml.erb
    format.js   # index.js.rjs or index.js.erb
  end
end
```

Expose Resources

Deliver a representation

- Using coercion

```
def index
  @users = User.find(:all)

  respond_to do |format|
    format.html # N/A
    format.xml { render :xml => @users.to_xml }
    format.js   { render :text => @users.to_json }
  end
end
```

Expose Resources

Deliver a representation

- Using Rails scaffold generator*
 - `ruby script/generate scaffold User`
 - Creates CRUD and index actions that map resources to an ActiveRecord model and respond to html and xml

* edge rails / Rails 2.0 (scaffold_resource in 1.2.x)

Eating RESTfully

- `ActiveResource`
- `Alternatives`

Eating RESTfully

ActiveResource

- Designed to consume RESTful web services via XML
- Treats resources as models
- Fairly strict about resource architecture
- Some examples please?

Eating RESTfully

ActiveResource

- Consuming our users

```
class User < ActiveResource::Base
  self.site = "http://www.example.com"
end
```

```
>> User.find(:all) # users.xml
```

```
>> User.find(1) # users/1.xml
```

- Consuming our photos

```
class Photo < ActiveResource::Base
  self.site = "http://www.example.com"
end
```

Eating RESTfully

Alternatives

- `rest-open-uri`
- `jQuery.getJSON()`

Eating RESTfully

```
gem install rest-open-uri
```

- Wraps Bare metal Net::HTTP library
- Extends open-uri to allow PUT, POST, and DELETE
- Some examples please?
 - <http://pastie.caboo.se/117513>

Eating RESTfully

`jQuery.getJSON()`

- Asynchronous HTTP request for text/javascript
- JSON object can be passed to anonymous function
- Some examples please?

Eating RESTfully

jQuery.getJSON()

```
var flickr_cats_uri = "http://api.flickr.com/services/feeds/photos\_public.gne?  
tags=cat&tagmode=any&format=json&jsoncallback=?"
```

```
$.getJSON(flickr_cats_uri, function(data){  
  $.each(data.items, function(i,item){  
    $("").attr("src", item.media.m).appendTo("#images");  
    if ( i == 3 ) return false;  
  });  
});
```



RESTful Tips & Tricks

- DRY ActiveRecord
- DRY RESTful Controllers
- Turn verbs into nouns
- Custom MIME types

RESTful Tips

DRY ActiveSupport

- Consuming our users

```
class User < ActiveSupport::Base
  self.site = "http://www.example.com"
end
```

- Consuming our photos

```
class Photo < ActiveSupport::Base
  self.site = "http://www.example.com"
end
```

RESTful Tips

DRY ActiveSupport

- Create a base class

```
class ExampleSite < ActiveSupport::Base
  self.site = "http://www.example.com"
end
```

- Consuming our users and photos

```
class User < ExampleSite; end
class Photo < ExampleSite; end
```

RESTful Tips

DRY RESTful Controllers

- resource_controller plugin

```
class PostsController < ResourceController::Base  
end
```

http://jamesgolick.com/resource_controller/rdoc/index.html

RESTful Tips

Turn Verbs into Nouns

Replace RPC* style overloaded POST interfaces with resources that respond to a Uniform Interface by using standard HTTP verbs

Method	Overloaded POST
User.find(1).activated?	GET users/1?q=activated
User.find(1).activate!	POST users/1?m=activate
User.find(1).deactivate!	POST users/1?m=deactivate

* Remote Procedure Call

RESTful Tips

Turn Verbs into Nouns

Replace RPC* style overloaded POST interfaces with resources that respond to a Uniform Interface by using standard HTTP verbs

Method	Uniform Interface
<code>User.find(1).activated?</code>	GET <code>users/1/activation</code>
<code>User.find(1).activate!</code>	POST <code>users/1/activation</code>
<code>User.find(1).deactivate!</code>	DELETE <code>users/1/activation</code>

* Remote Procedure Call

RESTful Tips

Custom MIME Types

```
Mime::Type.register "image/png", :png
```

```
Mime::Type.register "text/x-vcard", :vcard
```

RESTful Resources

RESTful Web Services

<http://www.oreilly.com/catalog/9780596529260/>



O'REILLY

Leonard Richardson & Sam Ruby
Foreword by David Heinemeier Hansson

Peepcode: REST Basics

[http://nubyonrails.com/
articles/peepcode-rest-basics](http://nubyonrails.com/articles/peepcode-rest-basics)

Resource Routes (in config/routes.rb)

```
Sample
nap.resources :users, :sessions

Nested
nap.resources :teams do |teams|
  teams.resources :players
end

Customized
nap.resources :articles,
  :collection => {:sort => :put},
  :member => {:deactivate => :delete},
  :new => {:preview => :post},
  :controller => 'articles',
  :singular => 'article',
  :path_prefix => '/book/:book_id',
  :name_prefix => 'book.'
```

REST Screencast \$9 at <http://peepcode.com>
85 minutes of RESTful goodness

Standard Methods	Verb	Path	Action	Formats in the URL	Path
plural_path	GET	/teams	index	formatted_plural_path(:xml)	/teams.xml
singular_path(id)	GET	/teams/1	show	formatted_singular_path(id, :rss)	/teams/1.rss
new_singular_path	GET	/teams/new	new	formatted_players_path(@team, :atom)	/teams/1/players.atom
plural_path	POST	/teams	create	formatted_player_path(@team, @player, :js)	/teams/1/players/5.js
edit_singular_path(id)	GET	/teams/1/edit	edit	formatted_player_path(:team_id => 1, :id => 5, :format => :js)	/teams/1/players/5.js
singular_path(id)	PUT	/teams/1	update		
singular_path(id)	DELETE	/teams/1	destroy		

Each method also has a counterpart ending in _url that includes the protocol, domain, and port. There is also a `hash_for` version of each method that returns a hash instead of a string.

```
button_to "Destroy", team_path(@team), :confirm => "Are you sure?", :method => :delete
link_to "Destroy", team_path(@team), :confirm => "Are you sure?", :method => :delete
link_to_remote "Destroy", :url => team_path(@team), :confirm => "Are you sure?", :method => :delete
form_for :team, @team, :url => team_path(@team), :html => { :method => :put } do |f| ...
```

Nested Resources	Path	Useful Plugins
players_path(@team)	/teams/:team_id/players /teams/1/players	Beast Forum (an app built with RESTful design) RESTful Authentication Plugin Simply Helpful Plugin
player_path(@team, @player)	/teams/:team_id/players/:id /teams/1/players/5	

Nested resources must be defined in `routes.rb`. See above for an example.

Custom Methods	Path	Action	Map Options
sort_tags_path	/tags;sort	sort	:collection => {:sort => :put}
deactivate_tag_path(id)	/tag/1;deactivate	deactivate	:member => {:deactivate => :delete}
preview_new_tag_path	/tags/new;preview	preview	:new => {:preview => :post}
tags_path(book_id)	/book/:book_id/tags	-	:path_prefix => "/book/:book_id"
tag_path(book_id, id)	/book/:book_id/tags/:id	-	(You get this for free with nested resources)
book_tags_path(book_id)	Usually used in a nested block or with a path_prefix	-	:name_prefix => "book."
book_tag_path(book_id, id)			(Should be used with a :path_prefix or in a nested resource declaration)
book_new_tag_path(book_id)			
book_deactivate_tag_path(book_id, id)			

Repeated resource names in `routes.rb` will override previous declarations. Use `name_prefix` to preserve dynamic method names for multiple declarations of the same resource.

```
respond_to { |wants| wants.all | .text | .html | .js | .ics | .xml | .rss | .atom | .yaml }
```

Add New MIME types (in config/environment.rb)

```
Mime::Type.register "image/jpg", :jpg
Mime::Type.register "application/vnd.visa+xml", :visa
```

Types listed here can be used in a `respond_to` block and as a forced format extension at the end of URLs.

Scaffold Resource Generator

```
./script/generate scaffold resource  
Epsilon  
title:string  
description:text  
program_id:integer
```

Use a singular word for the model/resource name. The other arguments will be used to pre-populate the database migration and fields in view templates.

Fielding Dissertation: Chapter 5

[http://www.ics.uci.edu/
~fielding/pubs/dissertation/
rest_arch_style.htm](http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm)

CHAPTER 5

Representational State Transfer (REST)

This chapter introduces and elaborates the Representational State Transfer (REST) architectural style for distributed hypermedia systems, describing the software engineering principles guiding REST and the interaction constraints chosen to retain those principles, while contrasting them to the constraints of other architectural styles. REST is a hybrid style derived from several of the network-based architectural styles described in Chapter 3 and combined with additional constraints that define a uniform connector interface. The software architecture framework of Chapter 1 is used to define the architectural elements of REST and examine sample process, connector, and data views of prototypical architectures.

5.1 Deriving REST

The design rationale behind the Web architecture can be described by an architectural style consisting of the set of constraints applied to elements within the architecture. By examining the impact of each constraint as it is added to the evolving style, we can identify the properties induced by the Web's constraints. Additional constraints can then be applied to form a new architectural style that better reflects the desired properties of a modern Web architecture. This section provides a general overview of REST by walking through the process of deriving it as an architectural style. Later sections will describe in more detail the specific constraints that compose the REST style.

5.1.1 Starting with the Null Style

There are two common perspectives on the process of architectural design, whether it be for buildings or for software. The first is that a designer starts with nothing—a blank slate, whiteboard, or drawing board—and builds-up an architecture from familiar components until it satisfies the needs of the intended system. The second is that a designer starts with the system needs as a whole, without constraints, and then incrementally identifies and applies constraints to elements of the system in order to differentiate the design space and allow the forces that influence system behavior to flow naturally, in harmony with the system. Where the first emphasizes creativity and unbounded vision, the second emphasizes restraint and understanding of the system context. REST has been developed using the latter process. Figures 5-1 through 5-8 depict this graphically in terms of how the applied constraints would differentiate the process view of an architecture as the incremental set of constraints is applied.

The Null style ([Figure 5-1](#)) is simply an empty set of constraints. From an architectural perspective, the null style describes a system in which there are no distinguished boundaries between components. It is the starting point for our description of REST.



REST is a hybrid style derived from several of the network-based architectural styles described in Chapter 3 and combined with additional constraints that define a uniform connector interface. The software architecture framework of Chapter 1 is used to define the architectural elements of REST and examine sample process, connector, and data views of prototypical architectures.

These Slides
[http://reinh.com/2007/11/13/
rest-rails](http://reinh.com/2007/11/13/rest-rails)

ReinH

REST & Rails

Posted by ReinH

I am giving a presentation on Rails, REST, and the Resource Oriented Architecture today entitled "REST & Rails: Web Services for the Rails World". Check out the slides.



Sections: Rails
Tags: rest rails
Meta: permalink

Comments

[Leave a response](#)

ReinH

ReinH

ReinH

ReinH



“My God, it’s full of resources!”