Using XQuery as a template language in Web applications.

Table of Contents

Abstract

XML Query shows an interesting capacity to be used as a "Server Pages" technology, a template language for generating Web pages. Several implementations of this functionality have already been proposed (GNU/Qexo[1], eXist[2]). This paper introduces *XQuery Server Pages* (*), a fairly complete implementation which is part of Qizx/open. We assume that the reader has a some knowledge of both XML Query and writing web applications with servlets or JSP.

A demonstration .war (Web Application Archive) with a few simple examples can be downloaded from Qizx/open's site. See the download page[3].

(*) this obvious reference to JSP or ASP should be regarded as humorous, as XQSP has no pretension to compete with these widely used technologies. However an effort to standardize the use of XQuery in Web applications could become of interest in a not-so-distant future.

1. Introduction

Many template languages are available today for generating Web pages: JSP, ASP, PHP... Most of these formalisms rely on some special markup that allows to distinguish between instructions or expressions in a particular programming language (to be evaluated) and raw HTML or XML (to be simply sent to the HTTP pipe): for example ASP and JSP use `<% ... %>` tags, PHP uses processing-instructions `<?php ... ?>`.

The advantages of this approach are: simplicity; efficiency (templates are in general preprocessed and compiled); compatibility with HTML editors.

The drawbacks are also well-known: see for example the article by Jason Hunter "the problems with JSP[4]": unstructured mixing of HTML markup and executable instructions produce cluttered source code, incline programmers to put too much code in templates, hence a poor separation between logic and presentation. In addition a language like Java is sometimes not very convenient in some common tasks like iterating on data structures.

Several techniques have been used to palliate such problems: components (COM, Java Beans) encapsulate business logic complexities; Tag Libraries delegate control logic to specialized tags à la XSLT; Model-View-Controller approach ("JSP Model 2") completely separate logic and presentation;

[1] http://www.gnu.org/software/qexo/XQ-Gen-XML.html
[2] http://exist.sourceforge.net/devguide.html
[3] http:///www.xfra.net/qizxopen/download.html
[4] http://www.servlets.com/soapbox/problems-jsp.html

2. XML Query

XML Query is not merely a query language to retrieve XML from databases; It is in itself a quite powerful processing language which can very well be used as a "Server Pages" technology:

- A XML Query expression evaluating as a document or an element is actually a template, that always generates a well-formed XML structure. XQuery is basically capable of generating XML documents: it has powerful instructions to construct, access and combine XML nodes.
- XML Query mixes executable instructions and "tags" in a clean way: instructions and tags can be nested at any level with a consistent syntax. In fact, "tags" are not different than instructions. They are integral part of the XML Query language (they are called "element constructors").
- Hence using XQuery as a template language can be very interesting in applications
  - which directly manipulate XML fragments: this is typically the case when XQuery is used as the query and processing language of a native XML database / search engine.

    By contrast, manipulating XML fragments for example with Java and DOM is fairly uneasy.
  - Where the generation task is non-trivial:
    - Unlimited nesting of instructions and "tags" allow sophisticated yet clean coding.
    - Functions (returning elements) can be used as building-blocks or templates, at several levels.

    By contrast, with classical template languages it is hardly possible to go beyond simple and fixed template structures.
- Limitations: XQuery in web applications cannot be as efficient as JSP, however this will rarely be an issue.

A sample: when setup in the XQSP environment provided by Qizx, the following snippet is able to echo the headers of an HTTP request:

Figure 1. echo headers:

```
<html><body>
<p> Date: { current-date() }, time: { current-time() }</p>
<h4>Your request contains these headers:</h4>
<ul>{
    for $h in request:getHeaderNames() return
        <li>{ $h } = { request:getHeader(string($h)) }</li>
}
</ul>
</body></html>
```

Expressions embedded in element constructors are marked in blue. A nested element constructor <li> itself containing expressions can be noticed inside the FLWOR loop. The template contains calls to standard functions (current-date and current-time) and to Java extension functions (request:getHeaderNames and request:getHeader).

3. Qizx Server Environment

A Web Application Archive (WAR) containing the XQuery Server implementation and a few simple examples is available on Qizx/open's download page[5]. This distribution can also serve as a basis for developing new applications.

[5] download.html

Installation: deploying this example should be very simple. Most J2EE-compliant servlet containers simply require to drop the war archive in a directory named `webapps`, then it is deployed automatically. Refer to the documentation of your servlet container.

3.1. Features

The *XQuery Server Pages* implemented by Qizx provide the following features:

- A generic servlet recognizes requests with `.xqsp` extension, is able to load and cache corresponding queries and to run them with the XQuery engine. The result is serialized and sent back as HTTP response.
- Access to the entire Java Servlet API through the Java extension mechanism available in Qizx.
- Convenience functions to ease basic tasks (access HTTP request and response, manipulate attributes and beans in page/request/session/application, forward request to other pages etc...)
- Serialization options can be specified through pragmas.
- Optional XSLT post-processing of the query output: provides additional means for separating processing and presentation.

3.2. Generic Servlet

In the example Web Application, this servlet (net.xfra.qizxopen.server.XQServlet) is configured to invoke XQuery Pages on HTTP requests which have a path ending with ".xqsp". This is achieved through a mapping in application descriptor web.xml:

```
  <servlet-mapping>
    <servlet-name>XQServlet</servlet-name>
    <url-pattern>*.xqsp</url-pattern>
  </servlet-mapping>
```

It is generally possible to define a server-wide mapping from the .xqsp extension to the generic XQServlet: this is however a server-dependent issue. Refer to the documentation of your preferred Servlet Container.

The generic servlet compiles XQuery Pages as needed and caches them. If a page resource was modified, it is automatically reloaded. Similarly, the servlet can load and cache XSLT templates, optionally used as a post processing of XQuery output. The cache sizes can be defined in configuration file web.xml.

3.3. API

The HttpServlet API is accessible through the Java extension mechanism. To make programming in XQuery easier, three namespace prefixes are predefined:

`xqsp:`	gives access to a few convenience functions (described below), for example `xqsp:headerNames()`, `xqsp:forward(path)`, `xqsp:usebean(...)`.
`request:`	gives access to interface `javax.servlet.http.HttpServletRequest`
`response:`	gives access to interface `javax.servlet.http.HttpServletResponse`

It means that all methods of these interfaces HttpServletRequest and HttpServletResponse are accessible in XQuery. For example `request:get-parameter(name)`, or equivalently `request:getParameter(name)` maps the Java method `HttpServletRequest.getParameter(String name)`.

Notice that it is not necessary to pass the request object itself (this is done implicitly). However the request and response objects can be accessed through variables `$xqsp:request` and $xqsp:response.

An example using several methods of interfaces HttpServletRequest and HttpSession. Notice that method names are written in the two possible styles: normal (aka camelCase) or in lowercase with dashes:

Figure 2. A more comprehensive excerpt of echo.xqsp

```
declare namespace session = "java:javax.servlet.http.HttpSession"

<html>
<body>
<h2>Echo of your request:</h2>
<p>Received on { request:get-server-name() } from {
    request:get-remote-host(), " address: ", request:get-remote-addr()
}</p>
<ul>
<li><b>method: </b>{ request:getMethod() }</li>
<li><b>protocol: </b>{ request:getProtocol() }, { request:getScheme() }</li>
<li><b>request-uri: </b>{ request:getRequestURI() }</li>
<li><b>locale: </b>{ request:getLocale() }</li>
<li><b>content-length: </b>{ request:getContentLength() }</li>
<li><b>session: </b>{
  let $s := request:getSession() return
  (  "id=", session:getId($s),
     ", last access=", session:getLastAccessedTime($s),
     ", timeout=", session:getMaxInactiveInterval($s),
     session:setMaxInactiveInterval($s, xs:int(10) ) )
 }</li>
</ul>
```

Convenience functions (prototypes are given in XQuery style, prefix is always **xqsp:**):

```
function xqsp:getResourceAsString($name as xs:string) as xs:string?
```

Loads a text file as a string: the file must be a resource of the current web application context.

```
function xqsp:forward($path as xs:string) as xs:boolean
```

Forwards the request to another page on the server. The path must begin with the "/" and be relative to the application context.

```
function xqsp:parameterNames() as xs:string*
```

Returns a sequence of HTTP parameter names. This is simply a convenience wrapper: request:get-ParameterNames() is almost equivalent, but this function's type is xs:string*, which makes easier to use.

```
function xqsp:headerNames() as xs:string*
function xqsp:initParameterNames() as xs:string*
```

similar wrappers for header names and init parameter names.

```
function xqsp:get-attribute($name as xs:string) as item()?
function xqsp:set-attribute($name as xs:string, value as item())
function xqsp:remove-attribute($name as xs:string)
```

Manage attributes on the current page. Arbitrary objects can be associated as named attributes of the page.

```
function xqsp:use-bean( $name as xs:string, $classname as xs:string,
                        $scope as xs:string) as item()?
```

This is similar to the use-bean functionality in JSP: an object can be instantiated from a Java class (if necessary) and associated as an attribute in a particular *scope*: the scope can be `"page"`, `"request"` (attributes survive to a *forward*), `"session"` (attributes are valid along the same user session), `"application"` (attributes are shared by all users of the same web application).

Argument $classname must be the fully qualified name of a loadable class. If it does not exist yet, it is instantiated like a Java Bean.

```
function xqsp:get-attribute($name as xs:string, $scope as xs:string)
         as item()?
```

Retrieves an attribute i.e a bean) from a scope. Returns the empty sequence if not found (no instantiation performed).

3.4. Serialization and XSLT Transforms

Serialization

Serialization options can be specified inside pages by a pragma put anywhere in the page: the name of the pragma must be qizx:serialization or x:serialize. The body of the pragma contains name=value pairs for serialization options, with the same semantics as the x:serialize function.

```
(::pragma x:serialize
   method=XHTML media-type=text indent=no encoding=iso8859-1 ::)
```

It is not currently possible to specify such options dynamically.

Pipelining with XSLT transformations

The result of a XQuery Page evaluation can be passed to a XSLT1 stylesheet. This allows a kind of separation between presentation and logic. It can also be more convenient for formatting data extracted from databases.

This is also achieved by using a pragma. The name of the pragma must be x:transform.

A particular option is **stylesheet**: it specifies the path of a XSLT stylesheet resource. Other options are passed as parameters of the stylesheet.

```
(::pragma x:transform
   stylesheet=shakespeare.xsl param1=value1 param2=value2 ::)
```

3.5. Security

The Java binding feature, which allows to call any Java method from inside a XQuery script, can represent a serious security hole in a server environment, at least if clients are allowed to execute arbitrary expressions. If they are not, no real issue exists here.

For this reason, the Java binding feature has a security mechanism, by which it is possible to specify selectively which Java classes may be used. In the Server pages environment, it is brought in operation in the following way:

- By default, only Servlet-related classes are allowed: HttpServlet, HttpServletRequest, HttpServletResponse, HttpSession.
- To allow other classes, modify the web application descriptor web.xml: there is an initialization parameter called `allowed-classes` which specifies a list of fully-qualified class names, for example:

```
    <init-param>
        <param-name>allowed-classes</param-name>
        <description>Classes allowed for Java extension
        (comma or space-separated list of class names.) </description>
        <param-value> java.util.Vector java.util.Calendar </param-value>
    </init-param>
```