
A practical introduction to XPath

Collaboratively Defining Open Standards for Portable XPath Extensions

Florent Georges

Copyright © 2009 Florent Georges. Used by permission.

Abstract

For some time now the demand for standardized extensions within the core XML technologies, especially XSLT and XQuery, has been increasing. A closer look at the EXSLT and EXQuery projects shows that their goals align to address the issue of extensions, but also that they overlap, predominantly because of their common ancestor: XPath. It is thus reasonable to assume that both projects should collaborate together on common areas influenced by the underlying XPath specification. In addition, by jointly working at the lower level, any XPath based language or processor could also benefit from this work, like XProc, XForms or plain XPath engines.

Table of Contents

Introduction	1
The project	2
Two example modules	3
SOAP Web service client	5
Compound Document Template pattern	7
Google Contacts to ODF	9
Packaging	10
Conclusion	12

Introduction

XPath is a textual language to query XML content. Besides a very convenient way to navigate within an XML document through a path system, it provides the ability to make various kinds of computation, for example string manipulations or date and time processing. Such features are provided as functions, defined in a separate recommendation detailing a standard library. It is also possible to use additional defined functions; XSLT and XQuery allow one to define new functions directly in the stylesheet or query module. An XPath processor also usually provides implementation specific libraries of additional functions, as well as a way to augment the set of available functions, these are typically written using an API in the same language as the processor.

This leads to the concept of an extension function. Such a function is not defined by XPath, but inspired by XSLT this is defined as any function available in an expression although not defined in any recommendation. There are two major kinds of extension functions: those provided by the processor itself, and those written by the user. A processor can provide extensions to deal with database management for instance, and the user can implement any processing he could not write in XPath directly (for instance using a complex mathematical library available in Java.) Examples of possible useful extensions could provide support for performing HTTP requests, using WebDAV, reading and writing ZIP files (like EPUB eBook, Open XML and OpenDocument files,) parsing and serializing XML and HTML documents, executing XSLT transforms and XQuery queries, etc.

Extension functions are very powerful ways to extend XPath functionality. Many extensions have been written over the years and by many different people. Reflecting on this, it appears almost impossible to create and then share an extension function among several processors and to maintain interoperability with them all over time. Extension functions provided by processors themselves are not compatible, so that an expression that uses them is not portable between processors. It is that situation which has led to the creation of a project that would define libraries of extension functions unrelated to any single processor, allowing existing processors to choose to implement them natively, or allowing a user to install an existing implementation as an external package in their processor. In this way it becomes

possible to use extension functions already defined and implemented, and if not more importantly, to write expressions that become portable across every processor that supports the extension or that an implementation was pre-provided for.

This project is EXPath [<http://www.expath.org/>]!

The project

EXPath was launched four months ago in April 2009. It is divided into several modules, quite independent from each other. The module is the delivery unit of EXPath. It is a consistent set of functions providing support for a particular domain, and can be viewed as a specific library of functions. Besides the specification of the functions itself, implementations can also be provided for various processors. The first module that has been defined is the HTTP Client. It provides support to send HTTP requests and to handle the responses. This module defines one single function, how it represents a HTTP request and response as XML elements and how it can be used to perform requests. Implementations are provided as open source packages via the website for a number of processors - Saxon, eXist and MarkLogic.

If there is a need for new extension functions to address missing functionality, a new module may be proposed by anyone with the help of other people interested in supporting the same features. The first step in this process is to identify precise use cases for the desired features, and describe examples of use; this sets the scope of the module and sets us on the road to defining specifications. Of course the scope can evolve over time, as well as the use case examples, but this gives a formal basis for further discussion.

Once the use cases have been identified, function signatures and definitions of behaviour for the module must be proposed. That proposal is discussed among the community through the mailing list, and people interested in this module help to solve issues and improve its design. Extensions addressing the same problem might already exist, either as part of a known processor, or developed as independent individual projects; this is valuable material to learn from during this stage. In parallel with this definition, it is encouraged to develop at least one implementation to validate the feasibility of what is being specified. Before being officially endorsed as a module, at least two implementations should have been provided, to ensure that the specification is not too tightly linked to a particular processor.

Each module has its own maintainer who is responsible for editing the specification, leading the discussions and making design choices in according to those discussions. Similarly, every implementation of a module for a particular processor has its own maintainer. If someone needs the module to be implemented for another processor, they may volunteer for that work, and propose an implementation. A test suite for the module should be provided and carefully designed to ensure every implementation provides the same functionality. The vendor of a processor might also propose a native implementation of a module, in which case responsibility for the implementation and ensuring it meets the specification and tests lies with the vendor.

An interesting point about implementing a module is when a processor already provides a similar feature through processor-specific extension functions. It may then be sometimes possible to reuse the existing extensions through a lightweight wrapper exposing the EXPath interface to the user. For instance, the HTTP Client has been implemented for MarkLogic Server with an XQuery module using MarkLogic's own HTTP Client set of extension functions. This example also shows the limitations of this approach: the implementation is only partial, as the MarkLogic's extensions do not provide exactly all the possibilities required by the EXPath HTTP Client module. It is thus not always possible to use this technique to provide full conformance, but this is a convenient way to quickly provide a (possibly partial) implementation.

Needless to say, lots of work needs to be done, and everyone is welcome to help, at every level: writing specifications, writing documentation and tutorials, peer review, discussion, coding, testing or simply using the extensions and providing feedback.

But let's call it a day on the theory, and let's examine some real examples using two modules: the HTTP Client and ZIP modules.

Two example modules

Before going through the examples, here is a simple introduction to the modules. The HTTP Client module, without any surprise, provides the ability to send HTTP requests and handle the responses. The ZIP module provides ZIP files support, either to extract entries from them, add new entries, or update entries in existing ZIP files.

The HTTP Client provides one single function: `http:send-request` (actually it defines four different arities for this function, but this is mainly a convenient way to pass some values as separate parameters.) This function has been inspired by the corresponding XProc step, `p:http-request`. This function makes it possible to query REST services, Google services, Web services through SOAP, or simply to retrieve resources on the Web. The signature of this function (the one-parameter version) is:

```
http:send-request($request as element(http:request)) as item()+
```

Its parameter is an element representing the request. It is structured as in the following sample:

```
<http:request href="http://www.example.com/..." method="post">
  <http:header name="X-Header" value="some value"/>
  <http:header name="X-Other" value="other value"/>
  <http:body content-type="application/xml">
    <hello>World!</hello>
  </http:body>
</http:request>
```

The request will result in a HTTP POST, with Content-Type `application/xml`, sent to the specified URI and with a header explicitly set by the user (`X-Header: some value`.) The result is described with another element, `http:response`, that describes the response returned by the server:

```
<http:response status="200" message="OK">
  <http:header name="..." value="..." />
  ...
  <http:body content-type="application/xml" />
</http:response>
```

The structure of this element is not dissimilar to the request element, instead of the URI and HTTP method, it contains the status code and the message returned by the server. One significant difference as opposed to the request, lies in the way in which the response body is returned: the response body element (if any) is only a description of the body. The body content itself is returned as a subsequent item in the result sequence (or several items in case of a multipart response.) Let's analyse an example in XQuery:

```
http:send-request(
  <http:request href="http://www.balisage.net/" method="get"/>)
```

This simply sends a GET request to the web server of Balisage. The result of this function call is a sequence of two items: the `http:result` element and the body content as a document node (holding an XHTML document):

```
<http:response status="200" message="OK">
  <http:header name="Server" value="Apache/1.3.41 (Unix)" />
  ...
  <http:body content-type="text/html" />
</http:response>

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Balisage: The Markup Conference</title>
```

...

It may sound strange to have a sequence as the result of the function, but this is the only way to provide the response description alongside the payload expressed as a full document node. This design choice prevents mixing different layers in the same document. The body content is analysed based on the content-type returned by the server. If the content type is of an XML type it is parsed and returned as a document node, if it is an HTML type it is tidied up, parsed and returned as a document node, if it is a textual type it is returned as a string, in any other case it is returned as a `xs:base64Binary` item. In the case of a multipart response this rule is applied to each part which is then returned as subsequent items after the `http:response` element.

The ZIP module defines the following functions to read the structure of a ZIP file, to create a completely new ZIP file from scratch, and to create a new ZIP file, based on an existing file by changing only some entries:

```
zip:entries($href) as element(zip:file)
zip:zip-file($zip as element(zip:file)) as empty-sequence()
zip:update-entries($zip, $output) as empty-sequence()
```

The first function takes a ZIP file's URI as parameter, and returns a description of its structure (its entries) as a `zip:file` element. This element looks like the following:

```
<zip:file href="some.zip">
  <zip:entry name="file.xml"/>
  <zip:entry name="index.html"/>
  <zip:dir name="dir">
    <zip:entry name="file.txt"/>
  </zip:dir>
</zip:file>
```

Only the structure is returned, not the whole content of each file entry. The function `zip:zip-file()` takes a similar element as parameter, and create a new ZIP file based on that content. The element is similar, but in addition it contains the content of each file entry, so that the function has all the needed information to actually create the whole file. For each `zip:entry`, the content of the element can be an XML document, a string, or binary encoded as a base 64 string (an attribute tells if the content has to be serialized as XML, HTML, text or binary.) An existing file can also be copied verbatim to the entry, by giving its URI instead of the actual content. An example of a `zip:file` element to pass to this function is:

```
<zip:file href="some.zip">
  <zip:entry name="file.xml" output="xml">
    <hello>World!</hello>
  </zip:entry>
  <zip:entry name="index.html" href="/some/file.html"/>
  <zip:dir name="dir">
    <zip:entry name="file.txt" output="text">
      Hello, world!
    </zip:entry>
  </zip:dir>
</zip:file>
```

The third function, `zip:update-entries()` looks a lot like `zip:zip-file()`, but it uses an existing ZIP file to create a new one, replacing the entries in the `zip:file` element in parameter. It then becomes possible to use a pattern file, replacing only a few entries, with content computed in this expression. To be complete, the module also provides 4 functions to read one specific entry from an existing ZIP file, for instance depending on the result of `zip:entries()`. They return either a document node, a string or a `xs:base64Binary` item, following the same rules as `http:send-request()`:

```
zip:xml-entry($href, $entry) as document-node()
zip:html-entry($href, $entry) as document-node()
```

```
zip:text-entry($href, $entry) as xs:string
zip:binary-entry($href, $entry) as xs:base64Binary
```

SOAP Web service client

EXPath works at the XPath level. But XPath is never used standalone, it is always used from within a host language. Among the various languages providing support for XPath, XSLT and XQuery have a place of choice, by their close integration with XPath. We will thus use XQuery and XSLT to demonstrate complete examples of the functions introduced earlier. XQuery's ability to create elements directly within expressions allows for very concise examples that are easier to understand. However, where transforming XML trees is fully part of the example, XSLT is used instead.

The first sample consumes a SOAP Web service. The Web service takes as input an element with a country code and a city name. It returns an element with a corresponding place name and the local short term weather forecast. The format of the request and response elements is as follows:

```
<tns:weather-by-city-request>
  <tns:city>Montreal</tns:city>
  <tns:country>CA</tns:country>
</tns:weather-by-city-request>

<tns:weather-by-city-response>
  <tns:place>Montreal, CA</tns:place>
  <tns:detail>
    <tns:day>2009-08-13</tns:day>
    <tns:min-temp>16</tns:min-temp>
    <tns:max-temp>26</tns:max-temp>
    <tns:desc>Ideal temperature for a conference.</tns:desc>
  </tns:detail>
  <tns:detail>
    ...
  </tns:detail>
</tns:weather-by-city-response>
```

Thus, the XQuery module has to send a SOAP request to the Web service HTTP endpoint, check any error conditions and then if everything went well, process the result content. These steps fit naturally into three different functions defined within the XQuery module. After a few declarations (importing the HTTP Client module, declaring namespaces and variables,) the three functions are defined and then composed together:

```
xquery version "1.0";

import module namespace http = "http://www.expath.org/mod/http-client";

declare namespace soap = "http://schemas.xmlsoap.org/soap/envelope/";
declare namespace tns = "http://www.webservicex.net";

declare variable $endpoint as xs:string
  := "http://www.webservicex.net/WeatherForecast.asmx";
declare variable $soap-action as xs:string
  := "http://www.webservicex.net/GetWeatherByPlaceName";

(: Send the message to the Web service.
: )
declare function local:send-message()
  as item()+
{
  http:send-request(
    <http:request method="post" href="{ $endpoint }">
      <http:header name="SOAPAction" value="{ $soap-action }"/>

```

```
<http:body content-type="text/xml">
  <soap:Envelope>
    <soap:Header/>
    <soap:Body>
      <tns:weather-by-city-request>
        <tns:city>Montreal</tns:city>
        <tns:country>CA</tns:country>
      </tns:weather-by-city-request>
    </soap:Body>
  </soap:Envelope>
</http:body>
</http:request>
)
};

(: Extract the SOAP payload from the HTTP response.
: Perform some sanity checks.
: )
declare function local:extract-payload($res as item()+
  as element(tns:weather-by-city-response)
{
  let $status := xs:integer($res[1]/@status)
  let $weather := $res[2]/soap:Envelope/soap:Body/*
  return
    if ( $status ne 200 ) then
      error(xs:QName('ERRSOAP001'),
        concat('HTTP error: ', $status, '-', $res[1]/@message))
    else if ( empty($weather) ) then
      error(xs:QName('ERRSOAP002'), "SOAP payload is empty?")
    else
      $weather
};

(: Format the Web service response to a textual list.
: )
declare function local:format-result(
  $weather as element(tns:weather-by-city-response))
  as xs:string
{
  string-join((
    'Place: ', $weather/tns:place, '&#10;',
    for $d in $weather/tns:detail return $d/concat(
      ' - ', tns:day, ':&#09;', tns:min-temp, ' - ',
      tns:max-temp, ':&#09;', tns:desc, '&#10;'
    )
  ),
  '')
};

(: The main query, orchestrating the request, extracting and
: formatting the response.
: )
let $http-res := local:send-message()
let $payload := local:extract-payload($http-res)
return
  local:format-result($payload)
```

If everything goes well, the result should look like the following:

```
Place: Montreal, CA
- 2009-08-13: 16 - 26: Ideal temperature for a conference
- 2009-08-14: 24 - 32: Enjoy holidays
...
```

This simple example shows a client call to a SOAP Web service. Given that such Web services are usually described by a WSDL service description, it is actually possible to automatically generate something like the previous example, but once for all operations described in the WSDL. This has been implemented in the WSDL Compiler, an XSLT stylesheet that transforms a WSDL file into an XSLT stylesheet or an XQuery module which can then be used as a library module. As each WSDL operation becomes a function, the XPath processor actually checks at compile time that the namespace URI, function name and parameters are correct. Here is the same example but using the module generated by this WSDL compiler:

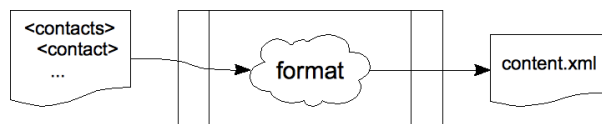
```
(: The module generated by the WSDL compiler.
: )
import module namespace tns = "http://www.webservicex.net" at "weather.xq";

(: The main query call the Web service like a function, passing
: directly its parameters and using its result.
: )
local:format-result(
  tns:weather-by-city(
    <tns:weather-by-city-request>
      <tns:city>Montreal</tns:city>
      <tns:country>CA</tns:country>
    </tns:weather-by-city-request>
  )
)
```

All the HTTP and SOAP details have been hidden, and calling the Web service looks exactly like using a library module. Technically speaking, this is the case: only that module uses the HTTP extension and has been generated from the WSDL file, so that it knows how to communicate with the Web service.

Compound Document Template pattern

This section introduces a new design pattern to generate compound documents based on the ZIP format, such as OpenDocument or Open XML, by using a template file. It is particularly well-suited to file generation based on transformations from data-oriented document to ODF for presentation purpose. Let us take a concrete example, transforming a simple contact list to an OpenDocument (ODF) text file (I use ODF as an example here, but the pattern could equally applied to other similar formats.) A typical transform of an input data-oriented document to an OpenDocument content.xml file can be represented by the following workflow (by which a contact list is transformed to ODF using XPath technologies, for instance an XSLT stylesheet):

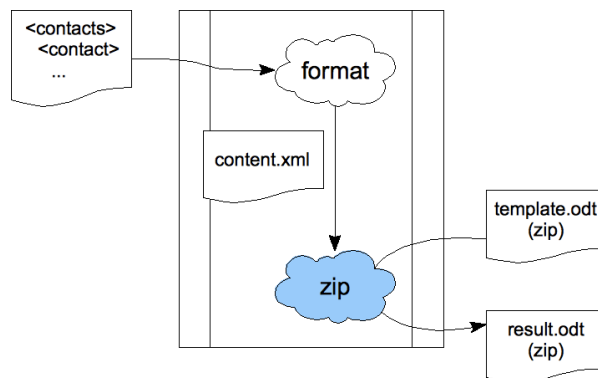


The `content.xml` file is just one file in a whole ODF document. Such a document is actually a ZIP file, containing several XML files (the structure of the ZIP file in a manifest, the several styles used in other parts, the content...) among other files (for instance pictures in an OpenDocument Text file.) For instance, a typical ODF text document could have a structure like the following (if you open it as a ZIP file):

```
mimetype
content.xml
styles.xml
```

```
meta.xml
settings.xml
Thumbnails/
META-INF/
  manifest.xml
Pictures/
  d9e69.map.gif
  d9e80.map.gif
  d9e82.photo.png
```

So the above transformation requires some post-processing to create the ZIP file that bundles all those files into a proper ODF document. In addition, some files depend on the way `content.xml` is generated (for instance the style names it uses) even if they could be seen as static files. ODF is a comprehensive and quite complex specification, and it could rapidly become complicated to deal with all of its aspects while generating a simple text document. This pattern simplifies this by allowing one to create a template document using an application like Open Office. In this specific example, this is as simple as creating a table, with several columns, where each line is a contact. By using Open Office, we can set up all the layout visually, and create one single contact in the table. The transform will then extract `content.xml` from the template file, copy it identically except for the fake contact line that will be used as a template for each contact in the input contact list. Then the result is used to create a new ZIP file, based on the template file, where all entries are copied except `content.xml`, which is replaced by the result of the transform. The general diagram for this pattern is:



There are several possible variants: the transform could read some entries from the template file, or not; it can generate several different entries or just the main content file... But the important point is to be able to create and maintain the overall structure and the layout details from within end-user applications for one ZIP-and-XML based format, and then to be able to use this template and just fill in the blanks with actual data. Here is how the last step (creating a new ZIP file based on the template file by updating some entries) can be implemented in XSLT:

```
<xsl:param name="content" as="element(office:document-content)"/>

<xsl:variable name="desc" as="element(zip:file)">
  <zip:file href="template.odt">
    <zip:entry name="content.xml" output="xml">
      <xsl:sequence select="$content"/>
    </zip:entry>
  </zip:file>
</xsl:variable>




<xsl:sequence select="zip:update-entries($desc, 'result.odt')"/>
```

This ability to manipulate ZIP files opens up several possibilities with ODF. Spreadsheets or text documents could for instance be used directly from within XSLT stylesheets or XQuery modules as a natural human front-end for data input. But let us examine a simpler transformation from a contact list to a text document.

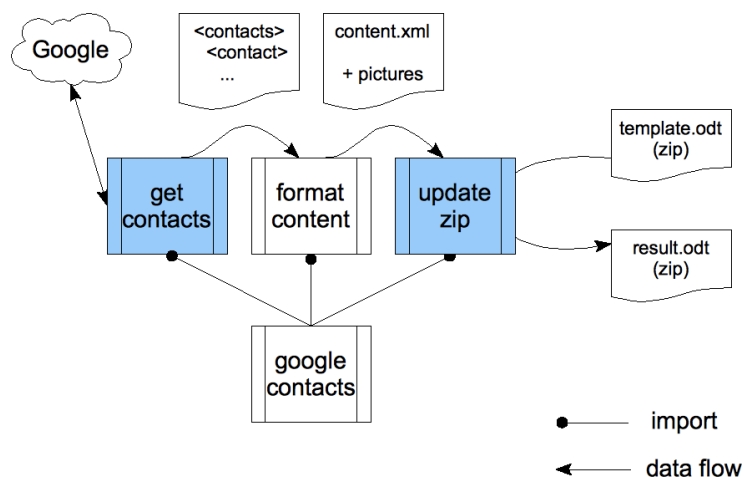
Google Contacts to ODF

Using the `http:send-request()` extension (introduced above) it is possible to access Google's Data REST Web services. It is even possible to write a library to encapsulate those API calls into our own reusable functions. I have written such a library for XSLT 2.0, supporting the underlying Data Service, as well as various dedicated services, such as contacts and calendar (unfortunately, it is not published yet, but feel free to contact me in private, or through the EXPath mailing list if you are interested in obtaining a draft version.) Let's take for instance the Google Contacts API [<http://code.google.com/apis/contacts/>], and use the contact information it provides to create a contact book in ODF Text format. The contact book is a simple table with three columns: a picture of the contact if any, then its textual information (name, email, address, etc.) and finally a last column with a thumbnail of a Google Map of its neighbourhood:

Contacts

Photo	Name	Map
	Michael Kay Saxonica mike@saxonica.com Reading, UK Group: XSL List	
	Florent Georges fgeorges.test@gmail.com rue de Savoie 73 1060 Brussels	
	Jirka Kosek jirka@kosek.cz Groups: XML Prague, XSL List	
	Jim Fuller FlameDigital Ltd. jim.fuller@xmlprague.cz	

By using the Compound Document Template pattern, it is possible to create this table directly in Open Office, by just creating one single row with a picture, a name in bold, an email address as a link, other textual info, and a map. Let's save the file in, for instance, `template.odt`. The transform contains three rather independent parts: retrieve info from Google, format them into a new `content.xml` document, then create a new `result.odt` file by updating `content.xml` and adding the picture files:



By using an intermediate document type for the contacts, we apply the *separation of concerns* principle to isolate the formatting itself, making it independent of the specific Google Contacts format. The *get contacts* and *update zip* modules use the same kind of code shown in previous sections to ask the Google servers on the one hand, and to read and update ZIP files on the other hand. The orchestration of those three modules is controlled by the main driver: *google contacts*.

Packaging

XPath and XSLT 1.0 recommendations are ten years old, and still there are very few libraries and applications distributed in XSLT. You can find looking around very valuable pieces of work written in XSLT like FunctX [<http://www.functx.com/>], XSLStyle [<http://www.cranesoftwrights.com/resources/xslstyle/>], Schematron [<http://www.schematron.com/>] or of course DocBook [<http://www.docbook.org/>], but the absence of a standardized way to install them tends to slow down their adoption. However, each time one has to go through the same process: figuring out the exposed URIs, creating a catalog mapping those URIs to the install location, checking whether there are dependencies to install, plug the catalog to the cataloging system of her processor or IDE (for each of them,) etc. But there is no homogeneity in the way each project packages its resources and it documents the install process.

It would be nice though to be able to download a single file, to give that file to a processor to install it automatically, and to rely only on the public URIs associated to XSLT stylesheets, XQuery modules and XProc pipelines. Or with any library of functions, including extension functions. Where other languages define packages, core XML technologies use URIs. But the concept is the same, and we should only care about those URIs when developing, instead of constantly keeping trace of physical files, and installing them again and again on each machine we have to work on. Everything else than those public URIs should be hidden deep in the internals of a simple packaging system supported by most processors, IDEs and server environments.

There is not already a formal proposal for such a system, but the idea is quite well-defined, and there is even an implementation of a prototype for Saxon, supporting standard XSLT stylesheets and XQuery modules, as well as extension functions written in Java. I have successfully packaged projects like the DocBook XSLT stylesheets, the DITA Open Toolkit XSLT stylesheets, the XSLStyle stylesheets, or the FunctX library (for XSLT as for XQuery.) The principle is very simple: a graphical application helps you to manage a central repository where it installs the libraries, as well as generated XML Catalogs, and a shell script wrapper or Java helper class configure correctly Saxon, so you can still use Saxon from the command line or from within a Java program. You can then use the public import URI of the installed stylesheet to import the components it provides. This is also possible with extension functions, even though Saxon does not provide any (simple) way to link a URI to an extension function. The trick is to provide a wrapper stylesheet that uses the Saxon-specific URI to access extension functions in Java, while providing a wrapper function defined in the correct namespace for each exported function from Java:

```
<xsl:stylesheet xmlns:http="http://www.expath.org/mod/http-client"
               xmlns:http-java="java:org.expath.saxon.HttpClient"
               ...>

  <xsl:function name="http:send-request" as="item()+">
    <xsl:param name="request" as="element(http:request)?" />
    <xsl:sequence select="http-java:send-request($request)" />
  </xsl:function>

  ...
```

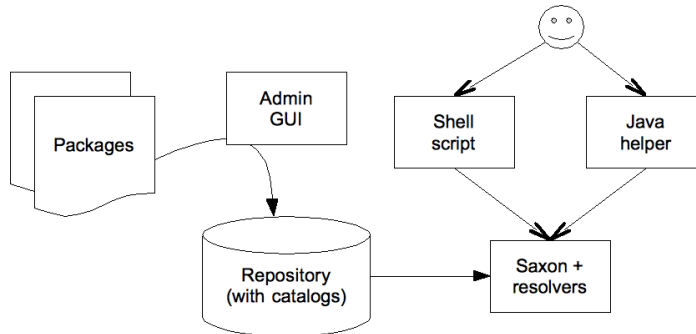
On the other hand, the stylesheet using the extension function does not rely on any detail of its implementation. It uses a public absolute URI to import the library and the namespace defined in the specification instead of the Java-bound namespace:

```
<xsl:stylesheet xmlns:http="http://www.expath.org/mod/http-client" ...>
```

```
<xsl:import href="http://www.expath.org/mod/http-client.xsl"/>

<xsl:template match="/">
  <xsl:sequence select="http:send-request(...)" />
  ...
</xsl:template>
```

Here is how the different pieces fit together:



The admin GUI is used to manage the repository (you can use it from the command line too.) It can install, remove and rename packages. Once installed, everything needed by a module resides in that repository (you can have several repositories, for different purposes or projects,) like stylesheets and queries, XML catalogs and JAR files for extension functions. The shell script is used to execute Saxon from the command line, while the Java helper class is used from within Java programs, for instance in a web application. Both do configure Saxon with the appropriate resolvers to locate properly the modules in the repository. For instance, you can add the following line to your project's Makefile in order to generate the documentation with XSLStyle:

```
> saxon -xsl:urn:isbn:978-1-894049:xslstyle:xslstyle-dita.xsl -s:my.xsl
```

This is an interesting example as XSLStyle relies on DocBook or DITA (or both.) Let us examine how I packaged XSLStyle. The original distribution includes a private copy of both DocBook and DITA. This is a bad practice in software engineering, but one has no other choice as long as there is no packaging system out there. The first step is thus to remove those external libraries, and package them separately. Then we have to define a public URI to allow other stylesheets to import the stylesheets provided by this package. This URI does not have to point to an actual resource; this is a logical identifier. Following Ken's advice, I chose `urn:isbn:978-1-894049:xslstyle:xslstyle-dita.xsl`, and a similar one for the DocBook version. The same way, because we removed the private copies of DocBook and DITA, we have to change the XSLStyle stylesheet accordingly, to use public URIs in their import instructions. As long as the package for DocBook or for DITA are installed, Saxon will find them through the packaging system, by their public URIs.

Furthermore, because Saxon is a standalone processor and can be invoked from the command line directly by humans, the shell script allows you to use symbolic names in addition to URIs. URIs are convenient from within program pieces like stylesheets and queries, but are not well-suited to human beings. Depending on how exactly you installed and configured the package, the above example can be rewritten as:

```
> saxon --xsl=xslstyle-dita -s:my.xsl
```

This prototype has been implemented only for Saxon so far, but the concept of a central repository is particularly well-suited for XML databases as well, like eXist or MarkLogic, as they already own a private space on the filesystem to save and organize their components and user data. And because the resolving mechanism is built on top of XML Catalogs, any serious XML tool or environment (like oXygen) can be configured to share the same repository. If this packaging system becomes widely used to deliver libraries and applications, we can even think about a central website listing all existing packages, allowing for automatic package management over the Internet. Like the APT system for

Debian, CTAN for TeX or CPAN for Perl. The later inspired in fact the concept and the name of CXAN for such a system, thanks to Jim Fuller and Mohamed Zergaoui, which stands for Comprehensive X* Archive Network (replace X* with your preferred core XML technology.)

Conclusion

EXPath has just been launched, right after XML Prague 2009, and yet there are very valuable modules. The most exiting of them being maybe the packaging system. Besides other modules defined as function libraries, this system is a good example of a module that is not a library, but rather a tool or a framework which various core XML technologies could benefit from. Other ideas include nested sequences, useful to define and manipulate complex data structures, first-class function items, to define higher-order functions, XML and HTML parsers, or filesystem access. Another module under investigation aims to define an abstract web container and the way a piece of X* code can plug itself into this container to be evaluated when it receives requests from clients, as well as the communication means between the container and the user code. Something similar to Java EE's servlets. For now, there is no abstract description of the services provided by an X* server environment, like an XML database. Given such a description of an abstract container, third-party frameworks could be written in an implementation-independent way, like web MVC frameworks and frameworks for XRX applications.

That gives me the opportunity to introduce three sibling extension projects: EXQuery, EXSLT 2.0 and EXProc. Without any surprise, their goal is to define extensions to respectively XQuery, XSLT 2.0 and XProc. Each of those projects is complementary with EXPath, which has been created in the first place to avoid having to specify several times the same features, in a non-compatible way.

There is a bunch of work to define new extensions, to write their specifications, to implement and test them, to document them. Everyone is welcome to help on the project (and I am sure on the other EX* projects.) Just use the extensions and give us feedback on the mailing list: <http://www.expath.org/>.

See you soon!