MarkLogic

USER CONFERENCE 2011
SAN FRANCISCO | 26 - 29

REST URI rewriter

DEVELOPER LOUNGE LAB

Table of Contents

One aspect of presenting a clean, modern interface to your web applications is cleaning up their URI structure. An application page located at `/path/name/something` is easier to remember and understand than `/path/dispatch.xqy?task=name&arg1=something`.

Luckily, MarkLogic Server gives you access to a powerful URL rewriter that lets you clean up your URIs. The goal of this lab is to make you familiar with the rewriter and introduce a REST library that makes building applications with clean URIs easier and more robust. We'll start with an existing application that uses the URL rewriter and “raw code” to do the job, then we'll rewrite that code using the REST library.

1. The application

The application that we'll start with is one that display slides or foils. It's a web-based alternative to PowerPoint built on MarkLogic Server. Each slide deck is a single XML file that contains one or more slide pages. The underlying application takes URIs that look like this:

`/slides.xqy?deck=`*`deckname`*`&`*`slide`*`=slide-number&slide=`*`slide-number`*

If both the deck name and the slide number are provided, we return that slide. If only the deck name is given, we return the title page for the deck.

We don't want to expose those ugly URIs though, instead, we want `/slides/deckname` to show the title page for the deck and `/slides/deckname/slide-number` to show the individual slide.

2. The rewriter: the hard way

Assuming for the moment that we have `slides.xqy` written, let's think about the rewriter. What the URL rewriter does is transform the incoming URI into a new URI. The server then processes the new URI as if it was what had originally been sent. In other words, it looks at the incoming request and maps that request to a new URL. It doesn't

change other parts of the request.

Here's a URI rewriter that does the task set out above:

```
xquery version "1.0-ml";

declare default function namespace "http://www.w3.org/2005/xpath-functions";

declare option xdmp:mapping "false";

let $uri := xdmp:get-request-url()
return
  if ($uri = "/") ❶
  then
    "/default.xqy"
  else
    if (matches($uri, "^/slides/(.+)/(\d+)$")) ❷
    then
      replace($uri, "^/slides/(.+)/(\d+)$", "/slides.xqy?deck=$1&slide=$2")
    else
      if (matches($uri, "^/slides/(.+)/?$")) ❸
      then
        replace($uri, "^/slides/(.+)/(\d+)$", "/slides.xqy?deck=$1")
      else
        (xdmp:set-response-code(404, "Not found"), ❹
         "/no/such/resource")
```

It works like this:

❶ If the incoming URI is just "`/`" then map it to `/default.xqy`. This is probably unnecessary, but it never hurts to be explicit.

❷ Otherwise, if the URI matches the regular expression for a deck with a slide number, map it to the `/slides.xqy` with both the deck and slide parameters.

❸ Otherwise, if the URI matches the regular expression for just a deck, map it to the `/slides.xqy` with just the deck parameter.

❸ And finally, if we haven't matched it yet, turn it into a 404.

The imperative nature of this approach makes the rewriter very complex. With only four possibilities, it already requires careful review to understand. The complexity will only get worse as new URLs are added and new conditions are added such as supporting POST requests by the `admin` user or additional, optional parameters.

3. A simpler rewriter using rewrite rules

The REST API addresses this problem by provide a vocabulary for making declarative statements about URI mappings. Here, for example, is the description of the first two cases:

```
<options xmlns="http://marklogic.com/appservices/rest"> ❺

  <request uri="^/$" endpoint="/default.xqy"/> ❻

  <request uri="^/slides/(.+?)/(\d+)$" endpoint="/slides.xqy"> ❼
    <uri-param name="deck">$1</uri-param>                    ❽
    <uri-param name="slide" as="decimal">$2</uri-param>      ❾
  </request>
</options>
```

❺ The REST API elements are in the `http://marklogic.com/appservices/rest` namespace. Following the pattern of the search API, they're collected together in an `options` node.

❻ In the simplest case, the request just describes the mapping from a URI, still selected with a regular expression, to an endpoint.

❼ In the case where parts of the URI are extracted to make parameters, the regular expression is still used to identify the parts.

❽ But the parameters are called out separately.

❾ In addition to making the parameters easier to read and identify, it's possible to assign specific types to them. In the case of ordinary parameters, ones not extracted piecemeal from the URI (and described with a `param` element), it's possible to assert that they are required, provide enumerations, and default values.

Exercise 1

Following the examples above, write the `request` element for the third case where a slide deck is provided without a slide number.

4. The rewriter: the easy way

Given an `$options` node that describes the endpoint mappings, the URI rewriter becomes small and simple.

```
xquery version "1.0-ml";

import module namespace rest="http://marklogic.com/appservices/rest"
       at "rest.xqy";

declare default function namespace "http://www.w3.org/2005/xpath-functions";

declare option xdmp:mapping "false";

let $options := <options xmlns="http://marklogic.com/appservices/rest">
  <!-- the request elements go here -->
  </options>

let $result := rest:rewrite($options)
return
  if (empty($result))
  then
    (xdmp:set-response-code(404, "Not found"),
     "/no/such/resource")
  else
    $result
```

It's the call to `rest:rewrite` that does all the work. That function considers each of the `request` elements in turn and returns the mapping associated with the first request which matches the constraints expressed.

Supporting new URI mappings becomes a simple matter of writing a description for them, and adding that description at the right place in the `$options` node.

Exercise 2

Complete the options node in the preceding code sample to make a completely functioning rewriter.

5. Simpler end points

But wait, there's more! The other half of the equation in handling requests is the code in each end point module. Without the REST API library, each of them is required to process the request using ad hoc means. The resulting code

probably looks something like this:

```
...

let $deck       := concat("/slides/", xdmp:get-request-field("deck"), ".xml")
let $slideparam := xdmp:get-request-field("slide")
let $slide      := if (empty($slideparam)) then () else xs:decimal($slideparam)
return
  ...
```

Each request field is extacted “by hand”. This is often a two-step process to avoid errors. (You might, for example, have written

```
let $slide := xs:decimal(xdmp:get-request-field("slide"))
```

only to discover later that doing so raises a coercion error when there is no `slide` parameter. As in the rewriter case, this code becomes progressively more complicated as new parameters are added and/or validation requirements become more sophisticated.

But wait. In the URL rewriter, we've already demonstrated that we have code that can process a declarative description of a request. Can't we reuse that code here?

Yes, we can. The function you need to use is `rest:process-request`. Processing a request returns a map. Each parameter name is a key in the map, the value associated with that key is the value obtained from the request. This value will be correctly typed (so “`slide`” will be a `xs:decimal`). If a parameter is repeated, the value will be a list.

Exercise 3

Rewrite the code fragment above using

```
let $map := rest:process-request($request)
```

to get the processed request. Assign the other necessary variables by extracting them from the map.

6. More request features

In the short opportunity that this lab represents, there's no time to describe every possible feature of the REST API. For a taste, however, here are two more elements that you can use inside a request: `http` and `auth`.

The `http` element allows you to specify which HTTP methods are allowed. By default, if you don't include any `http` elements, the description matches HTTP “GET” requests. If you wanted to match a POST, you would specify: `<http method="POST"/>`.

The `auth` request tests the authentication of the user. For example,

```
<auth>
  <privilege>http://marklogic.com/xdmp/privileges/infostudio</privilege>
</auth>
```

is only satisfied if the user has the “infostudio” privilege. The `auth` request can occur either as a child of `request` or as a child of `http` to apply it on only a single method.

Exercise 4

Using these new features, write a `request` that accepts POST requests to a URI that matches “`/slides/(.+)/?$`” and maps them to the end point “`/post.xqy`” if and only if the user has the “infostudio” privilege.

7. An exercise left to the reader

The REST API library has been designed so that the same `request` element can be used in both the `options` node and in the call to `rest:process-request`.

As a more substantial exercise, work out a way to share the nodes between the rewriter and the endpoint so that you can be certain they will never get out-of-sync.

Hint: a separate library module with functions named `options` and `request` might form the basis for one approach.

A. Answers to Exercises

Answer to exercise 1

```
  <request uri="^/slides/(.+?)/?$" endpoint="/slides.xqy">
    <uri-param name="deck">$1</uri-param>
  </request>
```

Answer to exercise 2

```
let $options := <options xmlns="http://marklogic.com/appservices/rest">

      <request uri="^/$" endpoint="/default.xqy"/>

      <request uri="^/slides/(.+?)/(\d+)$" endpoint="/slides.xqy">
        <uri-param name="deck">$1</uri-param>
        <uri-param name="slide" as="decimal">$2</uri-param>
      </request>

      <request uri="^/slides/(.+?)/?$" endpoint="/slides.xqy">
        <uri-param name="deck">$1</uri-param>
      </request>
  </options>
```

Answer to exercise 3

```
...

let $request    := <request uri="^/slides/(.+?)/(\d+)$" endpoint="/slides.xqy">
                     <uri-param name="deck">$1</uri-param>
                     <uri-param name="slide" as="decimal">$2</uri-param>
                   </request>
let $map        := rest:process-request($request)
let $deck       := concat("/slides/", map:get($map, "deck", ".xml")
let $slide      := map:get($map, "slide")
return
  ...
```

Answer to exercise 4

```
  <request uri="^/slides/(.+?)/?$" endpoint="/post.xqy">
    <uri-param name="deck">$1</uri-param>
    <http method="POST">
      <auth>
        <privilege>http://marklogic.com/xdmp/privileges/infostudio</privilege>
      </auth>
    </http>
  </request>
```