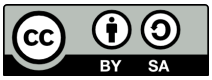
White Paper

MarkLogic / Application Integration

Preferring a REST model over language-specific connectors when integrating applications with MarkLogic

Ron Hitchens, Principal Consultant
OverStory Ltd
overstory.co.uk
ron@overstory.co.uk



Abstract

This paper discusses methods by which software applications can interface to MarkLogic to retrieve, update and run queries against stored content. MarkLogic provides several access protocols (appservers) for this purpose which include HTTP, XDBC and WebDAV.

Traditionally, applications (software rather than people) have used the XDBC appserver protocol to access MarkLogic using either the Java or C# version of XCC as a connector. The author of this paper, while employed at MarkLogic, designed and implemented the XCC connector in Java and C#.

In this paper we advocate moving away from language-specific connectors like XCC and toward a language-neutral access model based on HTTP and using the design principles embodied in REST. There are many robust solutions available for implementing REST, both client and server side, that make it the preferred choice over XCC for application integration with MarkLogic.

Table of Contents

The Issue 4

Advantages of XCC *4*

Disadvantages of XCC *4*

The Solution 6

REST is Language Neutral *6*

REST Means More Flexible Architectures *7*

Downsides of REST vs XCC *7*

MarkLogic and REST Are a Natural Fit *8*

Conclusion 10

The Issue

MarkLogic is a powerful XML database and search platform that can store, update and query vast amounts of XML content very efficiently. XQuery, a Turing-complete functional programming language designed expressly for querying XML documents, runs natively on MarkLogic and can be used to implement complex business applications.

However, the majority of major software applications built around MarkLogic are architected in multiple tiers - Java or .Net usually making up the middle tier - where MarkLogic is accessed primarily by code running externally to it. To address this need MarkLogic created the XCC [1] connector that has language bindings for Java and C#.

Advantages of XCC

XCC provides many services to client application code. It manages connection details (given a URL that identifies the MarkLogic server) and automatically pools and re-uses network connections without requiring that client code take any action. This can greatly improve throughput when an application submits many small queries in rapid succession.

XCC also handles marshaling of data values from Java/.Net variables for transport across the wire to MarkLogic, and un-marshals the response. Optionally, XCC can also stream very large result values, such as XML or text documents, that are too large to fit in temporary working memory. Streaming is useful for “passing through” large data values that need not be reified in the client.

XCC provides a suite of classes that map Java or C# values to corresponding XML Data Model (XDM) data types. These data types are used by XCC to marshal parameters to be sent to an XQuery program and to translate a result sequence for interpretation by Java/C#. This gives the application the ability to submit any arbitrary XQuery request and interpret any possible result sequence it might produce.

Lastly, XCC can insert documents in MarkLogic, of any size, specifying various meta information (URI, permissions, collections, etc) for each document. Documents may also be inserted into MarkLogic via HTTP by constructing simple XQuery programs that include the XML or text to be inserted - essentially a program whose purpose is to create a document. But the document insertion function of XCC (insertContent()) goes beyond this, working in a streaming mode where no XQuery code is invoked and the data is sent directly to the database.

Disadvantages of XCC

XCC is widely used by most major MarkLogic customers who implement business applications with a traditional Java or .Net software stack. When used with one of the two supported languages, XCC provides a powerful integration solution. But XCC also has its drawbacks.

---

[1] **X**ML **C**ontent **C**onnector (http://developer.marklogic.com/pubs/5.0/javadoc/index.html)

First, XCC has a bias toward Java over C#. The XCC connector is actually implemented only in Java, even in the C# version. To create XCC/.Net, the compiled Java code is processed by a tool that converts the Java byte code to executable .Net CIL[2] code. A thin C# API layer sits atop the converted Java code and translates between client C# code above and the Java XCC core underneath.

To support the embedded Java XCC code, a converted Java compatibility library is bundled with the .Net XCC library (GNU Classpath[3]). This works fine and the C# version of XCC runs at full speed in the CLR[4], but there is a startup penalty because the large JRE[5] library layer needs to be loaded on first use. For long-running applications, this penalty is insignificant. But for programs that are started each time they need to communicate with MarkLogic, this penalty (about one to two seconds) can be problematic.

Another issue is the XDBC/XCC communication protocol itself. Under the covers, XCC uses an unpublished variation of the HTTP protocol to interact with MarkLogic. Although it's based on HTTP, the XDBC/XCC protocol is not fully HTTP compliant.

As a private protocol between XCC and MarkLogic, the incompatibilities with HTTP shouldn't matter. But as an practical matter it has implications. For example, because the XDBC/XCC protocol does not honor cookies or preserve other custom HTTP headers, XCC connections do not interact well with intermediaries such as hardware load balancers or caching agents. This can reduce the ability of administrators to manage traffic load on clusters, to provide sticky connections for server affinity or serve identical responses out of a cache. [*Update: MarkLogic v7 has addressed some of these issues, particularly interoperability with load balancers*]

Another drawback to using XCC is that it creates a dependency in your application code on the MarkLogic XCC library. Admittedly, MarkLogic is unlikely to discontinue support for XCC or break compatibility in any significant way. But using XCC does limit your options to those provided by the XCC API, and it means your application code is closely coupled to the XCC data types and classes. It also means that, to some extent, your application's evolution is constrained by the current feature set of XCC.

But the biggest downside to using XCC is that if you're not using Java or C#, you're out of luck. If you're writing your application in another JVM-based language, such as Scala, Clojure or JRuby, the Java version of XCC can still be used but you will need to write some amount of glue code to cross the language barrier.

But if you're using a language for which there is no native XCC port, such as C/C++, Ruby, Erlang, PHP, etc, then there is no MarkLogic-supported connector solution.

---

[2] **C**ommon **I**ntermediate **L**anguage - .Net equivalent of Java byte code

[3] http://www.gnu.org/software/classpath/

[4] **C**ommon **L**anguage **R**untime - .Net equivalent of the Java Virtual Machine

[5] **J**ava **R**untime **E**nvironment

The Solution

Using language-neutral, resource-oriented interfaces is an alternative, less restrictive approach to a language-specific connector. XCC relieves the user of most of the heavy lifting required to communicate with MarkLogic. However, this convenience comes at the cost of tightly coupling your application code to the connector and closing off other access options.

The architectural style known as REST[6] has gained popularity in recent years. REST interfaces are inherently language-neutral and make no assumptions about the implementation details of the communicating parties. REST leverages the generality and ubiquity of the HTTP protocol to transport messages between clients and servers.

XCC is a bridge between Java/C# on the client and XQuery on the server. By definition this means that it couples to both languages and must control the conversation. If you need or want a channel from your app into XQuery on MarkLogic, this works well. But as a design principle it is best to keep interfaces loosely coupled and not allow implementations details to leak through the interface.

Experience has shown that although XCC provides a rich modeling of XDM data types and XQuery result sequences, most customers don't make much use of these. The most common scenario is to write a few adapter classes that call MarkLogic and return a single String or XML document. These adapters may invoke XQuery modules by name or send canned XQuery code snippets for evaluation, but in most cases the XQuery is isolated from the rest of the application code - as it should be.

REST is Language Neutral

REST is an architectural style that focuses on resources (nouns) and a small number of standard actions (verbs) that can be can be applied to those resources. RESTful interfaces are, for the most part, declarative rather than imperative. You address a resource (with a URI), specify a verb (GET, PUT, POST, etc) and, where applicable, a new desired state (value) for the resource. REST does *not* specify which function to execute or how a resource should be processed.

By using RESTful interfaces to communicate with MarkLogic, rather than XCC, the language separation between client and server is complete. Client application code need not know which XQuery modules to invoke, submit XQuery snippets for evaluation or be aware of XQuery at all. All information needed to handle the request is encoded as a REST message (URI, verb and resource representation).

Nor are there any language-specifics inherent in the interface. Any client, written in any language, can access a RESTful service using standard HTTP semantics and receive a meaningful response. Breaking the programming language dependency opens up a wide range of possibilities in your architecture that specialized connectors like XCC preclude.

---

[6] **RE**presentational **S**tate **T**ransfer, a term found in Roy Fielding's doctoral thesis. Dr. Fielding was one of the architects of the ubiquitous HTTP protocol that underpins the World Wide Web.

REST Means More Flexible Architectures

A well designed RESTful service allows evolution on both sides of the interface. The service itself can change its implementation in any way it chooses so long as it continues to respond consistently to requests on that interface. For example, a test implementation of an interface can be scripted using simple shell or PHP code to drive automated testing of client code without the need to even deploy code on a MarkLogic instance.

Clients of the service can also evolve independently. A given client is free to switch out modules or libraries for newer versions or even be entirely re-implemented in a completely different language. Specialized clients for different environments (desktop, server, mobile, etc) are also possible, in whichever language is appropriate. Everything will continue to work fine as long as the simple HTTP semantics are honored by both parties.

REST is also a much better approach to versioning. HTTP has well defined rules for content negotiation and specifying meta information about a request. Honoring this simple communication style makes a RESTful service not only backward compatible but also forward compatible - working properly with newer clients as well as older ones.

A truly RESTful service interface is compliant with the HTTP protocols and conventions. Those protocols and conventions allow (in fact condone and encourage) intermediaries between client and server. Intermediaries can perform many valuable services, such as caching and traffic management, that can increase scalability and reliability of a service by orders of magnitude. Judicious use of cache control headers, for example, can act as a scalability multiplier, intercepting requests that can be satisfied from cache rather than hitting MarkLogic again.

Most importantly, additional clients, perhaps unanticipated when the service was developed, can access the service through the well-defined REST interface. These new clients can be written in any language and need not even be aware that they are accessing a MarkLogic system and invoking XQuery. New apps can be rolled out quickly, leveraging the investment already made in the existing service(s).

Downsides of REST vs XCC

There are some disadvantages of using REST rather than XCC - but surprisingly few. XCC marshals and unmarshals data for you and automatically creates, pools and reaps connections. XCC needs only a standard URL (with a scheme of “xcc”) to identify the MarkLogic server location and provide credentials. A client accessing MarkLogic via REST will be responsible for managing its own connections and interpreting responses.

MarkLogic can throw certain types of retryable exceptions that indicate your request is not in error but cannot be handled right now. When these happen, XCC will automatically resubmit the request on your behalf. The most common cause of these exceptions is dead-lock breaking in a multi-machine cluster. These exceptions are quite rare these days since automatic query restart is now attempted within the cluster using a more robust back-off algorithm before being surfaced as an XCC exception.

In order to communicate with a RESTful service, a client must create and manage network connections, encode messages in the HTTP protocol and interpret the HTTP response. This is a non-trivial amount of complexity but, fortunately, is effectively a non-issue in practice. Many robust, full-featured HTTP libraries are widely available on any platform that can realistically be used as a REST client, including mobile devices. For Java, Jersey[7] (the JAX-RS reference implementation) includes an easy to use REST client. There are also newer frameworks such as Play[8] that make use of asynchronous libraries like Netty[9] and Akka[10] for distributed communication. For C#, RestSharp[11] plays a similar role.

The one thing that you can't do with a REST approach that you can do with XCC is to insert content directly into MarkLogic. The XDBC/XCC protocol has a non-HTTP mode that bypasses XQuery and streams data straight to the database. This allows very large documents, which may too large to be held in working memory, to be loaded directly. A RESTful MarkLogic HTTP interface, which invokes an XQuery module to handle a request, cannot exactly duplicate this behavior.

As a practical matter, this is not important for most applications. Reasonably sized documents (a few tens of megabytes or less) can easily be handled through a REST endpoint that receives a document (or documents) and then does programmatic inserts. This works fine in the majority of cases. For those situations where very large documents must be ingested, XCC can still be used for that purpose while using REST for everything else.

In content-heavy applications, querying and loading content are usually different activities. A web-facing application usually makes read-only queries against MarkLogic, and these queries can easily be modeled as a REST service. Content loading is a back-office activity, typically implemented in a separate codebase that runs independently of the front-end application. In many cases, production and loading of content are done by different teams, departments or even organizations.

MarkLogic and REST Are a Natural Fit

There are several ready to use REST libraries that make it easy to create a RESTful web service directly on MarkLogic. Luckily you need look no further than the MarkLogic developer site.

---

[7] http://jersey.java.net/ JAX-RS = **J**ava **A**PIs for **X**ML - **R**ESTful **S**ervices

[8] https://www.playframework.com

[9] http://netty.io

[10] http://akka.io

[11] http://restsharp.org/

The original MarkLogic REST Library (ml-rest-lib[12]) is a light-weight URL mapper that dispatches to XQuery modules by matching request URLs against patterns. Not to be confused with the more recent general purpose MarkLogic REST API (see below), ml-rest-lib gives you complete control of the REST URL space. Only the endpoints you define are available, which makes it easy to enforce an interface contact. This library is also very handy for "REST-ifying" an existing XQuery codebase while still controlling entry points into that code. This library ships with MarkLogic and can be used by including a simple import declaration in your code. This is the REST solution that OverStory usually recommends and we make available an open source template[13] to make it easier to get started with it.

Another option is the RestXQ library[14]. This library makes use of annotations in XQuery 3.0 to let you tag functions in your code as handlers for specific resource URLs. Similar to ml-rest-lib, you can completely control the URL space of your interface. The difference is primarily one of style and philosophy. With ml-rest-lib, configuration is in a single place and endpoints map to XQuery main modules. With RestXQ, configuration is distributed throughout the code base and endpoints map to functions in library modules. Existing XQuery code cannot be used for RestXQ without adding the annotations. RestXQ requires at least MarkLogic 6.x whereas ml-rest-lib will run on older versions of MarkLogic.

The newer REST API[15] that ships with MarkLogic since 6.x is a general purpose API that makes most of the internal functions of MarkLogic available as REST endpoints. It's easy to get started with this API, you simply click a few buttons to install it. However, you have less control over how clients may access your content in MarkLogic. It consists of a set of pre-defined endpoints with pre-defined behaviors and semantics. This can make it more difficult to map your abstractions onto the generic API or use resource naming schemes that matche your domain. Although it is possible to impose some access controls, it can be very difficult to enforce a specific interface contract using this API. That is, preventing clients from "going through the back door" to access content directly thereby exposing internal details (or changing things) that you may want to keep private. As designed, this API depends on a middle tier to define the contract and mediate access to MarkLogic, which makes it less suitable as a replacement for XCC (ironically, because it has too much in common with XCC). Using a REST interface to replace XCC should hide the details of MarkLogic from your application, not advertise them. Doing so retains the coupling you wish to avoid by moving to REST.

---

[12] https://github.com/marklogic/ml-rest-lib

[13] https://github.com/overstory/rest-template

[14] https://github.com/xquery/rxq

[15] https://docs.marklogic.com/REST

Conclusion

At the time XCC was designed and implemented (2005/2006, by the author of this paper), vendor-supplied connectors with specific language bindings - like JDBC - were the accepted best practice. In the intervening years software architecture has come to favor a loosely-coupled service orientation, with REST as the ascendent exemplar of that trend.

XCC has been extremely successful. Most major MarkLogic-based applications incorporate it and it has served well since it was introduced, with minimal updates to the API over the years. It performs well (the aforementioned startup penalty for C# notwithstanding) and is easy for developers to use. There is nothing wrong with XCC, but it is a specialized tool that incurs an unnecessary language dependency.

With one exception, an HTTP-based REST approach can do everything XCC can do, without any language dependencies, while remaining loosely coupled and keeping your options open. There are several open source tools available such as Corona and the MarkLogic REST library that make it very easy to do REST on the MarkLogic side and a wealth of REST libraries available for the client side.

Doing REST directly on MarkLogic rather than using XCC has a great many advantages and just a few disadvantages. A RESTful architecture lets you compose system components easily and in innovative ways. It's the best way to future-proof your valuable MarkLogic investment, maximize the benefits of that investment and make your life easier in the bargain.

For more information or if you have any questions about the material in this paper, please contact the author, Ron Hitchens at: ron@overstory.co.uk.