iRules: Simulating RESTful Behavior

Lori MacVittie, 2007-02-11

One of the premises of REST (Representational State Transfer) is that it is simpler to use well-known HTTP methods (PUT, DELETE, GET, POST) to perform actions upon resources than it is to construct complex SOAP or traditional HTTP-based application messages. REST resources are identified by URI (Uniform Resource Identifiers) that are specific to the resource.

For example, instead of retreiving information about a city with a URI something like this:

> http://www.example.com/getcityinformation.php?city=Madison&state=WI

you would use the GET HTTP method along with a URI that looks more like this:

> http://www.example.com/Madison/WI

You could also (ostensibly) use the PUT method to add a new city, the POST method to update an existing city, and the DELETE method to remove the resource entirely. All assuming you had permission to do so, of course. The problem is that many browsers do not support PUT and DELETE. Some even *gasp* fail to properly support POST, but in most instances you can rely upon GET and POST support at the very least.

Also problematic are the security concerns around allowing PUT and DELETE methods on a web or application server. For years we've warned and even threatened administrators with horror stories about the potential vulnerabilities that can be exposed by enabling these methods, so it's highly unlikely that you could convince that admin to enable them now.

But what if you really, ***really*** want to support REST-like URIs? All you need is a BIG-IP and some iRule fu.

Rewrite that Request

iRules can help here because it enables you to manipulate just about every aspect of an HTTP request, including rewriting the URI and transforming the actual request message.

This means you can use URIs like: http://www.example.com/put/Madison/WI or http://www.example.com/delete/Madison/WI in order to simulate REST-like behavior without encoding the resource values in hidden form fields. This is especially nice for AJAX-based applications which can rely more upon URI query parameters to perform actions than form fields.

To enable this transformation, encode the appropriate REST action and subsequent resource identifiers in the URI *path.*

> http://www.example.com/put/resource/identifier/

Use the POST method if you're sending along a message that will modify the resource using PUT (add) or POST (update). The following example shows a form whose action indicates it is a REST POST operation (update) and the resource identifier is a city-state combination, in this case Madison, WI. The actual request and its associated data could also be constructed dynamically using JavaScript and the XMLHTTPRequestObject. The important thing is to encode the REST-like parameters in the URI path in such a way that it can be easily deconstructed by an iRule and is **consistent** across the application.

> ```
> http://www.example.com/post/Madison/WI>
> 200000
> ```
>
> Update Population

Next, write an iRule to extract the method and resource identifiers from the URI *path,* making sure to (1) append the appropriate resource identifiers and parameter names within the **HTTP::payload** and (2) update **HTTP::uri** so your request is properly handled by the receiving server. This rule assumes a very simple application approach - each REST action corresponds to a PHP script bearing the same name as the action. It also assumes the client is using POST. We can parse out the parameters from a GET query string just as easily in an iRule, but I prefer POST because it alleviates any potential issues arising from very long URIs.

```
when HTTP_REQUEST {
# parse out the REST like parms to build a POST body
set pathvalues [split [HTTP::path] "/"]
set action [lindex $postvalues 1]
set city [lindex $postvalues 2]
set state [lindex $postvalues 3]


# create the new POST body to include the rewritten parameters
set payload [HTTP::payload]
set newpayload "$payload&action=$action&city=$city&state=$state"
set clen [string length $newpayload]

# replace the payload
if {$action eq "put"} {
   HTTP::payload replace 0 $clen $newpayload
}
HTTP::uri "/city/$action.php"
}
```

Disclaimer: there is no error checking in this rule, it assumes everything is perfect, which of course is unlikely to be the case in the real world. I am the archtypical developer, so I'll just say "it works on MY BIG-IP". :-)

Obviously this particular iRule is highly customized for a specific application, so YMMV. The basic concept of using the URI path and rewriting it is not only applicable for REST-like applications, but can also be used to support vanity URIs and other customizable path-based applications. The question certainly arises: *Why would I use REST-like behavior when I could achieve the same functionality using traditional web-based forms and eliminate the transformation step?*

That's a good question, and there are several benefits to this approach:

1. It completely eliminates the path to the application/script, which hides the implementation language from the client. This is often referred to as *resource cloaking* or *service virtualization* and is a security mechanism designed to hide as much information from attackers as possible.
2. The resulting code required to support AJAX-based requests and applications is much cleaner and somewhat smaller, resulting in a better performing application and less application logic on the client that can be manipulated (or exploited!).
3. For Web 2.0 applications it can provide a simpler URI to be shared with other members of the community and a URI that is more easily recalled by users.
4. It lays the groundwork for further transformations that enable integration between WOA (Web Oriented Architecture) and SOA (Service-oriented Architecture). This would require rewriting the request in a SOAP-XML format for exchange with a Web Service instead the conversion to a simple POST structure and a subsequent rewrite of the response.

Given the flexbility of iRules there are likely many other ways in which you could implement REST-like application support. That's what makes iRules so cool - there's always one more way to solve any given problem and it's likely that one of those ways is appropriate for your unique environment.

Imbibing: Pink Lemonade

Technorati tags: F5, MacVittie, iRules, REST, application delivery, Web 2.0, AJAX

F5 Networks, Inc. | 401 Elliot Avenue West, Seattle, WA 98119 | 888-882-4447 | f5.com

F5 Networks, Inc.
Corporate Headquarters
info@f5.com

F5 Networks
Asia-Pacific
apacinfo@f5.com

F5 Networks Ltd.
Europe/Middle-East/Africa
emeainfo@f5.com

F5 Networks
Japan K.K.
f5j-info@f5.com