# XQuery Server Pages

using XQuery in Web Applications

XQuest "Milestone 1" release



## Table of Contents



## 1. Overview

This chapter describes a mechanism that makes it possible to write web applications with XQuery as a page template language. This is somewhat similar to the JSP (Java Server Pages)

XML Query shows an interesting capacity to be used as a "Server Pages" technology, a template language for generating Web pages:

- 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 an integral part of the XML Query language (they are called "element constructors").

Hence using XQuery as a template language can be very interesting in web 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.
- 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.

**Example:** the following figure shows a very simple example of a "XQuery Server Page". This snippet can be executed in a Java Servlet container, within the environment provided by XQuest. It displays the current date and time and the HTTP headers sent by the *user-agent*, i.e. the user's browser:

**Figure 1. echo.xqsp**

```
<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>
```

In this example, function calls embedded in element constructors are marked in blue. There are calls to standard functions (current-date and current-time) and calls to Java extension functions provided by the XQuest environment (request:getHeaderNames and request:getHeader).

# 2. Environment

The *XQuery Server Pages* implemented by XQuest 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 a HTTP response.
- Access to the entire Java Servlet API through the Java extension mechanism available in XQuest.
- 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 evaluated query: provides another form of separation between processing and presentation.

In the example Web Application, this servlet (net.axyana.xquest.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. Serialized and XSLT Output

The well-formed tree generated by a XQuery page can be output in two different ways: direct serialization into text, or intermediate transformation with XSLT (in turn serialized by the XSLT processor).

**Serialization**

Serialization options can be specified inside pages by a pragma put anywhere in the page: the name of the pragma must be 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:serialization
          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 ::)
```

# 4. API

All 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:use-bean(...)`. |
| `request:` | gives access to the interface `javax.servlet.http.HttpServletRequest` |
| `response:` | gives access to the 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.

**Parameter $name :** path of a resource of the current web application context. Must begin with a slash.

```
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:getParameterNames() 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).

# 5. Security

The Java binding feature, which allows calling of 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 into 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>
   <param-value> java.util.Vector java.util.Calendar </param-value>
   <description>Classes allowed for Java extension
       (comma or space-separated list of class names.) </description>
</init-param>
```