QlikView Expressor: Accessing MarkLogic

As with many No-SQL databases, MarkLogic stores content as XML or JSON documents, although text and binary content may also be stored. While MarkLogic provides an ODBC driver and interface, using this approach requires some pre-configuration of the database and compromises your ability to extract the hierarchal structure of the stored documents.

QlikView Expressor can extract data from MarkLogic using the vendor supplied REST API. Consequently, it is possible to retrieve information without losing the structural information inherent in the stored content.

Let's see how this works.

MarkLogic Database Content

The procedure described in this document is based on the examples described in the MarkLogic REST Application Developer's Guide and the Learning the MarkLogic REST API tutorial. The database contained the two documents (one.xml and two.json) described in the Developer's Guide and the XML versions of Shakespeare's plays and the JSON documents describing talks presented at a recent MarkLogic World conference provided as part of the tutorial.

Once the documents have been loaded into the database, you may execute a REST GET request by placing the following URL into a browser (where hostname is the name or IP address of the server running MarkLogic). This request will return all database entries that contain the word 'caesar.'

```
http://hostname:8003/v1/search?format=json&q=caesar
```

To format the returned document into a more readable representation, submit the document to a JSON formatter. From the formatted document you can easily see that there are eleven entries in the database that include the word 'caesar' and that some of these entries include multiple references. For example, the first document returned includes four references contained within the array **matches**.


```
{
         "index":1,
         "uri":"/j_caesar.xml",
         "path":"fn:doc(\"/j_caesar.xml\")",
         "score":123648,
         "confidence":0.828815,
         "fitness":1,
         "href":"/v1/documents?uri=%2Fj_caesar.xml",
         "mimetype":"text/xml",
         "format":"xml",
         "matches":[
            {
               "path":"fn:doc(\"/j_caesar.xml\")/PLAY/TITLE",
               "match-text":[
```

```
            "The Tragedy of Julius ",
            {
               "highlight":"Caesar"
            }
         ]
      },
      {
         "path":"fn:doc(\"/j_caesar.xml\")/PLAY/PERSONAE/PERSONA[1]",
         "match-text":[
            "JULIUS ",
            {
               "highlight":"CAESAR"
            }
         ]
      },
      {
         "path":"fn:doc(\"/j_caesar.xml\")/PLAY/PERSONAE/PGROUP[1]/PERSONA[1]",
         "match-text":[
            "OCTAVIUS ",
            {
               "highlight":"CAESAR"
            }
         ]
      },
      {
         "path":"fn:doc(\"/j_caesar.xml\")/PLAY/PERSONAE/PGROUP[1]/GRPDESCR",
         "match-text":[
            "triumvirs after death of Julius ",
            {
               "highlight":"Caesar"
            },
            "."
         ]
      }
   ]
}
```


QlikView Expressor Datascript Module

While Expressor does not include an input operator pre-configured to read from MarkLogic, you may add code to the Read Custom operator that uses cURL to issue a REST GET request against the database. This code can be written so that it may be used to issue any REST API request against MarkLogic or any other REST API that returns JSON documents. Since this code has utility beyond this simple example, it should be placed into an Expressor Datascript Module in a dedicated library so that it may be incorporated into other projects.

In this example, the code, in a function named `query`, has been placed into a datascript module named `RESTget` in a library named `RESTapi`.

On lines 5 and 8 datascript modules included with QlikView Expressor are referenced. Lines 18-42 execute the GET request, and line 43 converts the JSON document returned by the REST call into an Expressor Datascript table. The code that invokes the `query` function must then extract the desired

information from the Datascript table. The fact that the credentials and query URL are passed as arguments to the `query` function provides for code reusability.

```
1  -- module definition to load functions using the module name.
2  -- Calls of functions in this module are formed as RESTapi.RESTget.<functionname>();
3
4  -- include dscurl package
5  curl = require('dscurl')
6
7  -- include json package
8  json = require('json')
9
10 module("RESTapi.RESTget", package.seeall)
11
12 -- @function query(user, pwd, url_query)
13 -- @tip url_query is the REST API url with query
14 -- @tip returns a datascript table
15 -- @access public
16 --
17 function query(user, pwd, url_query)
18   assert(type(user) == 'string' and #user > 0, 'Missing user parameter')
19   assert(type(pwd) == 'string' and #pwd > 0, 'Missing pwd parameter')
20
21   -- local table of string fragments to hold XML results from the call
22   local results = {}
23
24   -- create a curl request option
25   local c = curl.new()
26
27   c:setopt(curl.OPT_URL, url_query)
28   c:setopt(curl.OPT_USERPWD, user .. ':' .. pwd)
29   c:setopt(curl.OPT_HTTPAUTH, curl.AUTH_ANY)
30   c:setopt(curl.OPT_VERBOSE, false)
31   c:setopt(curl.OPT_WRITEDATA, results)
32   c:setopt(curl.OPT_WRITEFUNCTION,
33     function (r, buffer)
34       table.insert(r, buffer)
35       return #buffer
36     end)
37
38   local s, e1, e2 = c:perform()
39   if not s then
40     error('Failed to perform REST call: ' .. tostring(e1) .. ' ' .. tostring(e2) )
41   end
42   c:close()
43   return json.decode(table.concatenate(results))
44 end
45
```

This code is not as complex as it appears. Lines 25 through 36 set up, and line 38 executes, the REST GET request. These statements are pretty much boiler-plate code in that you will most likely not need to edit them. Once the request is made (line 38) the returned JSON document is stored row by row into a numerically indexed table named `results` by the anonymous callback function (lines 33-36). Line 43 converts the contents of this table (`table.concat`) into a single string value (which is a JSON

document) that is then converted into an Expressor Datascript table (`json.decode`) and returned to the calling function.

The Expressor Dataflow

The dataflow can be quite simple. All of the processing can be included in the Read Custom operator and the extracted information written to an output file such as a QlikView QVX or QVD file. Since it is this code that must extract the desired information from the datascript table returned from the REST GET request, you may want to place this parsing code into a helper function, either within the Read Custom operator or in a second datascript module. Since this code is specific to this application, the datascript module is most likely included within the project, although it could be placed into a library.

Let's place the code in a datascript module named `ParseJSON` and include it, along with the dataflow, in a project named `Shakespeare`. The processing implementation depends on what information you want to extract. For this example, extract the document URI, the number of citations in each returned document and the text from each citation.

```
1  -- module definition to load functions using the module name.
2  -- Calls of functions in this module are formed as Shakespeare.ParseJSON.<functionname>();
3  module("Shakespeare.ParseJSON", package.seeall)
4
5  -- @function parseCitations(result)
6  -- @tip extracts identifying information and citation text from each query match
7  -- @access public
8  --
9  function parseCitations(document)
10   citationtext = ""
11   -- capture the array of citations
12   citationarray = document.matches
13
14   -- iterate through the citation array
15   for key, citation in ipairs(citationarray) do
16     -- each element is a string indexed table
17     -- the element at index 'match-text' contains the citation text
18     for k, text in ipairs(citation['match-text']) do
19       -- the citation text is an array
20       -- but at index 2 the element is a string indexed table
21       -- indexes 1 and 3 (if present) just contain text
22       citationtext = string.concatenate(citationtext,(k==2 and text.highlight or text))
23     end
24     -- put space characters between each citation
25     citationtext = string.concatenate(citationtext,"     ")
26   end
27
28   output = {}
29   output.Title = document.uri
30   output.Matches = tointeger(#document.matches)
31   output.Citations = citationtext
32   return output
33 end
34
```

The coding in the Read Custom operator invokes on the REST API within the `initialize` function and uses the iterative `read` function to extract the information from each document satisfying the query.

Note that three output attributes have been defined.

- Title: a string attribute that holds the document URI
- Matches: an integer attribute that contains the number of citations in the document
- Citations: a string attribute that contains the citations
    - Multiple citations are concatenated together separated by space characters

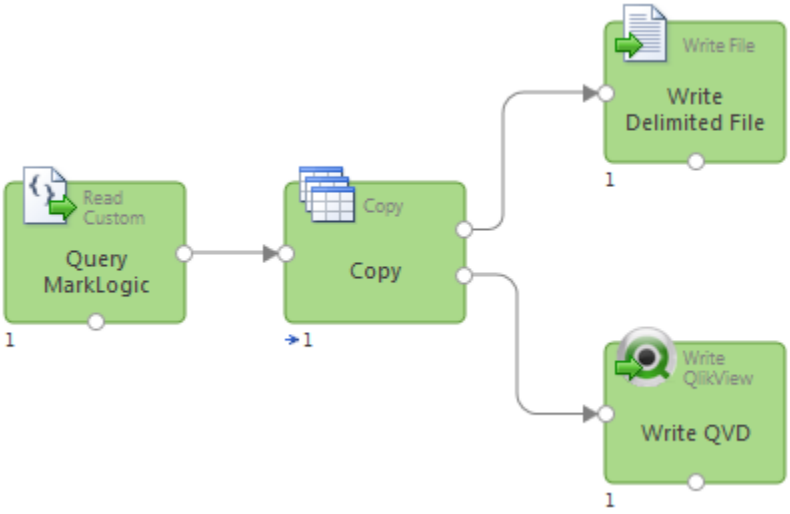
```
-- import the RESTapi.RESTget datascript module
-- @datascript module dependency
require("RESTapi.0.RESTget")

-- import the Shakespeare.ParseJSON datascript module
-- @datascript module dependency
require("Shakespeare.0.ParseJSON")

resultarray = nil
numberOfDocuments = 0
cnt = 0

function initialize()
  -- Perform any initialization before the main read loop.
  datascripttable = RESTapi.RESTget.query(
    'expressor','expressor',"http://localhost:8003/v1/search?format=json&q=caesar"
  )

  resultarray = datascripttable['results']
  numberOfDocuments = #resultarray
  cnt = 1
end;

function read()
  if cnt <=numberOfDocuments then
    document = resultarray[cnt]
    cnt = cnt + 1
    return  Shakespeare.ParseJSON.parseCitations(document)
  else return true
  end
end;
```

Outputs

Output (Copy)

- Title
- Matches
- Citations

The output from the Read Custom operator is connected to one or more output operators.



Understanding the Processing Logic

Let's start with the code in the Read Custom operator.

The `initialize` function uses the `query` function in the `RESTapi.RESTget` datascript module to issue the GET request against MarkLogic. Note that the credentials (`"expressor","expressor"`) and REST URL are provided as arguments. You can replace the credentials and URL with those appropriate to your MarkLogic installation and desired query. The URL includes a simple query and a request to format the response as a JSON document. Expressor could handle an XML return document, but the coding would be a bit more complex.

Remember that the function `query` returns a datascript table whose content mirrors the structure of the JSON document returned by the GET request. This datascript table is captured into the variable named `datascripttable` (line 15). Line 18 then extracts an array containing the matching documents from the datascript table. The index to this array is the string key `'results'`. How did we know this was the key value? Let's look at the first few lines of the JSON document returned by the GET request.

```
{
   "snippet-format":"snippet",
   "total":11,
   "start":1,
   "page-length":10,
   "results":[
```

The JSON document includes a top level entry `'results'` whose value is an array, indicated by the opening square brace `[` character. The datascript table includes a corresponding element with the string index `results` whose associated value is a numerically indexed datascript table (the datascript equivalent of the JSON array). Therefore, to retrieve this array, you use the datascript notation tableName.elementName or tableName[elementName] – `datascripttable.results` or `datascripttable['results']` – and assign this numerically indexed table to the variable named `resultarray`. Since `resultarray` is a numerically indexed datascript table, the `#` operator returns the number of elements, which is captured into the variable named `numberOfDocuments` (line 20).

The Expressor runtime engine then repeatedly invokes the function `read`. The `if..then..else` statement (lines 25-30) iterates through the array of documents, passing each document to the `parseCitations` function in the `Shakespeare.ParseJSON` datascript module. The function `parseCitations` returns the record that the Read Custom operator then emits. Once the array has been completely processed, the function `read` returns `true`, which tells the Expressor runtime engine to stop invoking the function `read`. Thus the Read Custom operator will process each of the documents returned from the GET request and then shut down.

Now let's detail the function `parseCitations`. Line 12 extracts a numerically indexed array of citations in each returned document. Again, you might ask how the element containing this array was identified? Go back to the JSON document.

```
{
    "{snippet-format":"snippet",
     "total":11,
     "start":1,
     "page-length":10,
     "results":[
        {
           "index":1,
           "uri":"/j_caesar.xml",
           "path":"fn:doc(\"/j_caesar.xml\")",
           "score":123648,
           "confidence":0.828815,
           "fitness":1,
           "href":"/v1/documents?uri=%2Fj_caesar.xml",
           "mimetype":"text/xml",
           "format":"xml",
           "matches":[
```

Now notice that within each element of the `results` array (which is a JSON document), there is another array named **`matches`**. Therefore the datascript table, held in the variable named `document`, has a string indexed element with key `matches` that is an array of citations. The statement on line 12 captures this array into a variable named `citationarray`. Lines 15-26 then iterate through each of the citations in `citationarray`.

The coding in lines 15-26 is specific to this example and would need to be changed as appropriate to the REST GET query actually executed by the Read Custom operator's `initialize` function.

- The variable `citationarray` holds a numerically indexed datascript able where each element is a citation from within the document.
    - Line 15 uses the `ipairs` iterator function to sequentially process each element in this array.
    - Each iteration returns the `key` (the numeric index of the element) and the value of the element – the `citation`.
- Each `citation` is itself a string indexed datascript table with two index values: `path`, `match-text`.
    - And `match-text` is a numerically indexed table with two or three elements.
- Line 18 uses the `ipairs` iterator function to sequentially process each element in `match-text`.
    - Each iteration returns the `key` (`k`) and the value of the element – `text`.
    - For element 1 (and element 3 if present), `text` is a string value.
    - For element 2, `text` is a string indexed table with the index value `highlight`.
- Line 22 concatenates the components of each citation into a single string that will be returned from the function `parseCitations` in the attribute `Citations`.
    - The `and..or` statement uses the index to distinguish between string values and the nested table and extracts the value to be concatenated.

Conclusion

The MarkLogic REST API may be used from an Expressor Read Custom operator to retrieve rich hierarchal documents from the MarkLogic database as JSON documents. The returned JSON document is then transformed into an Expressor Datascript table and the desired content extracted into the records emitted by the Read Custom operator.

Since the structure of the datascript table exactly parallels the structure of the JSON document returned from MarkLogic, the JSON document can be used as a guide to select desired content.

This approach requires no setup or reconfiguration of the MarkLogic database and can exploit the full capabilities of the REST API. The Expressor developer is able to implement this approach without the assistance of the MarkLogic administrator or developer.

If desired, multiple record types can be derived from each document returned from MarkLogic, which allows a single Expressor dataflow to flatten hierarchal document content into the more normalized format required by relational databases and business intelligence and discovery tools.

Attached to this knowledge base article is an export of the `RESTapi` library that contains the `RESTget` datascript module. You may load this library into a workspace and reference it from any project or library in which you want to use the function `query`.

Appendix

The complete JSON response document generated by the query URL shown in the example follows. Note that by default only the first 10 of the 11 retrieved documents are returned in the JSON document.


```
{
   "snippet-format":"snippet",
   "total":11,
   "start":1,
   "page-length":10,
   "results":[
      {
         "index":1,
         "uri":"/j_caesar.xml",
         "path":"fn:doc(\"/j_caesar.xml\")",
         "score":123648,
         "confidence":0.828815,
         "fitness":1,
         "href":"/v1/documents?uri=%2Fj_caesar.xml",
         "mimetype":"text/xml",
         "format":"xml",
         "matches":[
            {
               "path":"fn:doc(\"/j_caesar.xml\")/PLAY/TITLE",
               "match-text":[
                  "The Tragedy of Julius ",
                  {
                     "highlight":"Caesar"
                  }
               ]
            },
            {
               "path":"fn:doc(\"/j_caesar.xml\")/PLAY/PERSONAE/PERSONA[1]",
               "match-text":[
                  "JULIUS ",
                  {
                     "highlight":"CAESAR"
                  }
               ]
            },
            {

"path":"fn:doc(\"/j_caesar.xml\")/PLAY/PERSONAE/PGROUP[1]/PERSONA[1]",
               "match-text":[
                  "OCTAVIUS ",
                  {
                     "highlight":"CAESAR"
                  }
               ]
            },
            {

"path":"fn:doc(\"/j_caesar.xml\")/PLAY/PERSONAE/PGROUP[1]/GRPDESCR",
               "match-text":[
                  "triumvirs after death of Julius ",
```

```
                  {
                     "highlight":"Caesar"
                  },
                  "."
               ]
            }
         ]
      },
      {
         "index":2,
         "uri":"/a_and_c.xml",
         "path":"fn:doc(\"/a_and_c.xml\")",
         "score":123648,
         "confidence":0.828815,
         "fitness":1,
         "href":"/v1/documents?uri=%2Fa_and_c.xml",
         "mimetype":"text/xml",
         "format":"xml",
         "matches":[
            {

"path":"fn:doc(\"/a_and_c.xml\")/PLAY/PERSONAE/PGROUP[1]/PERSONA[2]",
               "match-text":[
                  "OCTAVIUS ",
                  {
                     "highlight":"CAESAR"
                  }
               ]
            },
            {

"path":"fn:doc(\"/a_and_c.xml\")/PLAY/PERSONAE/PGROUP[3]/GRPDESCR",
               "match-text":[
                  "friends to ",
                  {
                     "highlight":"Caesar"
                  },
                  "."
               ]
            },
            {
               "path":"fn:doc(\"/a_and_c.xml\")/PLAY/PERSONAE/PERSONA[2]",
               "match-text":[
                  "TAURUS, lieutenant-general to ",
                  {
                     "highlight":"Caesar"
                  },
                  "."
               ]
            },
            {
               "path":"fn:doc(\"/a_and_c.xml\")/PLAY/PERSONAE/PERSONA[5]",
               "match-text":[
                  "EUPHRONIUS, an ambassador from Antony to ",
                  {
```

```
                "highlight":"Caesar"
             },
             "."
          ]
       }
    ]
 },
 {
    "index":3,
    "uri":"/xml/one.xml",
    "path":"fn:doc(\"/xml/one.xml\")",
    "score":43008,
    "confidence":0.488809,
    "fitness":0.685202,
    "href":"/v1/documents?uri=%2Fxml%2Fone.xml",
    "mimetype":"text/xml",
    "format":"xml",
    "matches":[
       {
          "path":"fn:doc(\"/xml/one.xml\")/one/child",
          "match-text":[
             "The noble Brutus has told ",
             {
                "highlight":"Caesar"
             },
             " was ambitious"
          ]
       }
    ]
 },
 {
    "index":4,
    "uri":"/json/two.json",
    "path":"fn:doc(\"/json/two.json\")",
    "score":43008,
    "confidence":0.488809,
    "fitness":0.685202,
    "href":"/v1/documents?uri=%2Fjson%2Ftwo.json",
    "mimetype":"application/json",
    "format":"json",
    "matches":[
       {
          "path":"fn:doc(\"/json/two.json\")/*:json/*:two/*:child",
          "match-text":[
             "I come to bury ",
             {
                "highlight":"Caesar"
             },
             ", not to praise him."
          ]
       }
    ]
 },
 {
    "index":5,
```

```
         "uri":"/rich_iii.xml",
         "path":"fn:doc(\"/rich_iii.xml\")",
         "score":32256,
         "confidence":0.423321,
         "fitness":0.593402,
         "href":"/v1/documents?uri=%2Frich_iii.xml",
         "mimetype":"text/xml",
         "format":"xml",
         "matches":[
            {

"path":"fn:doc(\"/rich_iii.xml\")/PLAY/ACT[3]/SCENE[1]/SPEECH[19]/LINE[2]",
               "match-text":[
                  "Did Julius ",
                  {
                     "highlight":"Caesar"
                  },
                  " build that place, my lord?"
               ]
            },
            {

"path":"fn:doc(\"/rich_iii.xml\")/PLAY/ACT[3]/SCENE[1]/SPEECH[27]/LINE[1]",
               "match-text":[
                  "That Julius ",
                  {
                     "highlight":"Caesar"
                  },
                  " was a famous man;"
               ]
            },
            {

"path":"fn:doc(\"/rich_iii.xml\")/PLAY/ACT[4]/SCENE[4]/SPEECH[88]/LINE[46]",
               "match-text":[
                  "And she shall be sole victress, ",
                  {
                     "highlight":"Caesar"
                  },
                  "'s ",
                  {
                     "highlight":"Caesar"
                  },
                  "."
               ]
            }
         ]
      },
      {
         "index":6,
         "uri":"/rich_ii.xml",
         "path":"fn:doc(\"/rich_ii.xml\")",
         "score":16128,
         "confidence":0.299333,
         "fitness":0.419599,
```

```
         "href":"/v1/documents?uri=%2Frich_ii.xml",
         "mimetype":"text/xml",
         "format":"xml",
         "matches":[
            {

"path":"fn:doc(\"/rich_ii.xml\")/PLAY/ACT[5]/SCENE[1]/SPEECH[1]/LINE[2]",
               "match-text":[
                  "To Julius ",
                  {
                     "highlight":"Caesar"
                  },
                  "'s ill-erected tower,"
               ]
            }
         ]
      },
      {
         "index":7,
         "uri":"/othello.xml",
         "path":"fn:doc(\"/othello.xml\")",
         "score":16128,
         "confidence":0.299333,
         "fitness":0.419599,
         "href":"/v1/documents?uri=%2Fothello.xml",
         "mimetype":"text/xml",
         "format":"xml",
         "matches":[
            {

"path":"fn:doc(\"/othello.xml\")/PLAY/ACT[2]/SCENE[3]/SPEECH[42]/LINE[2]",
               "match-text":[
                  "He is a soldier fit to stand by ",
                  {
                     "highlight":"Caesar"
                  }
               ]
            }
         ]
      },
      {
         "index":8,
         "uri":"/as_you.xml",
         "path":"fn:doc(\"/as_you.xml\")",
         "score":16128,
         "confidence":0.299333,
         "fitness":0.419599,
         "href":"/v1/documents?uri=%2Fas_you.xml",
         "mimetype":"text/xml",
         "format":"xml",
         "matches":[
            {

"path":"fn:doc(\"/as_you.xml\")/PLAY/ACT[5]/SCENE[2]/SPEECH[12]/LINE[3]",
               "match-text":[
```

```
                  "and ",
                  {
                     "highlight":"Caesar"
                  },
                  "'s thrasonical brag of 'I came, saw, and"
               ]
            }
         ]
      },
      {
         "index":9,
         "uri":"/lll.xml",
         "path":"fn:doc(\"/lll.xml\")",
         "score":16128,
         "confidence":0.299333,
         "fitness":0.419599,
         "href":"/v1/documents?uri=%2Flll.xml",
         "mimetype":"text/xml",
         "format":"xml",
         "matches":[
            {

"path":"fn:doc(\"/lll.xml\")/PLAY/ACT[5]/SCENE[2]/SPEECH[291]/LINE",
               "match-text":[
                  "The pommel of ",
                  {
                     "highlight":"Caesar"
                  },
                  "'s falchion."
               ]
            }
         ]
      },
      {
         "index":10,
         "uri":"/titus.xml",
         "path":"fn:doc(\"/titus.xml\")",
         "score":16128,
         "confidence":0.299333,
         "fitness":0.419599,
         "href":"/v1/documents?uri=%2Ftitus.xml",
         "mimetype":"text/xml",
         "format":"xml",
         "matches":[
            {

"path":"fn:doc(\"/titus.xml\")/PLAY/ACT[1]/SCENE/SPEECH[2]/LINE[2]",
               "match-text":[
                  "If ever Bassianus, ",
                  {
                     "highlight":"Caesar"
                  },
                  "'s son,"
               ]
            }
```

```
            ]
         }
      ],
      "qtext":"caesar",
      "metrics":{
         "query-resolution-time":"PT0.008S",
         "facet-resolution-time":"PT0S",
         "snippet-resolution-time":"PT0.108S",
         "total-time":"PT0.116S"
      }
   }
```