Mastering Grails: RESTful Grails

Build a resource-oriented architecture

Skill Level: Introductory

Scott Davis (scott@aboutgroovy.com)
Editor in Chief
AboutGroovy.com

16 Sep 2008

We live in the era of mashups. Creating Web pages that give users the information they want is a good start, but offering a source of raw data that other Web developers can easily mix in with their own applications is better. In this installment of *Mastering Grails*, Scott Davis introduces various ways to get Grails to produce XML instead of the usual HTML.

This month, I'm going to show how your Grails applications can be a source of raw data — specifically, XML — that other Web applications can use. I'd normally describe this as setting up Web services for your Grails application, but the term is loaded with hidden meanings these days. Many people associate Web services with SOAP and a full-blown service-oriented architecture (SOA). Two plug-ins for Grails allow you to expose a SOAP interface to your application if you choose that route (see Resources). But rather than deal with a specific implementation such as SOAP, I'll show you how to return Plain Old XML (POX) using an interface based on Representational State Transfer (REST).

When it comes to RESTful Web services, it's as important to understand the *why* as much as the *how*. Roy Fielding's doctoral dissertation (see Resources) — the source of the REST acronym — outlines two approaches to Web services: one that's *service-oriented* and another that's *resource-oriented*. Before I show you the code to implement your own RESTful resource-oriented architecture (ROA), I'll clarify the differences between these two design philosophies, and I'll discuss the two competing definitions of REST in popular usage. As your reward for taking in all the talk in the first part of this article, you'll find plenty of Grails code later on.

REST, in simple terms

When developers talk about offering RESTful Web services, they generally mean that they want to provide a simple, no-hassle way of getting XML out of their application. RESTful Web services typically provide a URL that returns XML in response to an HTTP `GET` request. (I'll give you a more formal definition of REST in just a moment that refines this definition in a subtle but important way.)

Yahoo! provides a number of RESTful Web services (see Resources) that return POX in response to a simple HTTP `GET` request. For example, enter `http://api.search.yahoo.com/WebSearchService/V1/webSearch?appid=YahooDemo` into your Web browser's location field. You get the same Web search results in XML that you'd normally get in HTML by entering `beatles` in the search box on the Yahoo! home page.

About this series

Grails is a modern Web development framework that mixes familiar Java™ technologies like Spring and Hibernate with contemporary practices like convention over configuration. Written in Groovy, Grails give you seamless integration with your legacy Java code while adding the flexibility and dynamism of a scripting language. After you learn Grails, you'll never look at Web development the same way again.

If, hypothetically, Yahoo! supported a SOAP interface (it doesn't), making a SOAP request would return the same data, but it would involve a bit more effort on the part of the developer making the request. The requester would need to submit a well-defined XML document with a SOAP header and body section instead of a simple set of name/value pairs in the query string — and submit the request using an HTTP `POST` instead of a `GET`. After all of that additional work, the response would come back in a formal XML document that, like the request, would have a SOAP header and body section that would need to be stripped away in order to get to the query results. The RESTful approach to Web services is generally embraced as a "low ceremony" alternative to SOAP's complexity.

Several trends indicate that the RESTful approach to Web services is gaining popularity. Amazon.com provides both RESTful and SOAP-based services. Real-world usage patterns indicate that users prefer the RESTful interface almost nine to one. In another notable case, Google officially deprecated its SOAP-based Web services in December 2006. All of its data services (grouped under the Google Data APIs umbrella) embrace a more RESTful approach.

Service-oriented Web services

If the difference between REST and SOAP boiled down to the relative merits of `GET` and `POST`, the distinction would be simple. The HTTP method used is important, but for a different reason than you might initially expect. To understand the difference between REST and SOAP fully, you need to appreciate the deeper semantics of these two strategies. SOAP embodies a service-oriented approach to Web services — one in which methods (or verbs) are the primary way you interact with the service. REST takes a resource-oriented approach in which the object (or the noun) takes center stage.

In an SOA, a service call looks like a remote procedure call (RPC). Hypothetically, if you have a Java `Weather` class with a `getForecast(String zipcode)` method, you can easily expose that method as a Web service. In fact, Yahoo! has a Web service that does just that. Type `http://weather.yahooapis.com/forecastrss?p=94089` into your browser, substituting your own ZIP code for the p parameter. The Yahoo! service supports a second parameter — u — that accepts either `f` for Fahrenheit or c for Celsius. It's not hard to imagine overloading the method signature on your hypothetical class to accept the second parameter: `getForecast("94089", "f")`.

Looking back at the Yahoo! search query I made earlier, it's not hard to imagine rewriting it as a method call, either. http://api.search.yahoo.com/WebSearchService/V1/webSearch?appid=YahooDemo&query=beatles translates easily into `WebSearchService.webSearch("YahooDemo", "beatles")`.

So if the Yahoo! calls are, in fact, RPC calls, doesn't that contradict my earlier assertion that the Yahoo! services are RESTful? Unfortunately, yes. But I'm not alone in my error. Yahoo! also calls these services RESTful, although to be fair, it recognizes that they don't meet the strictest sense of the definition. In the Yahoo! Web Services FAQ entry for "What is REST?", the answer is, "REST stands for Representational State Transfer. Most of the Yahoo! Web Services use 'REST-Like' RPC-style operations over HTTP `GET` or `POST`..."

This is an ongoing debate in the REST community. The problem is that no popular catchphrase succinctly describes "RPC-based Web services that favor HTTP `GET` over `POST` and simple URL requests over XML requests." Some folks call these HTTP/POX or REST/RPC services. Others call them Low REST to contrast them with High REST Web services — services that are closer to Fielding's original definition of a resource-oriented architecture.

Informally, I call services like Yahoo!'s *GETful*. This isn't meant to be pejorative — on the contrary, I think that Yahoo! has done a great job of putting together a collection of low-ceremony Web services. The word captures the key benefit of Yahoo!'s RPC-style services — getting XML results back by making a simple HTTP `GET` request — without misusing Fielding's original definition.

Resource-oriented Web services

POST vs. PUT

There's debate in the REST community over the roles of POST and PUT in inserting new resources. The definition of PUT in the original RFC for HTTP 1.1 (Fielding was the lead author) says if the resource doesn't exist, the server can create it. And that if the resource already exists, "...the enclosed entity SHOULD be considered as a modified version of the one residing on the origin server." So, if the resource doesn't exist, PUT equals INSERT. If the resource exists, PUT equals UPDATE. That would be simple enough if the definition of POST didn't say:

"POST is designed to allow a uniform method to cover the following functions:

- Annotation of existing resources;
- Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles;
- Providing a block of data, such as the result of submitting a form, to a data-handling process;
- Extending a database through an append operation."

"Annotation of existing resources" seems to imply UPDATE, and "posting a message to a bulletin board" and "extending a database" seem to imply INSERT.

Given the fact that all major browsers lack support for the PUT method when submitting HTML form data (they support only GET and POST), it's difficult to determine what the wisdom of crowds has to say about which method should be used in which case.

The Atom Publishing Protocol is a popular syndication format that follows RESTful principles. The authors of the RFC for Atom try to end the POST vs. PUT debate definitively:

"The Atom Publishing Protocol uses HTTP methods to author Member Resources as follows:

- GET is used to retrieve a representation of a known Resource.
- POST is used to create a new, dynamically named, Resource...
- PUT is used to edit a known Resource. It is not used for Resource creation.
- DELETE is used to remove a known Resource."

I use Atom as my guide in this article, using POST for INSERT and PUT for UPDATE. However, if you choose to reverse these methods in your application, you're in good company as well — the book *RESTful Web Services* (see Resources) advocates using PUT for

INSERTs.

So what does it take for a service to be truly resource-oriented? It all boils down to creating a good Uniform Resource Identifier (URI) and using four HTTP verbs (`GET`, `POST`, `PUT`, and `DELETE`) in a standardized way instead of one (`GET`) in conjunction with your custom method call.

Going back to the Beatles query, the first step toward a more formal RESTful interface is to adjust the URI. Rather than passing in `Beatles` as an argument to the `webSearch` method, `Beatles` becomes the resource at the center of the URI. For example, the URI for the Wikipedia article on the Beatles is http://en.wikipedia.org/wiki/Beatles.

But what truly differentiates the GETful from the RESTful philosophy is the method used to return a representation of the resource. The Yahoo! RPC interface defines a number of custom methods (`webSearch`, `albumSearch`, `newsSearch`, and so on). There's no way to know what the method call's name is without reading the documentation. In Yahoo!'s case, I can follow the pattern and guess that there are `songSearch`, `imageSearch`, and `videoSearch` method calls as well, but there's no guarantee. Also, other Web sites might use a different naming convention, such as `findSong` or `songQuery`. In the case of Grails, custom actions like `aiport/list` and `airport/show` are standard throughout the application, but these method names are by no means standard across other Web frameworks.

In contrast, the RESTful approach always uses an HTTP `GET` to return a representation of the resource in question. So for any resource on Wikipedia (http://en.wikipedia.org/wiki/Beatles, http://en.wikipedia.org/wiki/United_Airlines, or http://en.wikipedia.org/wiki/Peanut_butter_and_jelly_sandwich), I know that a `GET` is the standard way to get my hands on it.

The power of standardized method calls becomes more evident when you're dealing with a resource's full Create/Retrieve/Update/Delete (CRUD) life cycle. An RPC interface offers no standardized way to create a new resource. The custom method call could be `create`, `new`, `insert`, `add`, or almost anything else. In a RESTful interface, sending a `POST` request to a URI inserts a new resource. `PUT` updates the resource, and `DELETE` deletes the resource (see the POST vs. PUT sidebar).

Now that you have a better understanding of the distinction between GETful and RESTful Web services, you are ready to get started on creating your own in Grails. You'll see examples of both, but I'll start with simple POX example.

Implementing a GETful Web service in Grails

The quickest way to get POX out of your Grails application is to import the

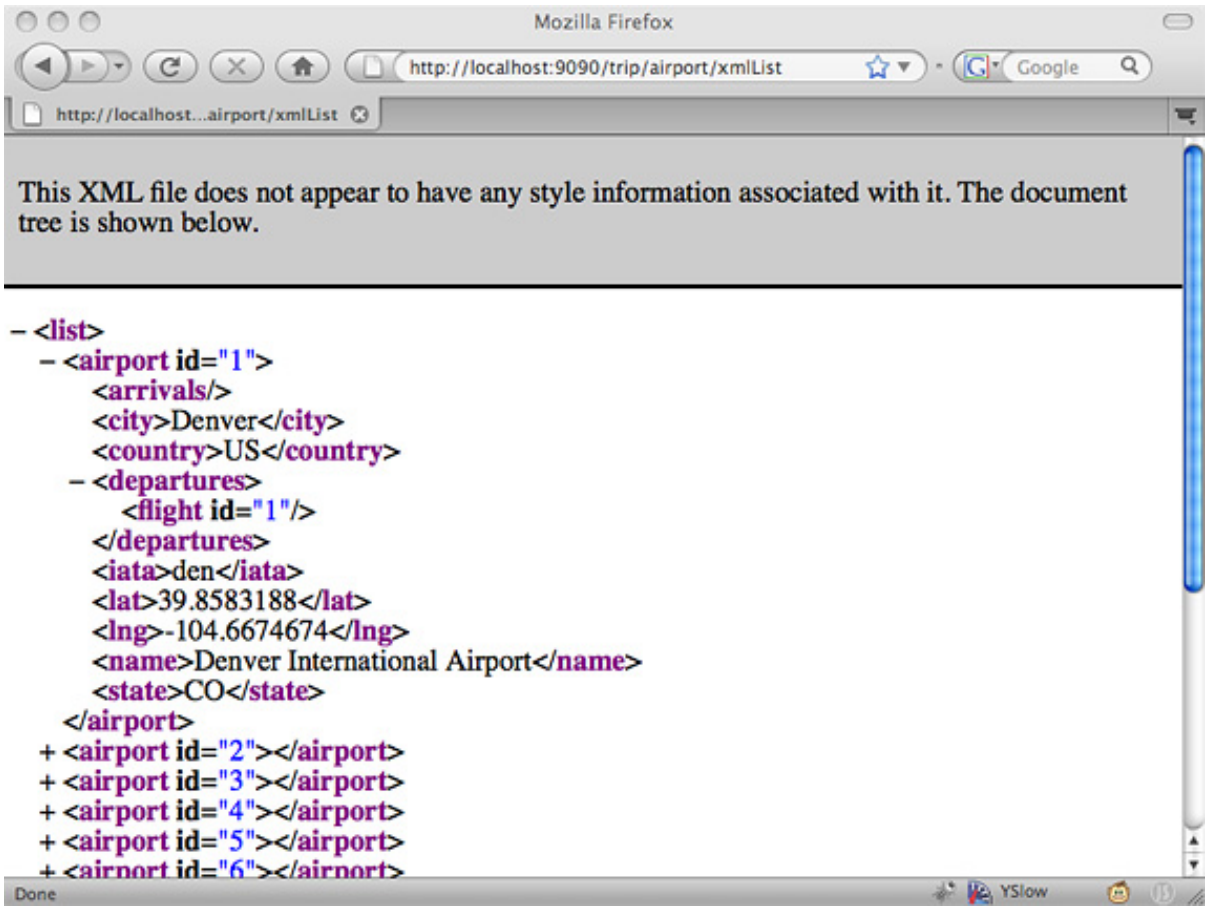
`grails.converters.*` package and add a couple of new closures, as shown in Listing 1:

Listing 1. Simple XML output

```
import grails.converters.*

class AirportController{
  def xmlList = {
    render Airport.list() as XML
  }

  def xmlShow = {
    render Airport.get(params.id) as XML
  }

  //... the rest of the controller
}
```

You saw the `grails.converters` package in action in "Many-to-many relationships with a dollop of Ajax." This package gives you incredibly simple JavaScript Object Notation (JSON) and XML output support. Figure 1 shows the results of a call to the `xmlList` action:

Figure 1. Default XML output from Grails



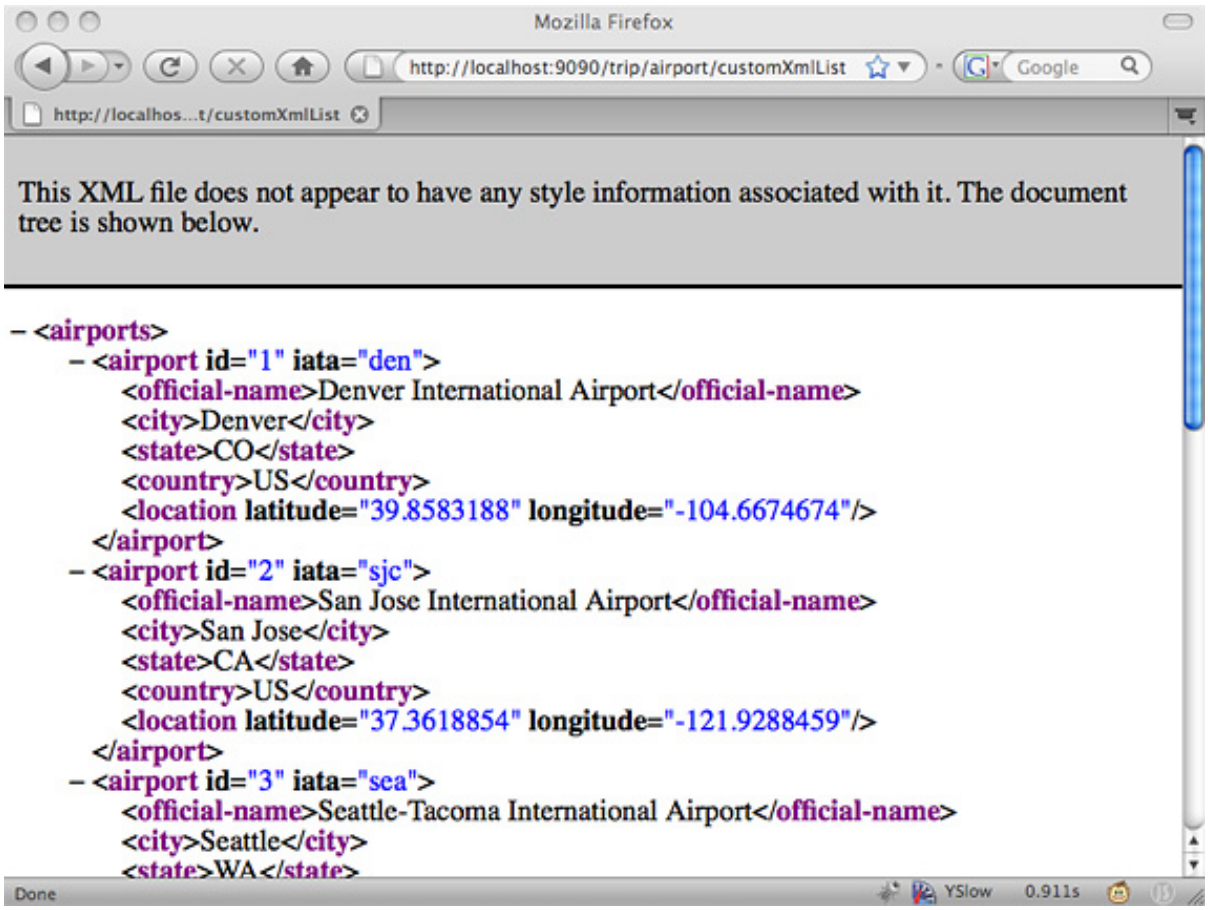
Although the default XML output is fine for debugging, chances are you'll want to customize the format a bit. Thankfully, the `render()` method offers you a Groovy `MarkupBuilder` that allows you to define custom XML on the fly (see Resources for a link to more information on `MarkupBuilder`). Listing 2 creates some custom XML output:

Listing 2. Custom XML output

```
def customXmlList = {
  def list = Airport.list()
  render(contentType:"text/xml"){
    airports{
      for(a in list){
        airport(id:a.id, iata:a.iata){
          "official-name"(a.name)
          city(a.city)
          state(a.state)
          country(a.country)
          location(latitude:a.lat, longitude:a.lng)
        }
      }
    }
  }
}
```

Figure 2 shows the resulting output:

Figure 2. Custom XML output using the Groovy MarkupBuilder



Notice how closely the source code corresponds to the XML output. You can define any element name you'd like (airports, airport, city) regardless of whether or not they correspond to the class's actual field name. If you want to provide a hyphenated element name (such as `official-name`) or add namespace support, simply surround the element name in quotes. Attributes (such as `id` and `iata`) are defined using Groovy hashmap *key*:*value* syntax. To populate an element's body, supply a value without the *key*:.

Content negotiation and the Accept header

Creating separate closures that return HTML and XML representations of the data is simple enough, but what if you want one closure that can handle both? This is possible, thanks to the `Accept` header included in the HTTP request. This little bit of metadata in the request says to the server, "Hey, you might have more than one representation of the resource at this URI — here's my preference."

cURL is a handy open source command-line HTTP tool (see Resources). Type `curl http://localhost:9090/trip/airport/list` at the command line to

simulate a browser request for the list of airports. You should see the HTML response scroll by on your screen.

Now, make two tiny tweaks to the request. This time, make a `HEAD` request instead of a `GET`. `HEAD` is a standard HTTP method that returns just the metadata for the response, not the body. (It is included in the HTTP specification for exactly the type of debugging you are doing here.) Also, put cURL in `verbose` mode so that you can see the request metadata as well, as shown in Listing 3:

Listing 3. Using cURL for HTTP debugging

```
$ curl --request HEAD --verbose http://localhost:9090/trip/airport/list
* About to connect() to localhost port 9090 (#0)
*   Trying ::1... connected
* Connected to localhost (::1) port 9090 (#0)
> HEAD /trip/airport/list HTTP/1.1
> User-Agent: curl/7.16.3 (powerpc-apple-darwin9.0)
       libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3
> Host: localhost:9090
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Language: en-US
< Content-Type: text/html; charset=utf-8
< Content-Length: 0
< Server: Jetty(6.1.4)
<
* Connection #0 to host localhost left intact
* Closing connection #0
```

Notice the `Accept` header in the request. When a client submits `*/*`, it is essentially saying, "I don't care what format you send back. I'll accept anything."

cURL allows you to override this value with the `--header` parameter. Type `curl --request HEAD --verbose --header Accept:text/xml http://localhost:9090/trip/airport/list` and verify that the `Accept` header is now asking for `text/xml`. This is the MIME type of the resource.

So, how does Grails respond to the `Accept` header on the server side? Add another closure to the `AirportController`, as shown in Listing 4:

Listing 4. The debugAccept action

```
def debugAccept = {
  def clientRequest = request.getHeader("accept")
  def serverResponse = request.format
  render "Client: ${clientRequest}\nServer: ${serverResponse}\n"
}
```

The first line in Listing 4 retrieves the `Accept` header from the request. The second line shows how Grails interprets the request and the response it will send back.

Now use cURL to do some exploring, as shown in Listing 5:

Listing 5. Adjusting the Accept header in cURL

```
$ curl  http://localhost:9090/trip/airport/debugAccept
Client: */*
Server: all

$ curl  --header Accept:text/xml http://localhost:9090/trip/airport/debugAccept
Client: text/xml
Server: xml
```

Where do the `all` and `xml` values come from? Take a look at grails-app/conf/Config.groovy. Near the top of the file, you should see a hashmap that uses simple names for the keys (names like `all` and `xml`) and the MIME types that correspond to them for the values. Listing 6 shows the `grails.mime.types` hashmap:

Listing 6. The grails.mime.types hashmap in Config.groovy

```
grails.mime.types = [ html: ['text/html','application/xhtml+xml'],
                      xml: ['text/xml', 'application/xml'],
                      text: 'text-plain',
                      js: 'text/javascript',
                      rss: 'application/rss+xml',
                      atom: 'application/atom+xml',
                      css: 'text/css',
                      csv: 'text/csv',
                      all: '*/*',
                      json: ['application/json','text/json'],
                      form: 'application/x-www-form-urlencoded',
                      multipartForm: 'multipart/form-data'
                    ]
```

Advanced content negotiation

The `Accept` header supplied by the typical Web browser is a bit more complicated than the simple one you used with cURL. For example, an `Accept` header from Firefox 3.0.1 on Mac OS X 10.5.4 looks like this:

```
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
```

It is a comma-separated list with optional q attributes to favor certain MIME types over others. (The q value — short for quality — is a `float` value ranging from 0.0 to 1.0.) Because `application/xml` is given a q value of 0.9, Firefox prefers XML data over anything else.

Here is the accept header from Safari version 3.1.2 on Mac OS X 10.5.4:

```
text/xml,application/xml,application/xhtml+xml,
```

```
text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
```

The `text/html` MIME type is given a q value of 0.9, so HTML is the preferred output type, followed by `text/plain` at 0.8 and `*/*` at 0.5.

See Resources for more information on server-side content negotiation.

So, now that you know a little bit more about content negotiation, you can add a `withFormat` block to the `list` action to return the proper data type based on the `Accept` header in the request, as shown in Listing 7:

Listing 7. Using the withFormat block in an action

```
def list = {
  if(!params.max) params.max = 10
  def list = Airport.list(params)
  withFormat{
    html{
      return [airportList:list]
    }
    xml{
      render list as XML
    }
  }
}
```

The last line of each format block must be a `render`, `return`, or `redirect` — no different from a normal action. If the `Accept` header resolves to "all" (`*/*`), the first entry in the block is used.

Tweaking the `Accept` header in cURL is fine, but you can also do some testing by tweaking the URI. Both http://localhost:8080/trip/airport/list.xml and http://localhost:8080/trip/airport/list?format=xml are ways that you can override the `Accept` header explicitly. Play around with `cURL` and the various URI overrides to ensure that your `withFormat` block is working as expected.

If you want to make this behavior standard in Grails, don't forget that you can type in `grails install-templates` and edit the files in /src/templates.

Now that you have all of the basic building blocks in place, the final piece of the puzzle is to move from a GETful interface to a truly RESTful one.

Implementing a RESTful Web service in Grails

To begin, you need to ensure that your controller starts responding to the four basic HTTP methods. Recall that the `index` closure is the entry point into the controller if

the user doesn't specify a custom action like `list` or `show`. By default, `index` redirects to the `list` action: `def index = { redirect(action:list,params:params) }`. Replace that code with the code in Listing 8:

Listing 8. Switching on the HTTP method

```
def index = {
  switch(request.method){
    case "POST":
      render "Create\n"
      break
    case "GET":
      render "Retrieve\n"
      break
    case "PUT":
      render "Update\n"
      break
    case "DELETE":
      render "Delete\n"
      break
  }
}
```

Use cURL, as shown in Listing 9, to verify that the `switch` statement is working correctly:

Listing 9. Using cURL for all four HTTP methods

```
$ curl --request POST http://localhost:9090/trip/airport
Create
$ curl --request GET http://localhost:9090/trip/airport
Retrieve
$ curl --request PUT http://localhost:9090/trip/airport
Update
$ curl --request DELETE http://localhost:9090/trip/airport
Delete
```

Implementing GET

Because you already know how to return XML, implementing the `GET` method should be pretty easy. There's one hitch, though. A `GET` request to http://localhost:9090/trip/airport should return a list of airports. A `GET` request to http://localhost:9090/trip/airport/den should return an instance of the airport whose IATA code is den. To accomplish this, you should set up a custom URL mapping.

Open grails-app/conf/UrlMappings.groovy in a text editor. The default `/$controller/$action?/$id?` mapping should look familiar. The URL http://localhost:9090/trip/airport/show/1 maps to the `AiportController` and `show` action, and the `params.id` value is set to `1`. The trailing question mark after action and ID means that the URL element is optional.

Add a line to the `static mappings` block that maps RESTful requests back to the `AirportController`, as shown in Listing 10. I've hard-coded the controller in for now because you haven't implemented REST support in the other controllers. Later on, you should probably replace the `airport` part of the URL with `$controller`.

Listing 10. Creating a custom URL mapping

```
class UrlMappings {
    static mappings = {
      "/$controller/$action?/$id?"{
         constraints { // apply constraints here
         }
        }
        "/rest/airport/$iata?"(controller:"airport",action:"index")
     "500"(view:'/error')
   }
}
```

This mapping ensures that all URIs starting with /rest are routed to the `index` action. (This eliminates the need to deal with content negotiation.) It also means that you can check for the presence of `params.iata` to decide if you should return a list or an instance.

Modify the index action as shown in Listing 11:

Listing 11. Returning XML from an HTTP GET

```
def index = {
  switch(request.method){
    case "POST":   //...
    case "GET":
      if(params.iata){render Airport.findByIata(params.iata) as XML}
      else{render Airport.list() as XML}
      break
    case "PUT":    //...
    case "DELETE": //...
  }
}
```

Type `http://localhost:9090/trip/rest/airport` and `http://localhost:9090/trip/rest/airport/den` into your Web browser to confirm that the custom URL mapping is in place.

Custom URL mapping by HTTP method

You could take a slightly different approach to setting up a RESTful URL mapping. You can route requests to a specific action based on the HTTP request. For example, this is how you would map `GET`, `PUT`, `POST`, and `DELETE` to the corresponding Grails actions that already exist:

```
static mappings = {
   "/airport/$id"(controller:"airport"){
```

```
        action = [GET:"show", PUT:"update", DELETE:"delete", POST:"save"]
    }
}
```

Implementing DELETE

Adding support for `DELETE` is not much different from `GET`. In this case, though, I only want to allow individual airports to be deleted by IATA code. If a user submits an HTTP `DELETE` without the IATA code, I'll return a 400 HTTP status code, `Bad Request`. If the user submits an IATA code that cannot be found, I'll return the ever-popular 404 status code `Not Found`. Only if the delete is successful will I return the standard `200 OK`. (See Resources for a link to more information on HTTP status codes.)

Add the code in Listing 12 to the `DELETE case` in the `index` action:

Listing 12. Reacting to an HTTP DELETE

```
def index = {
  switch(request.method){
    case "POST": //...
    case "GET":  //...
    case "PUT":  //...
    case "DELETE":
      if(params.iata){
        def airport = Airport.findByIata(params.iata)
        if(airport){
          airport.delete()
          render "Successfully Deleted."
        }
        else{
          response.status = 404 //Not Found
          render "${params.iata} not found."
        }
      }
      else{
        response.status = 400 //Bad Request
        render """DELETE request must include the IATA code
                  Example: /rest/airport/iata
        """
      }
      break
  }
}
```

First, try deleting an airport that you know exists, as shown in Listing 13:

Listing 13. Deleting a good airport

```
Deleting a Good Airport</heading>
$ curl --verbose --request DELETE http://localhost:9090/trip/rest/airport/lga
> DELETE /trip/rest/airport/lga HTTP/1.1
< HTTP/1.1 200 OK
Successfully Deleted.
```

Then, try deleting an airport that you know doesn't exist, as shown in Listing 14:

Listing 14. Trying to DELETE an airport that doesn't exist

```
$ curl --verbose --request DELETE http://localhost:9090/trip/rest/airport/foo
> DELETE /trip/rest/airport/foo HTTP/1.1
< HTTP/1.1 404 Not Found
foo not found.
```

Finally, try making the `DELETE` request without supplying an IATA code, as shown in Listing 15:

Listing 15. Trying to DELETE all airports at once

```
$ curl --verbose --request DELETE http://localhost:9090/trip/rest/airport
> DELETE /trip/rest/airport HTTP/1.1
< HTTP/1.1 400 Bad Request
DELETE request must include the IATA code
Example: /rest/airport/iata
```

Implementing POST

Moving on, inserting a new `Airport` is your next goal. Create a file named simpleAirport.xml, shown in Listing 16:

Listing 16. simpleAirport.xml

```
<airport>
  <iata>oma</iata>
  <name>Eppley Airfield</name>
  <city>Omaha</city>
  <state>NE</state>
  <country>US</country>
  <lat>41.3019419</lat>
  <lng>-95.8939015</lng>
</airport>
```

If the XML representation of the resource is flat (no deep nesting of elements) and each element name corresponds to a field name in the class, Grails can construct the new class directly from the XML. The root element of the XML document is addressable via `params`, as shown in Listing 17:

Listing 17. Reacting to an HTTP POST

```
def index = {
  switch(request.method){
    case "POST":
      def airport = new Airport(params.airport)
```

```
      if(airport.save()){
        response.status = 201 // Created
        render airport as XML
      }
      else{
        response.status = 500 //Internal Server Error
        render "Could not create new Airport due to errors:\n ${airport.errors}"
      }
      break
    case "GET":    //...
    case "PUT":    //...
    case "DELETE": //...
  }
}
```

The XML needs to be flat because `params.airport` is, in fact, a hashmap. (Grails does the XML-to-hashmap transformation for you behind the scenes.) This means that effectively you are using the named arguments constructor for `Airport` — `def airport = new Airport(iata:"oma", city:"Omaha", state:"NE")`.

To test the new code, use cURL to POST the simpleAirport.xml file, as shown in Listing 18:

Listing 18. Using cURL to make an HTTP POST

```
$ curl --verbose --request POST --header "Content-Type: text/xml" --data
      @simpleAirport.xml http://localhost:9090/trip/rest/airport
> POST /trip/rest/airport HTTP/1.1
> Content-Type: text/xml
> Content-Length: 176
>
< HTTP/1.1 201 Created
< Content-Type: text/xml; charset=utf-8
<?xml version="1.0" encoding="utf-8"?><airport id="14">
  <arrivals>
    <null/>
  </arrivals>
  <city>Omaha</city>
  <country>US</country>
  <departures>
    <null/>
  </departures>
  <iata>oma</iata>
  <lat>41.3019419</lat>
  <lng>-95.8939015</lng>
  <name>Eppley Airfield</name>
  <state>NE</state>
</airport>
```

If the XML is more complex, you need to unmarshall it by hand. For example, remember the custom XML format you defined earlier? Create a file named newAirport.xml, shown in Listing 19:

Listing 19. newAirport.xml

```
<airport iata="oma">
  <official-name>Eppley Airfield</official-name>
  <city>Omaha</city>
  <state>NE</state>
  <country>US</country>
  <location latitude="41.3019419" longitude="-95.8939015"/>
</airport>
```

Now, in the `index` action, replace the single line `def airport = new Airport(params.airport)` with the code in Listing 20:

Listing 20. Parsing complex XML

```
def airport = new Airport()
airport.iata = request.XML.@iata
airport.name = request.XML."official-name"
airport.city = request.XML.city
airport.state = request.XML.state
airport.country = request.XML.country
airport.lat = request.XML.location.@latitude
airport.lng = request.XML.location.@longitude
```

The `request.XML` object is a `groovy.util.XmlSlurper` that holds the raw XML. It is effectively the root element, so you can ask for child elements by name (`request.XML.city`). If the name is hyphenated or namespaced, surround it in quotes (`request.XML."official-name"`). To get access to an element's attributes, use the @ symbol (`request.XML.location.@latitude`). (See Resources for a link to more information on `XmlSlurper`.)

Finally, test it using cURL: `curl --request POST --header "Content-Type: text/xml" --data @newAirport.xml http://localhost:9090/trip/rest/airport`.

Implementing PUT

The last HTTP method you need to support is `PUT`. Knowing what you know from the `POST` exercise, this code is virtually identical. The only difference is that rather than constructing the class directly from the XML, you need to ask GORM for the existing class. The line `airport.properties = params.airport` then replaces the existing field data with the new data from the XML, as shown in Listing 21:

Listing 21. Reacting to an HTTP PUT

```
def index = {
  switch(request.method){
    case "POST":  //...
    case "GET":   //...
    case "PUT":
      def airport = Airport.findByIata(params.airport.iata)
      airport.properties = params.airport
      if(airport.save()){
        response.status = 200 // OK
```

```
        render airport as XML
      }
      else{
        response.status = 500 //Internal Server Error
        render "Could not create new Airport due to errors:\n ${airport.errors}"
      }
      break
    case "DELETE": //...
  }
}
```

Create a file named editAirport.xml, shown in Listing 22:

Listing 22. editAirport.xml

```
<airport>
  <iata>oma</iata>
  <name>xxxEppley Airfield</name>
  <city>Omaha</city>
  <state>NE</state>
  <country>US</country>
  <lat>41.3019419</lat>
  <lng>-95.8939015</lng>
</airport>
```

Finally, test it using cURL: `curl --verbose --request PUT --header "Content-Type: text/xml" --data @editAirport.xml http://localhost:9090/trip/rest/airport`.

Conclusion

I've covered a lot of ground in a short amount of time. You should now understand the difference between an SOA and an ROA. You should also recognize that not all RESTful Web services are the same. Some of them are GETful — using an HTTP `GET` request for RPC-like method calls. Others are truly resource-oriented, where the URI is the key accessing the resource, and the standard HTTP `GET`, `POST`, `PUT`, and `DELETE` methods give you full CRUD capabilities. Whether you prefer the GETful or RESTful approach, Grails offers robust support for outputting XML and effortlessly ingesting it back.

In the next *Mastering Grails* installment, I'll turn my attention to testing. Grails ships with great testing facilities out of the box. What it doesn't provide can be added in as a plug-in after the fact. With all of the time you've invested in Grails development at this point, you need to make sure that it starts out bug free and stays that way for the life of the application. Until then, keep mastering Grails.

Resources

Learn

- *Mastering Grails*: Read more in this series to gain a further understanding of Grails and all you can do with it.
- Grails: Visit the Grails Web site.
- Grails Framework Reference Documentation: The Grails bible.
- Architectural Styles and the Design of Network-based Software Architectures: (Roy Thomas Fielding, University of California at Irvine, 2000): Fielding's doctoral dissertation describing REST.
- Representational State Transfer: Wikipedia's REST article.
- Creating a REST Request for Yahoo! Search Web Services: To use Yahoo! Web services officially, you should get and use your own application key instead of using `YahooDemo`.
- Google Data APIs: Learn more about Google's Web services, which are based on the Atom Publishing Protocol.
- *RESTful Web Services* (Leonard Richardson and Sam Ruby, O'Reilly Media, 2007): This book's authors advocate using `PUT` for INSERTs.
- Creating XML using Groovy's MarkupBuilder and Reading XML using Groovy's XmlSlurper: Produce and consume XML with Groovy.
- Content Negotiation: Learn more about server-side content negotiation.
- Status Code Definitions: Get the scoop on HTTP status codes.
- "Write REST services" (J. Jeffrey Hanson, developerWorks, October 2007): Create REST services with Java technology and the Atom Publishing Protocol in this tutorial.
- "Build a RESTful Web service" (Andrew Glover, developerWorks, July 2008): This tutorial introduces REST and the Restlet framework.
- "*Crossing borders*: REST on Rails" (Bruce Tate, developerWorks, August 2006): Find out how Ruby on Rails supports RESTful Web services.
- *Groovy Recipes*: Learn more about Groovy and Grails in Scott Davis' latest book.
- *Practically Groovy*: This developerWorks series is dedicated to exploring the practical uses of Groovy and teaching you when and how to apply them successfully.
- Groovy: Learn more about Groovy at the project Web site.

- AboutGroovy.com: Keep up with the latest Groovy news and article links.
- Technology bookstore: Browse for books on these and other technical topics.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- Grails: Download the latest Grails release.
- XFire and Apache Axis2 Plugin: Grails plug-ins for exposing a SOAP interface.
- cURL : cURL is installed by default on most UNIX®, Linux®, and Mac OS X systems. You can download a version for Windows® and virtually every other OS.

Discuss

- Check out developerWorks blogs and get involved in the developerWorks community.

About the author

Scott Davis
Scott Davis is an internationally recognized author, speaker, and software developer. His books include *Groovy Recipes: Greasing the Wheels of Java*, *GIS for Web Developers: Adding Where to Your Application*, *The Google Maps API*, and *JBoss At Work*.

Trademarks