RESTful XQuery
Standardised XQuery 3.0 Annotations for REST

Adam Retter
Adam Retter Consulting
<adam@adamretter.org.uk>

January, 2012

Abstract

Whilst XQuery was originally envisaged and designed as a query language for XML, it has been adopted by many as a language for application development This, in turn, has encouraged additional and diverse extensions, many of which could not easily have been foreseen.
This paper examines how XQuery has been used for Web Application development, current implementation approaches for executing XQuery in a Web context, and subsequently presents a proposal for a standard approach to RESTful XQuery through the use of XQuery 3.0 Annotations.

Keywords: XQuery 3.0, Annotations, REST, HTTP, Standard

1. Introduction

1.1 Background

XML Query Language (XQuery) was originally born from several competing query languages for XML[1]. All of these languages had in common the noble yet limited goal of querying XML. They focused on XML as a read-only store for data. In addition, whilst several of these predecessors recognised the Web as a critical factor, like their successor XQuery, none of them attempted to implement constructs in the language that supported use as a (Web) server-side processing language.

With the adoption and use of XQuery, because of its functional nature and module system which permit the organisation of code units, people attempted to write complex processing applications in XQuery. As the limits of what was achievable in XQuery were tested, real world scenarios emerged which called for additional XQuery facilities, resulting in extension standards: XPath and XQuery Update[2] and XQuery Full-Text[3].

Triggered by XQuery users developing increasingly complex applications in XQuery, and the understanding that XQuery could easily produce XHTML, an XQuery processor operating on an XML Database was for the first time in 2003 coupled with a Web Server and REST interface in the eXist Native XML Database project[4] [5].
With the advent of being able to use XQuery as a server-side processing language, developers were soon building complete data driven Web Applications entirely in XQuery.

Today most XQuery vendor's products operating on collections of XML documents, provide some mechanism for invoking this processing from the Web by URI[6–10].

The W3C XQuery Working Group has itself recognised the value in XQuery as a general purpose processing language through a new extension standard which enhances XQuery for this purpose: XQuery Scripting Extensions[11].

1.2 Problem Statement

The value in using XQuery as a server side processing language is well recognised by both vendors, users and the XQuery Working Group. However, to date there has been no effort to standardise how XQuery may be invoked in a Web context. Presently, each vendor has their own non-standard approach to wiring Web requests and XQuery scripts together; which in-turn causes developers to create non-portable platform dependent XQuery when coding for the Web.

Sadly non-portable XQuery code that relies on vendor extensions or mechanisms, limits and fragments the XQuery community; it is much harder to share useable code and promote an environment of learning from peers and building on existing work, when code that one would hope should run on any XQuery processor simply cannot.

Efforts such as the EXPath[12] and EXQuery[13] projects have attempted to promote portable XQuery code by creating community standardised versions of existing vendor extensions. The EXPath project has attempted to standardise a HTTP Client[14] request module for XQuery. However, there is no such vendor independent standard for invoking server-side XQuery on the Web.

1.3 Contributions

The W3C XQuery 3.0 language specification[15] (currently a Last Call Working Draft) introduces several new features to XQuery. This paper proposes a new vendor agnostic standard for invoking XQuery from the Web based on the new feature of Annotations present in XQuery 3.0.

1.4 Outline

This paper first attempts a brief description of the fundamentals of XQuery and REST and why it is desirable to combine these in Section 2. Section 3 reviews and critiques several current approaches. Based on this knowledge, a standard for a vendor agnostic approach is proposed in Section 4. Section 5 describes a concrete technical implementation of the proposed standard and Section 6 discusses the conclusions of this work and possible future work.

2. Fundamentals

2.1 XQuery

XML Query Language (XQuery) is a W3C Recommendation[1] for writing queries against the XPath and XQuery Data Model (XDM)[16], which is to say, the logical structure of XML documents. Now in its Second Edition of Version 1.0, the W3C XQuery Working Group is currently finalising the new upcoming version, numbered 3.0[15]. XQuery is a Turing-complete[17] functional programming language which is centred around FLWOR (For Let Where Order-by Return) statements which utilise path expressions to address the XDM. XQuery code may be grouped into functions and modules, of which there is always a main module where processing begins.

Whilst XQuery is most usually used for querying XML documents, there are several serialization options for the results of an XQuery: XML, XHTML, HTML and Text; The specification of the serialization mechanism has been formalised in XQuery 3.0. The capability to produce XHTML, HTML from querying XML documents whilst remaining in the same data type model, removes the impedance mismatch between the data query language and application programming language that is present in other environments[18]; XQuery therefore lends itself well to rapid (X)HTML generation. The ability to also serialize results as Text allows for dynamic generation of CSS, JavaScript and JSON amongst others, indeed some vendors already provide a JSON serialization option for their XQuery implementations.

Example 1. XQuery 3.0: Querying XML and generating some simple XHTML

```
xquery version "3.0";

declare namespace output = "http://www.w3.org/2010/xslt-xquery-serialization";
declare option output:method "xhtml";
declare option output:version "1.1";
declare option output:doctype-system "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd";
declare option output:doctype-public "-//W3C//DTD XHTML 1.1//EN";

<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <title>Word of the Day</title>
    </head>
    <body>
        <h1>Hello</h1>
        <p>Todays special word is:
        <span>{doc("mydoc.xml")//word[xs:date(@date) eq current-date()]/text()}</span>
        </p>
    </body>
</html>
```

2.1.1 XQuery 3.0 Annotations

Whilst XQuery 3.0 introduces many new features, an understanding of Annotations in XQuery 3.0 is fundamental to the contribution of this paper. Annotations declare properties of functions or variables, zero or more annotations may be added to a function or variable declaration. Annotations start with the '*%*' character and consist of an expanded qualified name and an optional value, the value being a sequence of literals.

Example 2. XQuery 3.0 Annotations

```
xquery version "3.0";

declare namespace java = "http://java";

declare
%java:method("java.lang.Math.sin")
function local:calculate-sin($a as xs:double) as xs:double external


<sin>{
    local:calculate-sin(1.4)
}</sin>
```

Apart from the '*%private*' and '*%public*' annotations, no other annotations are defined by the XQuery 3.0 specification. However, the specification states, “*Implementations MAY define further annotations, whose behaviour is implementation-defined*”, and it is this property which this paper exploits to define a standard set of RESTful Annotations for XQuery 3.0.

2.2 REST

Representational State Transfer (REST) is an architectural style developed by Dr. Roy Fielding for his doctoral thesis. REST describes the architectural design principles of the evolved Web and remains abstract from the implementation: “*REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state*”[19]. Applications that adhere to REST are described as RESTful.

The Web utilises Hyper-Text Transfer Protocol (HTTP) as its transport and Uniform Resource Identifier (URI) as its addressing mechanism, and can be described as RESTful.

However, Web Sites built with HTML (and possibly JavaScript) typically only use a subset of the full HTTP capabilities; commonly just HTTP verbs GET and POST, with a blanket HTTP Accept header which allow for the retrieval of resources and transmission of simple form data and encoded files. In contrast, RESTful Web Services implemented over HTTP, may use the full range of HTTP verbs and HTTP content negotiation of resources to retrieve or store representations against rich descriptive URI namespaces. Such RESTful applications benefit over the incumbents (SOAP, RPC, etc.), in that additional vocabularies and technology are not required; they, like the Web, are submittable to the same caching, transformation and intermediate security mechanisms due to the common layered architecture.

Arguably, RESTful HTTP Web Services, through their use of descriptive verbs (e.g. GET, PUT, POST, DELETE, OPTIONS etc.) and transfer of representations, are perhaps more applicable to Document Management Systems, such as XML Databases, than they are to the hypermedia systems like the Web which are largely still read-only.

It is the RESTful HTTP Web Service style of REST that shall be considered from hereon in this paper.

2.3 XQuery and REST together

XQuery processors are often integrated with large systems that maintain many XML documents, such as XML Databases. Considering the following as Resources in REST terminology, XML documents, or representations of, such as the output of an XQuery; RESTful HTTP Web Services have many desirable and symbiotic properties for addressing and describing manipulations of said XML resources, whilst XQuery affords the implementation power to realise such implementation.

3. Review of Current Approaches

Current approaches to invoking XQuery in a RESTful manner using HTTP are examined in this section.

There are many vendors of XQuery processors[21], most of the Document Repository or XML Database vendors with such processors provide HTTP API's to the document store with the facility to invoke XQuery scripts remotely against the document store. The notable exception to this is Servlex, an implementation of EXPath Webapp Module. Rather, it is concerned with mapping URI's to the invocation of any Servlex Servlet (i.e. an XPath function, XSLT named template, XQuery main module, XSLT stylesheet, XProc pipeline or XProc step)[22] without concern for document stores.

Providing a complete review of all current approaches would be too resource intensive and lengthy for this paper; as such, four vendors' products have been chosen for examination. These have been chosen to represent products which may contrast, yet are widely used in the industry, and/or provide justification and inspiration for developing a standard approach based on XQuery 3.0 Annotations.

3.1 eXist-db

eXist-db was chosen because 1) it is arguably the most popular and widely used Open Source Native XML Database, 2) it appears to have been the first such product to offer the facility to invoke XQuery via HTTP and 3) it has a history of transparently mapping XQuery functions directly to HTTP APIs, in the form of its XQuery SOAP Server[23].

eXist-db provides two mechanisms for invoking XQuery in a RESTful manner over HTTP.

3.1.1 REST Server

The first mechanism relies on a REST Server which is embedded into eXist-db. This REST Server allows any resource stored into the database to be addressed by a URI, and the database content to be manipulated by HTTP POST, PUT and DELETE, but it does not support content negotiation.

For example if one wanted to retrieve the XML document on Hamlet from the Classics, Shakespeare collection, a HTTP GET on http://localhost:8080/exist/rest/db/classics/shakespeare/hamlet.xml may retrieve the desired document.

eXist-db provides three proprietary XQuery function modules for accessing the HTTP context of the REST Server, namely, Request Module, Response Module and Session Module. XQuery invocation by REST Server is possible by:

1. Sending an XQuery to the REST Server for execution against a database collection or document URI context.
   a) For HTTP GET a parameter may be appended to the URL query-string which contains an XQuery to execute. For example to retrieve speeches given in Hamlet - http://localhost:8080/exist/rest/db/classics/shakespeare/hamlet.xml?_query=//speech or for example, to retrieve speeches given in any Shakespeare script - http://localhost:8080/exist/rest/db/classics/shakespeare/?_query=//speech
   b) For more complex XQuerys, the XQuery may be wrapped in a CDATA section of a simple XML document and sent to the server URI by HTTP POST.
2. An XQuery main module, and supporting modules may be pre-stored into the database. This approach is akin to stored procedures in a relational database, however the REST server makes the main module invokable by URI. For example performing a HTTP GET or POST against http://localhost:8080/exist/rest/db/some-script.xqy would invoke the some-script.xqy XQuery main module.

The REST Server is a powerful mechanism that has been used to build entire and complex enterprise web applications in pure XQuery, however URI's serviced by the REST Server implicitly mirror the database collection hierarchy in eXist-db which is not always desirable. Hyperlinking to child resources and collections are implicit in the response (when requesting a database or collection), thus supporting the REST promise of hypermedia for application state. However, when requesting resources themselves, there is no mechanism for hyperlinking to related resources, a disadvantage when compared to xDB's XML REST Framework in §3.3.3. In addition parameters can only be passed to XQuery modules via HTTP through URL query-string parameters or POST'ed form fields, which leads to complex URLs. This makes building applications with simple, logical and descriptive URI schemes a challenge.

3.1.2 XQuery URL Rewriting

XQuery URL Rewriting filters all HTTP requests to eXist-db. If a HTTP request URI matches a prefix defined in a configuration file, then that URI is mapped to a collection in the database. Typically that collection may then contain a Controller written as an XQuery main module and named '*controller.xql*'. If a Controller is found, then processing of the HTTP request is handed to the XQuery, and the XQuery has absolute access to the HTTP request and response and may take any action it wishes, returning any desirable HTTP response and status code, or handing the request off to another XQuery or Servlet.

Example 3. eXist-db, URL Rewriting Configuration snippet

```
<forward pattern="/webdav/" servlet="milton"/>
<forward pattern="/atom/" servlet="AtomServlet"/>
<root pattern="/solutions" path="xmldb:exist:///db/apps/local/solutions/"/>
<root server-name="www.example.com" pattern="/*" path="xmldb:exist:///db/com/example/www/"/>
```

Example 4. eXist-db, URL Rewriting Controller snippet (controller.xql)

```
(: homepage :)
if($exist:path eq "/" or $exist:path eq "/home.xml") then
    template:process-template($rel-path, $exist:path, $DEFAULT-TEMPLATE, ($menus,
fn:doc(fn:concat($rel-path, "/home.xml"))))

(: login page :)
else if($exist:path eq "/login") then
    if(security:login(request:get-parameter("username", "unknown"), request:get-
parameter("password", "unknown")))then
        local:redirect("entry/browse")
    else
        local:redirect("./?login=failed")

(: user sign-up page :)
else if($exist:path eq "/register") then
    if(request:get-method() eq "GET")then
        template:process-template($rel-path, $exist:path, $DEFAULT-TEMPLATE, ($menus,
fn:doc(fn:concat($rel-path, "/registration.xml"))))
    else if(request:get-method() eq "POST")then
        let $request-data := request:get-data()/user return
            if(security:register-user($request-data))then
                local:redirect("entry/browse")
            else
            (
                (: could not register the user - xform will show error :)
                response:set-status-code(400),
                <message>Unable to register the user '{$request-data/username}'</message>
            )
    else
        local:ignore()

else
    local:ignore()
```

XQuery URL Rewriting is much more flexible than the REST Server as it decouples the logical application URI space from the logical database URI space, it also allows you complete control over the lifecycle of HTTP Requests made to the database. However, with this flexibility comes complexity. For real-world enterprise applications the Controller can end up becoming several thousand lines of XQuery code, which is really just encoding if/else statements to match URI patterns and/or HTTP actions. The order of statements in the Controller becomes a concern, and this non-declarative approach becomes very hard to maintain and debug as the code size grows.

3.2 MarkLogic

MarkLogic was chosen because 1) it is almost certainly the most successful commercial Native XML Database Server, 2) whilst a newer creation than eXist-db, it ultimately provides similar REST capabilities but through a different approach, and in addition has layered some new frameworks atop these, 3) MarkLogic caters for the enterprise market whereas Open Source projects like eXist-db are better known in smaller, educational or public institutions.

MarkLogic provides three mechanisms for invoking XQuery in a RESTful manner over HTTP.

3.2.1 HTTP App Server

MarkLogic's HTTP App Server varies somewhat from that of eXist-db's REST Server. MarkLogic differentiates between a Modules database and a Content database.

Resources stored into the Content database (i.e. XML documents) are not directly accessible by URI from the HTTP App Server. They may only be accessed via an XQuery Module. Resources stored into the Modules database (i.e. XQuery Modules, or Binary files such as JPEG, CSS etc) are all accessible by URI. An advantage over eXist-db is that resources stored into the database may be assigned an arbitrary logical URI, rather than the URI being representative of a logical collection or folder hierarchy.

XQuery invocation by HTTP App Server is possible by pre-storing XQuery modules into MarkLogic's Modules database. Like eXist-db, this is akin to stored procedures in a relational database. Likewise, the HTTP App Server makes the main module invokable by URI. For example, performing a HTTP GET or POST against http://localhost:8060/some-script.xqy would invoke the some-script.xqy XQuery main module.

Unlike in eXist-db's REST Server, in MarkLogic's HTTP App Server, resources in the Modules or Content databases cannot be manipulated by HTTP POST, PUT or DELETE. Similarly there is no support for content negotiation by default. Like eXist-db, passing parameters to XQuery modules via HTTP must be achieved through URL query-string parameters or POST'ed form fields. Similarly, MarkLogic also provides a proprietary XQuery function module for accessing the context of the HTTP interaction, namely the functions, xdmp:get-request-*, xdmp:set-response-*, xdmp:get-session-* and xdmp:set-session-*.

When comparing the HTTP App Server against eXist-db's REST Server, as the default out-of-the-box experience, it is less RESTful in its approach, as it does not support URI addressing of document resources or manipulation by HTTP methods. However, it does not purport to being an advanced REST server, and more complex RESTful requirements may be addressed by additional mechanisms discussed below.

3.2.2 URL Rewriting

URL Rewriting may be setup independently for each HTTP App Server. When enabled, all HTTP requests to a specific HTTP App Server are filtered by executing the configured XQuery main module for each HTTP Request that is received. This XQuery module is typically called '*url_rewrite.xqy*' and is responsible for rewriting URL's. The approach is much simpler than that of eXist-db described in §3.1.2, and the purpose of the script is to solely return a single String value which is the rewritten URI path.

Example 5. MarkLogic, URL Rewriting snippet (url_rewrite.xqy)

```
let $url := xdmp:get-request-url() return

    (: homepage :)
    if(fn:matches($url, "^/$") or fn:matches($url, "^/home.xml$")) then
        "/home.xqy"

    (: login page :)
    else if(fn:matches($url, "^/login$")) then
        "/login.xqy"

    (: user sign-up page :)
    else if(fn:matches($url, "^/register$")) then
        if(xdmp:get-request-method() eq "GET")then
            "/registration-form.xqy"
        else if(xdmp:get-request-method() eq "POST")then
            "/sign-up.xqy"
        else
            "/nowehere.html"
    else
        "/nowehere.html"
```

URL Rewriting adds greater flexibility to MarkLogic's HTTP App Server by decoupling the logical URI space from the actual modules database URI space. It is much simpler than the approach taken by eXist-db, with the disadvantage in functionality leading to the advantage of there being less non-declarative XQuery URL Rewriting code to maintain. However, it results in the same problem, which is that large applications will call for unwieldy and complex rewriting rules, creating a spaghetti of if/else statements, which will ultimately obscure the very URI's that are of such importance.

3.2.3 XQuery Libraries

A number of additional XQuery libraries such as the MarkLogic REST Library or Corona[24] are available for MarkLogic App Server. Each of these libraries may be installed as a URL Rewriter for a particular HTTP App Server, and in addition provide an XQuery module whose functions may be invoked from your own XQuery code to simplify handling of incoming HTTP REST requests.

These libraries substantially improve upon the standard URL Rewriting, by allowing you to have declarative XML files that define the URL mappings, and can make working with RESTful requests much simpler. However, they still have the disadvantages of, requiring additional XQuery glue code, and moving the declarative URL mappings away from the end-points of execution, which could lead to difficulty when identifying which URI rules apply to which code sites.

Example 6. MarkLogic, REST Library, declarative XML URL rewriting snippet

```
<options>
    <request uri="^/(.+)/act(\d+)$" endpoint="/endpoint.xqy">
        <uri-param name="play">$1.xml</uri-param>
        <uri-param name="act" as="integer">$2</uri-param>
    </request>
    <request uri="^/(.+)/?$" endpoint="/endpoint.xqy">
        <uri-param name="play">$1.xml</uri-param>
    </request>
</options>
```

3.3 EMC xDB

EMC xDB was chosen because 1) it has a different technical heritage, as it was originally designed as a product to be embedded within Java Applications, 2) it purports to be the most widely used XML Native Database (due to its embedded nature), 3) whilst it can operate as a standalone server, it is typical to have to construct your own REST API layer.

xDB provides a single mechanism for invoking XQuery in a RESTful manner over HTTP. In addition, there are also two proscribed self-build mechanisms from EMC publications which will also be briefly reviewed.

3.3.1 xDB REST API

With the release of version 10.1 of xDB, a Web Client was provided for the standalone server, which allows administration and management of the database from a Web Browser. This Web Client is underpinned by the xDB REST API[25]. Whilst the xDB REST API is mentioned briefly in the documentation, and there is limited documentation accompanying an xDB installation, it is understood that this API is not intended for production consumption. However, there are some aspects of the xDB REST API which make it interesting to consider here.

This xDB REST API is most similar to eXist-db's REST Server in that it allows any resource stored into the database to be addressed by a URI, and the database content to be manipulated by HTTP PUT and DELETE. An advantage of the xDB REST API when compared to the others, is that it does support some content negotiation (i.e. XML or JSON) for retrieving lists of what is present in the database, and also metadata about the resources themselves. For example if one wanted to retrieve the XML document on Hamlet from the Classics database, a HTTP GET on http://localhost:1280/federation/classics/shakespeare/hamlet.xml may retrieve the desired document. However, for our focus on XQuery, a major disadvantage is that there are no XQuery function modules for accessing the HTTP context in xDB. It is also impossible to invoke the execution of XQuery main modules stored into the database, as calling these modules by URI simply results in their textual code content. XQuery invocation by xDB REST API is possible by:

1. Sending an XQuery to the REST API for execution against a database, library (a.k.a. collection) or document URI context.
    a) For HTTP GET a parameter may be appended to the URL query-string which contains an XQuery to execute. For example to retrieve speeches given in Hamlet – http://localhost:1280/federation/classics/shakespeare/hamlet.xml/_xquery?query=//speech or for example, to retrieve speeches given in any Shakespeare

script – http://localhost:1280/federation/classics/shakespeare/_xquery?query=//speech

b) For more complex XQuerys, the XQuery may be wrapped in a CDATA section of a simple XML document and sent to the server URI by HTTP POST.

The REST API is a capable mechanism and, whilst not supporting the execution of pre-stored XQuery main modules, it does have in its favour support for content negotiation, and hyperlinking to child resources and libraries are implicit in the response (when requesting a database or library), thus supporting the REST promise of hypermedia for application state. However, when requesting resources themselves, there is no mechanism for hyperlinking to related resources, a disadvantage when compared to EMC's xDB XML REST Framework described in §3.3.3. URI's serviced by the REST API, implicitly mirror the database hierarchy in xDB, which is not always desirable. In addition parameters can only be passed to the XQuery module via XQuery external variables, which may be bound in the XML document, when using the HTTP POST approach. This coupled with the lack of XQuery functions for accessing the HTTP context, makes building applications in pure XQuery impossible, and building applications with simple, logical and descriptive URI schemes a difficult challenge.

3.3.2 Implementing a RESTful API (JAX-RS)

Published by EMC is an approach designed by Martin Probst[10] which couples together xDB standalone server with JAX-RS (Java API for RESTful Web Services) for the purposes of database management and XQuery execution. The article describes implementing an example REST API for use with xDB, and allows anyone to build upon this. In fact the API described in the article is almost certainly the exact API discussed in §3.3.1. However, it is not the similarity that we are interested in, but rather the idea that JAX-RS, which is a REST API framework, is itself used to produce a domain specific REST API with relevance for XQuery. The ability to use JAX-RS in this way is conceptually no different to using the URL Rewriting support in eXist-db or MarkLogic for writing a domain specific REST API.

The reason for looking at JAX-RS here is that, like URL rewriting in eXist-db and MarkLogic, or WebApp Descriptors in Servlex, JAX-RS provides a declarative mechanism for defining the URL rewriting/mapping rules. However, JAX-RS has a major advantage over the other approaches, which enables the declarative URL rewriting rules to live alongside the code sites of execution. This is achieved through annotations. These annotations should allow easy determination of code and URI relationships due to their proximity to the executable code sites, whilst the declarative nature should make reasoning about the mappings simple.

Example 7. xDB, JAX-RS API Snippet

```
@GET
@Path("_query")
public Response doXQuery(@QueryParam("xquery") String query, @Context HttpServletRequest context) {

   //get the xquery sent to us
   if (query == null)
     throw new WebApplicationException(Response.status(Status.BAD_REQUEST).entity(
        "xquery parameter is required").build());

   //execute the query
   String results = executeXQuery(query, context.getParameterMap());

   //return the results in the http response
   String contentType = xquery.getOptions().get(new QName(XHIVE_NS, "content-type"));
   return Response.ok(results, contentType).build();
}
```

The JAX-RS Java annotations *@GET* and *@Path("_query")* imply that, should the HTTP application server encounter a HTTP GET for the path '*_query*', then the code in the function '*doXQuery(...)*' should be executed. In JAX-RS paths are always defined relative to the server's HTTP URI context. The *@QueryParam("xquery")* annotation injects the value of the HTTP URL query-string parameter named '*xquery*' into the String function parameter called '*query*'.

3.3.3 XML REST Framework

The XML REST Framework[20], an approach advocated by Cornelia Davis at EMC, advances the JAX-RS approach described in §3.3.2, to produce a framework for building domain specific REST API's for xDB. Resultant APIs do not permit the direct execution of XQuery by HTTP, rather the framework indirectly executes developer defined XQuery code for each REST API function invoked. Each REST endpoint is coded as a POJO (Plain Old Java Object) in Java with JAX-RS annotations, and it is this which mediates the execution of the XQuery and the marshalling and de-marshalling of the XQuery external variables and output over HTTP for the client.

Arguably, the most interesting aspect of this framework and its approach is that it recognises that whilst REST APIs are good at delivering representations or resources, they often lack hyperlinking in the returned resource content to enable further resource URI's to be autonomously determined.

The initial technical implementation detail of the XML REST Framework[26], describes the situation thusly, *"(as is sadly common in many RESTful services today) the consumer has no choice but to leverage the knowledge of these URI templates"*. In further implementation[27], the XML REST Framework is extended to allow XSLT to be applied to content before it is returned as the result of a REST response to a request; XSLT can be used to modify XML content responses and insert hyperlinks to other related resource representations. Whilst XSLT is used in this instance, this could easily be substituted for XQuery. The XML REST Framework has since been advanced, so that the REST end-points in Java now call an XProc pipeline rather than the XQuery directly. The advantage of this is that multiple processes can be placed inside the XProc pipeline, including XQuery. The disadvantage is the XQuery is moved further away from the HTTP context and this could make understanding the code used to fulfil the request for a specific URI more convoluted. The XML REST Framework for xDB stands alone from all other reviewed vendor approaches to REST in that it attempts to address the hyper-linking tenant of the REST architecture. The approach to embedding hyperlinks with XSLT is however only applicable when hypermedia instances such as XML or XHTML are used for application state.

3.4 Servlex

Servlex was chosen because 1) it is an implementation of a public common standard (the EXPath Webapp Module) for wiring HTTP Requests to XML processors, 2) unlike others, it does not require a Document Repository or XML Database, and 3) it provides a purely declarative approach to URI mapping, instead placing constraints on the interface with the underlying XML processing code. Servlex is the reference implementation of the EXPath Webapp Module.

Servlex provides a single mechanism for wiring HTTP Requests to XML processors, this mechanism is the declarative vocabulary of the Webapp descriptor, an XML file named '*expath-web.xml*'. This descriptor must be placed inside a larger EXPath Package[28]. Whilst Servlex supports mapping URI's to XPath, XQuery, XSLT and XProc code, inline with the focus of this paper, we will only consider herein Servlex's ability to interoperate with XQuery.

Example 8. Servlex, Webapp Descriptor snippet (expath-web.xml)

```
<webapp xmlns="http://expath.org/ns/webapp/descriptor"
    xmlns:app="http://example.org/ns/my-website"
    name="http://example.org/my-website"
    abbrev="myweb"
    version="1.3.0">
    <title>My example website</title>

    <resource pattern="/style/.+\.css" media-type="text/css"/>
    <resource pattern="/images/.+\.png" media-type="image/png"/>

    <servlet>
        <xquery function="app:product-page"/>
        <url pattern="/product/(.+)">
            <match group="1" name="id"/>
        </url>
    </servlet>
</webapp>
```

The example descriptor would map the URL starting with '*/product/*' to the function '*app:product-page*' in the XQuery module of the namespace '*http://example.org/ns/my-website*', which would need to be pre-defined in the EXPath Package descriptor '*expath-pkg.xml*'. Servlex places constraints on its interface with the XQuery processor[29], so an example module showing a possible function is illustrated below.

Example 9. Servlex, XQuery Module (products.xqy)

```
module namespace app = "http://example.org/ns/my-website";

declare namespace web="http://expath.org/ns/webapp";

declare function app:product-page($request as element(web:request), $bodies as item()*) as
item()* {
    (
        <web:response status="200" message="Ok"/>
        ,
        <debug>Chosen Product ID was:
            <id>{/web:path/web:match[@name eq 'id']/text()}</id>
        </debug>
    )
};
```

The declarative nature of the URL mapping in Servlex is an advantage over the eXist-db and MarkLogic mechanisms, as it allows the URIs to remain outside of the code and easily visible and maintainable; however, it is a disadvantage when compared with the JAX-RS approach recommended for use with xDB in §3.3.2, as the declarative URI patterns are moved away from the code execution sites. Another disadvantage of the Servlex approach is that it enforces an interface which must be present in XQuery function signatures which are URI mapped by Servlex. This can lead to the creation of adapter functions and glue code to act as a facade for Servlex to interface with the desired XQuery module functions to be invoked. Apart from URI mapping, for identification of resources, Servlex leaves all other REST requirements to the implementer of the XQuery processing code, which allows great flexibility at the expense of the time required for implementation of a Servlex Servlet.

3.5 Summary

Whilst all of the reviewed vendors' products differ in their approaches and enabling facilities for developers to build RESTful applications in XQuery, each offers at least one mechanism for XQuery to be executed by HTTP.

Table 1. Summary of compliance with RESTful interface constraints

		Identification of Resources	**Manipulation, Representations**	**Self-Descriptive Messages**	**Hyper-media for Application State**
eXist-db	**REST Server** §3.1.1	Direct	GET/POST/PUT/DELETE XQuery representations	Yes	Browsing XQuery developer option
	XQuery URL Rewriting §3.1.2	Direct and Indirect	GET/POST/PUT/DELETE XQuery representations Content negotiation	Yes	XQuery developer option
MarkLogic	**HTTP App Server** §3.2.1	Indirect - Only XQuerys	GET XQuery representations	Yes	XQuery developer option
	URL Rewriting §3.2.2	Indirect	GET XQuery representations	Yes	XQuery developer option
	XQuery Libraries §3.2.3	Indirect	GET/POST/PUT/DELETE XQuery representations Content negotiation	Yes	XQuery developer option
EMC xDB	**xDB REST API** §3.3.1	Direct	GET/POST/PUT/DELETE XQuery representations Content Negotiation	Yes	Browsing XQuery developer option

	JAX-RS §3.3.2	Indirect	GET/POST/PUT/DELETE Java programmed representations Content Negotiation	Yes	Java developer option
	XML REST Framework §3.3.3	Indirect	GET/POST/PUT/DELETE Indirect XQuery representations Content Negotiation	Yes	Yes. Injected hyperlinking via. XSLT
EXPath	**Servlex** §3.4	Indirect – Only XQuery/XSLT/ XProc	GET/POST/PUT/DELETE Indirect XQuery/XSLT/XProc representations	Yes	XQuery/XSLT/XProc developer option

eXist-db's REST Server and EMC's xDB REST API provide the most RESTful out-of-the-box experience without the need to write additional code because they permit manipulation of the database (including XQuery main modules) by the use of HTTP verbs. In addition, when browsing resources hyper-media state with hyper-linking is automatically delivered. These features meet the RESTful architectural requirements of: Identification of Resources, Manipulation of Manifestations, Self-Descriptive Messages and some limited support for Hyper-Media for Application State. eXist-db's REST Server has the slight advantage for building applications, because 1) XQuery can be pre-loaded into the database through the HTTP verbs PUT or POST and then invoked with HTTP GET, and 2) XQuery function modules are provided to support the HTTP context.

However, if we compare all options reviewed available for building XQuery RESTful Web Applications the conclusion is different. We will dismiss EMC xDB because without also writing Java alongside XQuery, its REST API is too limited. There is no access to the HTTP context from XQuery, and there is no support for URI Rewriting to decouple the application URI space from the physical database layout.

Both the URL Rewriting capabilities of eXist-db and MarkLogic are impressive, however by default they both use an XQuery main module to do the URL rewriting, which as previously discussed can lead to maintainability issues. MarkLogic probably has the advantage that there are several XQuery Libraries that extend their URL rewriting mechanism to both allow URL mappings to be declaratively defined, whilst also simplifying many common HTTP/REST functions required by developers. Servlex, like MarkLogic's XQuery Libraries, provides a declarative approach to URL rewriting, however unlike MarkLogic and eXist-db the mechanisms defined for accessing the HTTP context from the processors is still embryonic and not widely tested. There is a problem with the declarative URL rewriting approach in MarkLogic and Servlex in that the URL rewrite rules are moved away from the code sites, which in complex applications can make the mapping between functions and URI's difficult to maintain. Conceptually this is solved in JAX-RS as used in xDB as the URI's are still declarative yet precede the code site.

Section 4. proposes a set of Annotations for XQuery 3.0 which enable the best features found in the reviewed products above, but that maintain both the advantages of declarative URI Rewriting and keeping URI rules close to the code site.

4. Standardised XQuery 3.0 Annotations for REST

Herein we present a set of XQuery 3.0 Annotations to enable the construction of RESTful Web Services in XQuery. The goals of our approach are -

1) Interoperability. It is envisaged that this paper will provide the basis for a public XQuery community standard, which if adopted by vendors, would permit portable XQuery Web Services. To enable this, an implementation agnostic description is provided.

2) Simplicity for XQuery developers. Developers should not have to maintain external or complex code for wiring RESTful services to XQuery functions.
3) Technical improvement. Having reviewed existing approaches in §3, we build upon the best aspects of each vendor's approach.

4.1 Approach

For mapping RESTful requests to executable code, the declarative approach discussed in §3, is felt to be the most advantageous, particularly when the declaration is an annotation on the code site (§3.3.2). Therefore, our approach is heavily influenced by that of JAX-RS[30]. However, we simplify and deviate from JAX-RS due to the language structure differences between Java and XQuery. Where JAX-RS describes Resource Classes and Resources Methods for Java, in XQuery we simply use the term Resource Function; for mapping HTTP calls to XQuery invocation, our unit of granularity is the XQuery function. Through the use of annotations on functions in XQuery, we declaratively mark-up the HTTP capabilities of a function.

To minimise refactoring by developers when adding annotations to existing code, two measures must be respected by implementations:

1) Implementations must support annotated functions which have additional function parameters which are not annotation mapped, providing the cardinality type of those un-mapped parameters accepts an empty sequence
2) Implementations must not enforce the order of function parameters. Whether mapped by annotations or not is unimportant, as annotations explicitly name the parameters to which they are mapped.

For the purposes of this paper, HTTP Multipart Requests and Responses are considered out of scope. However, some attention has been paid to not preventing support for these in future, and this is briefly discussed in §6.2.

4.2 Resource Functions

A Resource Function is an XQuery function which has been marked up with RESTful web service annotations. These annotations indicate to a processor that when presented with a RESTful web service request, that matches the constraints indicated by the annotations, the function should be invoked and the result returned as the result of the service request.

Whilst the concept of dynamically and transparently mapping web service calls to XQuery function invocation has previously been proved[23], this is the first time XQuery annotations have been used to provide a standardised and developer controllable approach.

4.3 Resource Function Constraints

Constraints restrict the service requests that a Resource Function may process.

4.3.1 URI Path and Templates

A '*Path Annotation*' provides for URI templates and allows the URI of a RESTful web service to be mapped to a Resource Function. A Resource Function must contain a single path annotation. Additional annotations may also be used to constrain the Resource Function.

The path annotation is named '*%rest:path*' and takes a single mandatory literal string, which describes the URI path for this service. The URI path is relative to a base URI defined by the implementation.

The URI path itself may contain zero or more URI templates which denote path segments that map to named function parameters. A URI template, has the syntax '*{$fn-param-name}',* where '*fn-param-name*', is the name of a parameter to the annotated function, whose value should be taken from the path. Parameters addressed by URI templates, must meet the following constraints:

1) Cardinality that allows for an atomic value, otherwise an error should be raised by the implementation.
2) Type that inherits from xs:anyAtomicType, otherwise, an error should be raised by the implementation. In addition, conversion from the URI segment string to the required type should be performed at run-time, and an error raised if conversion is impossible.

Example 10. XQuery Path Annotation

```
declare
    %rest:path("/stock/widget/{$id}")
function local:widget($id as xs:int) {

    (: get the widget :)
    fn:collection("/db/widgets")/widget[@id eq $id]
};
```

For example, a HTTP GET on the following URI, would cause the Widget with the '*id*' of '*1981*' to be retrieved: *http://www.widget-factory.com/stock/widget/{$id}*.

When many URI paths are defined, conflicts may occur. It is implementation defined how these should be resolved. However, most specific URI paths must always be evaluated before less specific URI paths, to ensure that lesser paths do not unintentionally consume requests.

4.3.2 HTTP Methods

Resource Functions may be constrained to zero or more HTTP methods by means of a method annotation. Unless otherwise constrained by a method annotation, the path annotation of a Resource Function applies to all HTTP methods.

Annotations are defined for all HTTP 1.1 methods except TRACE and CONNECT. All methods may return resources except for HEAD, which must only return a *rest:response* element.

Example 11. XQuery Method Annotation

```
declare
    %rest:GET
    %rest:POST
    %rest:path("/widget/{$id}")
function local:widget($id as xs:int) {

    (: get the widget :)
    fn:collection("/db/widgets")/widget[@id eq $id]
};
```

Method annotations POST and PUT may take an optional string literal which map the HTTP request body to a named function parameter. The same syntax as that used for URI templates is applied, for example '*%rest:POST("{$request-body")',* would inject the request body into the function through the function parameter named '*request-body*'. The function parameter for the request body must meet the following constraints:

1) Cardinality that allows for one or more of the typed item(s).
2) Typing that is compatible with the request body. The type of the request body is determined by the HTTP Content Type header and may be constrained by means of the '*%rest:consumes*' annotation (see §4.3.3). The interpretation of the request body is similar to that of the EXPath HTTP Client[14]:
    a) If the media-type is a text media-type, the function parameter type will be xs:string.
    b) If the media-type is an XML media-type, the request body is parsed as XML

and the function parameter type will be document-node().

c) If the media-type is a HTML media-type, the content is tidied-up and parsed as XML. The parameter type will be document-node(). The tidying process is implementation defined as no known standard exists.

d) Otherwise, a binary media type is assumed, and the function parameter type will be xs:base64Binary.

4.3.3 Media-Type Capabilities

Support for content negotiation is indirectly provided by two annotations:

1) '*%rest:consumes*', which constrains a Resource Function, by only accepting requests for which one of the defined Internet media-types matches the HTTP Content-Type header of the request.

2) '%rest:produces', which constrains a Resource Function, by only accepting requests for which the mime-type matches the HTTP Accept Header.

Both annotations take a single mandatory String Literal which contains an Internet media-type.

Example 12. XQuery Consumes and Produces Annotations

```
declare
    %rest:PUT("{$body}")
    %rest:path("/widget")
    %rest:consumes("application/xml", "application/atom+xml")
    %rest:produces("application/xml")
function local:widget($body as document-node(element(widget)) {

    (: store the widget :)
    let $db-uri := xmldb:store("/db/widgets", (), $body),

    (: return a hyper-link :)
    $rest-uri := rest:get-absolute-uri("wiget", $body/widget/@id) return
        <a xmlns="http://www.w3.org/1999/xhtml" href="{$rest-uri}">{$rest-uri}</a>
};
```

4.4 Resource Function Parameters

Parameters to Resource Functions are extracted from the RESTful Web Service request and passed in as additional function parameters. Unlike constraints, parameters are always optional. Resource Function Parameters use the same URI template syntax as described in §4.3.1 to map the parameter onto a function parameter. They may also provide a default value should the parameter not be present in the request. Resource Function Parameters always place the following constraints on the function parameters that they map to:

1) Cardinality that allows for: zero or many atomic values in the case of Query, Form or Header parameters, or zero or one atomic values in the case of Cookie parameters, otherwise an error should be raised by the implementation.

2) Type that inherits from xs:anyAtomicType, otherwise, an error should be raised by the implementation. In addition, conversion from the parameter string to the required type should be performed at run-time, and an error raised if conversion is impossible.

4.4.1 Query String Parameters

The annotation '*rest:query-param*' is provided for accessing parameters in the Query String of the URL used for the RESTful Web Service request.

Example 13. Query String Parameter Annotation with a default value

```
declare
    %rest:GET
    %rest:path("/widget/{$id}")
    %rest:query-param("client", "{$client}", “unknown”)
function local:widget($id as xs:int, $client as xs:string*) {

    (: get the widget :)
    fn:collection("/db/widgets")/widget[@id eq $id][@client = $client]
```

```
        };
```

4.4.2 Form Field Parameters

The annotation '*rest:form-param*' is provided for accessing parameters from a HTML Form submitted in the request body of the RESTful Web Service request with the Internet media-type '*application/x-www-form-urlencoded*'.

Example 14. Form Field Parameter Annotation

```
        declare
            %rest:GET
            %rest:path("/widget/{$id}")
            %rest:form-param("client", "{$client}")
        function local:widget($id as xs:int, $client as xs:string*) {

            (: get the widget :)
            fn:collection("/db/widgets")/widget[@id eq $id][@client = $client]
        };
```

4.4.3 HTTP Header Parameters

The annotation '*rest:header-param*' is provided for accessing HTTP Request headers from the RESTful Web Service request. If a single Header field value contains comma separated values, an implementation must extract each value from the comma separated list into an item in the sequence provided to the function parameter.

Example 15. HTTP Header Parameter Annotation

```
        declare
            %rest:GET
            %rest:path("/widget/{$id}")
            %rest:header-param("X-Client-Type", "{$client-type}")
        function local:widget($id as xs:int, $client-type as xs:string*) {

            (: get the widget :)
            fn:collection("/db/widgets")/widget[@id eq $id][@clientType = $client-type]
        };
```

4.4.4 Cookie Parameters

The annotation '*rest:cookie-param*' is provided for accessing HTTP Cookies from the RESTful Web Service request.

Example 16. Cookie Parameter Annotation

```
        declare
            %rest:GET
            %rest:path("/widget/{$id}")
            %rest:cookie-param("locale", "{$locale}")
        function local:widget($id as xs:int, $locale as xs:string?) {

            (: get the widget :)
            fn:collection("/db/widgets")/widget[@id eq $id][@locale = $locale]
        };
```

4.5 Resource Function Serialization

Herein we detail how the results of a Resource Function may be serialized back to an HTTP response for the RESTful web service response.

A Resource Function may return one of three response types:

1) A Resource, i.e. just content.
2) HTTP headers, for example acknowledging the request or providing a status code or additional information.
3) Both HTTP Headers and a Resource.

When returning Resources from Resource Functions, we need to consider how the Resource should be serialized for use in the RESTful web service response. Rather than define a serialization mechanism, XQuery 3.0 in §2.24 of the specification already specifies how serialization of an XDM[31] (i.e. the result of an XQuery) may be controlled by use of Output Declarations. However, Output Declarations are only applicable to XQuery Main Modules, so when re-purposing Output Declarations for serialization the following rules should be applied:

1) If the function is from within a Main Module, and an output declaration exists, then we use this as the default serialization settings for each Resource Function in that module.
2) Output Declarations may be re-written as annotations on any Resource Function e.g. *%output:method("xml").* These annotation output declarations override any defaults from (1). This is also applicable if our function comes from a Library Module, of which the XQuery specification forbids Output Declarations.
3) If no Output Declaration, annotated or otherwise, is provided, then the default is to serialize as XML, UTF-8 encoding, with indenting.

Each of the three possible result types of a Resource Function needs to be handled in a different manner by an implementation, and as such we provide appropriate function signature restrictions, and detail how annotations interact with these:

1) For a function that returns just a resource, either:
   a) If the result type is omitted from the function, it is assumed to be *document-type(element())* or just *element()*, and XML Serialization should be applied to the result of the function. The annotation '*%output:method*', if present, must be set to '*xml*'.
   b) If the result type is present, it must be a type which is compatible with the chosen serialization method, defined by either the XQuery 3.0 output declaration or overridden by the '*%output:method*' annotation on the Resource Function. The default serialization method is XML. If the result type is not compatible with the serialization, an implementation must throw an error.
2) For a function that returns just HTTP headers:

   The result type of the function must be defined as *document-type(element(rest:response)).* Any other annotations that effect the serialization of the result are ignored.
3) For a function that returns both HTTP headers and a resource:

   The result type of the function must be defined as *item()+*. The first item in the result sequence is the HTTP headers i.e. *document-type(element(rest:response)*, the second item in the result sequence is the resource itself. The rules of both (1) and (2) must be applied to the result sequence.

4.5.1 REST Response Format

As described above, a REST Response document may be returned from a function either with or without a Resource. The purpose of this document is to control the REST (in this case HTTP) response sent back to the client of the RESTful web service.

Example 17. REST Response Format

```
<rest:response>
    (http:response?)
</rest:response>

<http:response status?="integer" reason?="string">
    (http:header*)
</http:response>
```

```
<http:header name="string" value="string"/>
```

Should the status be omitted for the response, or a REST Response document not returned from a Resource Function, then the status defaults to 200 OK. It is expected that implementations will make use of sane defaults for HTTP headers as part of their HTTP responses, however any default headers must be overridable by those values set in the REST Response document.

4.6 REST Function Module

RESTful annotations are designed to be hosted by an implementation that provides a Web Server end-point. As the URI Paths of Resource Functions are defined relative to an implementation defined base URI (see §4.3.1), it is impossible without additional support to return hyper-links to the client for Resources. A simple module which provides just two XQuery extension functions is proposed to enable the resolution of absolute URIs, such as the use-case shown in Example 12 of §4.3.3.

```
rest:get-base-uri() as xs:anyURI
```

Returns the base URI of the web context in which the invoking Resource Function is executing. The result of this function should be stable across invocations within an implementation.

```
rest:get-absolute-uri($path-segments as xs:anyAtomicType+) as xs:anyURI
```

Returns an absolute URI by concatenating the base URI as returned by *rest:get-base-uri()* with each path segment in the parameter *$path-segments,* separating each by a '/' character. The result of this function should be stable across invocations within an implementation.

5. Proof of Concept

As a proof of concept, the approach described in §4 has been implemented in the eXist-db Open Source Native XML Database project by the author of this paper. eXist-db is written in the Java programming language, with an XQuery parser written in ANTLR. Both Java and ANTLR were used to implement the proof of concept.

5.1 Implementation

There were several steps to the implementation:

1) Adding support for XQuery 3.0 Annotations to eXist-db's XQuery parser. eXist-db supports XQuery 1.0 with a high-level of compliance, but recently has started a staged adoption of XQuery 3.0 support; annotations were previously missing.
2) Adding support for Multiple-Triggers per-database Collection. It was decided to develop the majority of the RESTful Annotations implementation outside of the core product code. eXist-db provides for database triggers, to enable developers to execute arbitrary code upon various events. Previously, eXist-db only supported a single trigger per database Collection. Rather than modify the XQuery parser further, a database Trigger was used to support this implementation.
3) Implementation of support for the RESTful, the RESTful web services that they declare and the supporting XQuery functions extension module.

The implementation is made up of two distinct processes:

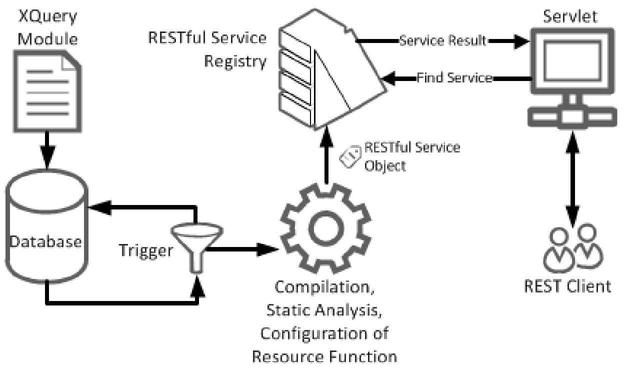
1) When a user stores an XQuery Module into the database, a Trigger is fired. This trigger initiates several steps in order:
    a) Compilation of the XQuery Module into an AST (Abstract Syntax Tree)
    b) Additional static analysis upon any User Defined Functions that have

RESTful annotations. These are known as Resource Functions.

   c) Compilation of a Regular Expression for the Path Annotation of each Resource Function. This expression enables both path matching and extraction of value for URI templates.
   d) Creation of a RESTful Service Object for each Resource Function, which contains the Path Regular Expression, any additional constraint annotations, any optional parameter annotations, a reference back to the database location of the XQuery Module, and the name and arity of the Resource Function (User Defined Function).
   e) Registration of the RESTful Service Object with the RESTful Service Registry for each HTTP Method that it supports.

2) When a HTTP Web Request is made to the system, the receiving Servlet initiates the several steps in order:
   a) Querying the RESTful Service Registry for any Resource Functions which can service the incoming Request.
   b) The Service Registry examines the RESTful Service Objects registered with it to determine if they are applicable to the incoming request. If so, the RESTful Service Object is returned.
   c) A new instance of the XQuery Module referenced by the RESTful Service Object is compiled, and function parameters (as defined by the RESTful annotations) are extracted from the request and mapped into a function call to the underlying User Defined Function.
   d) The User Defined Function is executed. The result is serialized back to the HTTP client based on the rules defined in §4.3.

Figure 1. Implemented Architecture for RESTful XQuery Annotations



5.2 Evaluation

The modifications to eXist-db to support the implementation of RESTful annotations took three working days. Implementation of the RESTful annotations and web server itself took another three days. This figure does not include the Resource Function Parameters defined in §4.4, which will be implemented in the near future.

The implementation certainly feels cleaner than that of the current REST Server and URL Rewrite in eXist-db, and in use is much simpler as a potentially complex controller.xql does not need to be authored or maintained. The new implementation permits for complete declarative decoupling of the URI space from the Resources themselves, without having to declare the URI space externally of the code sites themselves.

In addition the RESTful XQuery Annotations approach, due to its JAX-RS heritage, should feel very familiar to those with an existing knowledge of Java programming, hopefully easing any transition.

A port of the XML Summer School's[32] 'See What I Think' project[33] to the new RESTful Annotations was undertaken. The project is a learning tool originally written in XQuery atop eXist-db's REST Server and URL Rewriting framework. The port was undertaken to understand how the use of Annotations compares to the previous approach. The resultant port uses slightly more lines of XQuery code than the original project, however it is subjectively easier to understand the wiring of URIs to XQuery functions as the URIs are declarative. The port did not attempt to re-structure the original code layout which was written around eXist-db's URL Rewriting Framework. Arguably if the code was further restructured instead around the concept of Resource Functions, then the lines of code could be reduced and the code-base may become more modular.

5.3 Further Work

Certainly the majority of the code written for the eXist-db implementation could trivially be made useable for other implementations in Java, by removing any direct eXist-db dependencies and replacing them with implementation agnostic interfaces. However, other implementations would need to support something akin to Triggers or consider modifying their XQuery 3.0 parser with rules for XQuery RESTful Annotations.

In addition, as the majority of XQuery implementations appear to be written in either Java or C++, should the majority of the code be made implementation independent, it would be interesting to perform a port to C++.

6. Conclusion

In this work we tackled the problem of using XQuery as a Server Side processing language for the Web whilst maintaining vendor independence and portability of XQuery code. The presented approach, driven by the review of current vendors' products, was to propose an implementation agnostic set of RESTful Annotations for XQuery 3.0. The goals of the approach developed in this paper were Interoperability, Simplicity and Technical Improvement:

- Interoperability has been addressed by proposing a set of Annotations which do not require any particular product or programming language for implementation. Whilst these Annotations permit the development of RESTful services, they do not constrain implementors choice of platform or technology.
- Simplicity has been addressed by developing an approach that is not disruptive. XQuery developers can continue to write XQuery in the same method that they always have. Should they require to provide RESTful Web Services, they can simply add the RESTful annotations to their functions, enabling them as Resource Functions.
- Technical Improvement has been achieved by understanding the strengths and limitations of the approaches taken in current vendors' products and building upon these. The strength of using URI Rewriting, in a declarative mechanism from XQuery frameworks has been enhanced by removing the framework aspect and ensuring that the declarations of intention appear alongside the code target function.

Additionally, we justify how our approach meets the four interface constraints of REST:

1) Identification of Resources
URIs are used to identify resources (or representations) and are mapped (or partially

mapped) to XQuery functions for delivery. This is achieved through the use of Path Annotations and URI Templates as described in §4.3.1. XQuery Functions themselves enable further mechanisms for the addressing of XML documents or other resources through fn:doc() and fn:collection(), and/or XML nodes through XPath.

2) Manipulation of Resources Through Representations
Resource Functions (RESTful Annotated XQuery Functions) permit the generation of representations of resources through XQuery processing, and serialization of the the representation through repurposing the XDM Serialization rules as described in §4.5. In addition content negotiation may be enabled by the mechanisms described in §4.3.3 to enable a server to deliver different serializations (e.g. XML, XHTML, Text, JSON etc) of a resource depending on the constraints of a request.
Manipulation of resources in typical RESTful HTTP service requests is by the use of both URI addressing (1) and HTTP methods. Many XQuery implementations have vendor extensions for manipulating document stores and the XQuery Update specification provides for document modifications. The mechanism for mapping HTTP methods to XQuery functions, which may utilise vendor extensions and/or XQuery Update is described in §4.3.2.

3) Self-Descriptive Messages
The use of HTTP for RESTful services with support for all relevant HTTP methods provides for the constraint of Self-Descriptive messages. If the message content itself is XML or (X)HTML the message content in itself is also hopefully self-descriptive. Indeed, our approach attempts to be lightweight and not constrain developers and, as such, they have complete control over the message content.

4) Hypermedia as the Engine of Application State

   The result of a Resource Function should permit a developer to provide as much hyper-linking of the application state as they desire. Whilst attempts for specific systems have described mechanisms such as XSLT for embedding hyper-links in resultant content[20], the approach developed in this paper is rather to provide the generic constructs to enable this, without prescribing an approach to developers.

6.1 Limitations

Currently the approach developed by this paper does not provide support for either HTTP Matrix Parameters or HTTP Multipart requests or responses.

In addition, there is currently no mechanism for the default handling of URI's that do not meet a declared URI Path, for example, customised HTTP 404 Page Not Found responses. However, this could perhaps be considered out of scope for such a project.

6.2 Future Work

Future work would include technical support for both HTTP Matrix Parameters and HTTP Multipart requests and responses. Certainly it is envisaged that HTTP Multipart responses from Resource Functions could be achieved by returning a sequence of functions as the result, where each function is responsible for generating both the REST Response Headers and content for each part of a multipart response.

We believe that our declarative approach based on Annotations makes maintainability of RESTful web applications easier. However, in large applications there could still be some difficulty involved in developers understanding the bindings between Resource Functions and Web Services. As such, additional functions could be added to REST Function module in §4.6, to enable the lookup of functions based on Request parameters.

Whilst we have called our approach 'standardised', it is surely more independent that standardised.

Rather we hope that it will form the basis of a publicly available and agreed standard. It is, of course, recognised that further work would be required to create a thorough technical standard based on this work, and communities such as EXQuery or the W3C Community Groups might be an appropriate vehicle for such work.

The Annotations that we have developed, once standardised could also be applied to other XML processing languages such as XSLT and XProc. Whilst XSLT and XProc do not directly have the concept of Annotations, they certainly support mechanisms that would yield a similar implementation and result.

Bibliography

[1] "XQuery 1.0: An XML Query Language (Second Edition)." [Online]. Available: http://www.w3.org/TR/xquery/. [Accessed: 19-Nov-2011].
[2] "XQuery Update Facility 1.0." [Online]. Available: http://www.w3.org/TR/xquery-update-10/. [Accessed: 19-Nov-2011].
[3] "XQuery and XPath Full Text 1.0." [Online]. Available: http://www.w3.org/TR/xpath-full-text-10/. [Accessed: 19-Nov-2011].
[4] "SourceForge.net Repository - [exist] Contents of /trunk/eXist-1.0/src/org/exist/http/HttpServerConnection.java." [Online]. Available: http://exist.svn.sourceforge.net/viewvc/exist/trunk/eXist-1.0/src/org/exist/http/HttpServerConnection.java?limit_changes=0&revision=133&view=markup&pathrev=133. [Accessed: 20-Nov-2011].
[5] "SourceForge.net Repository - [exist] Contents of /trunk/eXist-1.0/src/org/exist/xpath/functions/request/RequestParameter.java." [Online]. Available: http://exist.svn.sourceforge.net/viewvc/exist/trunk/eXist-1.0/src/org/exist/xpath/functions/request/RequestParameter.java?revision=158&view=markup&pathrev=158. [Accessed: 20-Nov-2011].
[6] eXist-db, "Developer's Guide - Calling Stored XQueries," *eXist-db Developer's Guide*. [Online]. Available: http://www.exist-db.org/devguide_rest.html#d1915e781. [Accessed: 14-Jan-2012].
[7] 28msec, "RESTful Conventions," *28msec*. [Online]. Available: http://www.28msec.com/documentation/sausalitobasics-restfulservices. [Accessed: 14-Jan-2012].
[8] BaseX, "REST - BaseX Documentation," *BaseX Wiki*. [Online]. Available: http://docs.basex.org/wiki/REST. [Accessed: 14-Jan-2012].
[9] MarkLogic, "Application Programming in XQuery and XSLT - HTTP App Server Functions," *MarkLogic Server Documentation*. [Online]. Available: http://docs.marklogic.com/5.0doc/docapp.xqy#display.xqy?fname=http://pubs/5.0doc/xml/xquery/programming.xml%2353595. [Accessed: 14-Jan-2012].
[10] M. Probst and EMC, "EMC Community Network - ECN: Implementing a RESTful API for xDB using Jersey/JAX-RS," *EMC Developer Network*. [Online]. Available: https://community.emc.com/docs/DOC-4276. [Accessed: 14-Jan-2012].
[11] "XQuery Scripting Extension 1.0." [Online]. Available: http://www.w3.org/TR/xquery-sx-10/. [Accessed: 19-Nov-2011].
[12] "EXPath - Standards for Portable XPath Extensions." [Online]. Available: http://www.expath.org/. [Accessed: 19-Nov-2011].
[13] "EXQuery - Standards for Portable XQuery Applications." [Online]. Available: http://www.exquery.org/. [Accessed: 19-Nov-2011].
[14] "EXPath - HTTP Client." [Online]. Available: http://www.expath.org/modules/http-client/. [Accessed: 14-Jan-2012].

[15] “XQuery 3.0: An XML Query Language.” [Online]. Available: http://www.w3.org/TR/xquery-30/. [Accessed: 19-Nov-2011].

[16] “XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition).” [Online]. Available: http://www.w3.org/TR/xpath-datamodel/. [Accessed: 14-Jan-2012].

[17] S. Kepser, “A Proof of the Turing-completeness of XSLT and XQuery,” May 2002.

[18] M. Kaufmann and D. Kossmann, “Developing an Enterprise Web Application in XQuery,” *Web Engineering*, vol. 5648/2009, pp. 465–468, 2009.

[19] R. Fielding, “Architectural Styles and the Design of Network-based Software Architectures,” University of California, Irvine, 2000.

[20] C. Davis, “Programming Application Logic for RESTful Services Using XML Technologies,” in *Proceedings of Balisage: The Markup Conference 2011*, Montreal, Canada, 2011, vol. 7.

[21] L. Quin, “XML Query Implementations,” 15-Jan-2012. [Online]. Available: http://www.w3.org/XML/Query/#implementations. [Accessed: 15-Jan-2012].

[22] F. Georges, “EXPath - Servlet,” *fgeorges.org Wiki*, 15-Jan-2012. [Online]. Available: http://fgeorges.org/wiki/EXPath#Servlet_definition. [Accessed: 15-Jan-2012].

[23] Retter, Adam, “SOAPServer - SourceForge.net Repository - [exist] Revision 3626.” [Online]. Available: http://exist.svn.sourceforge.net/viewvc/exist?view=revision&revision=3626. [Accessed: 15-Jan-2012].

[24] “Corona - Installation,” *Corona - GitHub*, 15-Jan-2012. [Online]. Available: https://github.com/marklogic/Corona/wiki/Installation. [Accessed: 15-Jan-2012].

[25] “EMC xDB 10.1.0 manual - Web client.” [Online]. Available: http://developer.emc.com/docs/documentum/xdb/manual/index.html#doc:topic/web_client.html . [Accessed: 18-Jan-2012].

[26] C. Davis and EMC, “EMC Community Network - ECN: Building Domain Specific RESTful Services over xDB with the EMC XML REST Framework (Version 1).” [Online]. Available: https://community.emc.com/docs/DOC-6434. [Accessed: 18-Jan-2012].

[27] C. Davis and EMC, “EMC Community Network - ECN: Adding Hyperlink Insertion and XML Transformations to the XMLREST Framework.” [Online]. Available: https://community.emc.com/docs/DOC-6485. [Accessed: 18-Jan-2012].

[28] F. Georges, “Packaging System,” *EXPath*, 11-Nov-2010. [Online]. Available: http://expath.org/spec/pkg. [Accessed: 15-Jan-2012].

[29] F. Georges, “CXAN: a case-study for Servlex, an XML web framework,” in *XML Prague 2011 Conference Proceedings*, Lesser Town Campus Prague, Czech Republic, 2011, vol. 2011-519.

[30] M. Hadley and P. Sandoz, Eds., “JAX-RS: Java API for RESTful Web Services Version 1.0.” Sun Microsystems Inc., 08-Sep-2008.

[31] “XSLT and XQuery Serialization 3.0.” [Online]. Available: http://www.w3.org/TR/xslt-xquery-serialization-30/. [Accessed: 22-Jan-2012].

[32] “XML Summer School.” [Online]. Available: http://xmlsummerschool.com/. [Accessed: 23-Jan-2012].

[33] “See What I Think | Free software downloads at SourceForge.net.” [Online]. Available: http://sourceforge.net/projects/seewhatithink/. [Accessed: 23-Jan-2012].