
CXAN: a case-study for Servlex, an XML web framework

Florent Georges, H2O Consulting <fgeorges@fgeorges.org>

Copyright © 2011 Florent Georges. Used by permission.

Abstract

This article describes the EXPath Webapp Module, a standard framework to write web applications entirely with XML technologies, namely XQuery, XSLT and XProc. It introduces one implementation of this module: Servlex. It uses the CXAN website, the Comprehensive XML Archive Network, as a case study.

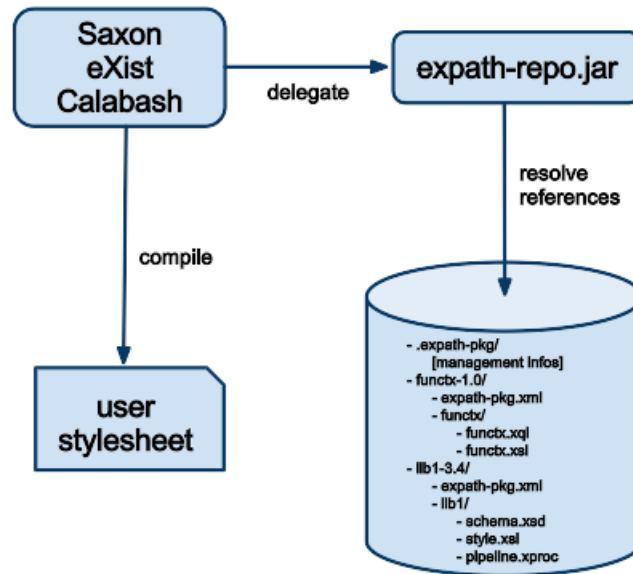
Table of Contents

Introduction	1
CXAN	2
Webapp and Servlex	4
The Webapp Module	4
Servlex	7
The CXAN website	10
The development project	14
Conclusion	14

Introduction

The EXPath project defines standard extensions for various XPath languages and tools. Most of them are extension function libraries, defining sets of extension functions you can call from within an XPath expression (e.g. in XSLT, XProc or XQuery), like the File Module, the Geo Module, the HTTP Client and the ZIP Module (resp. functions to read/write the filesystem, functions for geo-localisation, a function providing HTTP client features and function to read/write ZIP files). EXPath also defines two modules of a different nature: the Packaging System and the Webapp Module.

The Packaging System is the specification of a package format for XML technologies. It uses the ZIP format to gather in one single file all components and resources needed by a package (that is, a library or an application). The package contains also a package descriptor, which associates a public URI to each public component of the package. This URI can be used by user code to import those components exposed by the package. The Packaging System defines also an on-disk repository structure, so different processors and different implementations can share the same repository of packages. When compiling the user stylesheet / pipeline / query, the processors simply delegate the resolution of imported components to the repository:



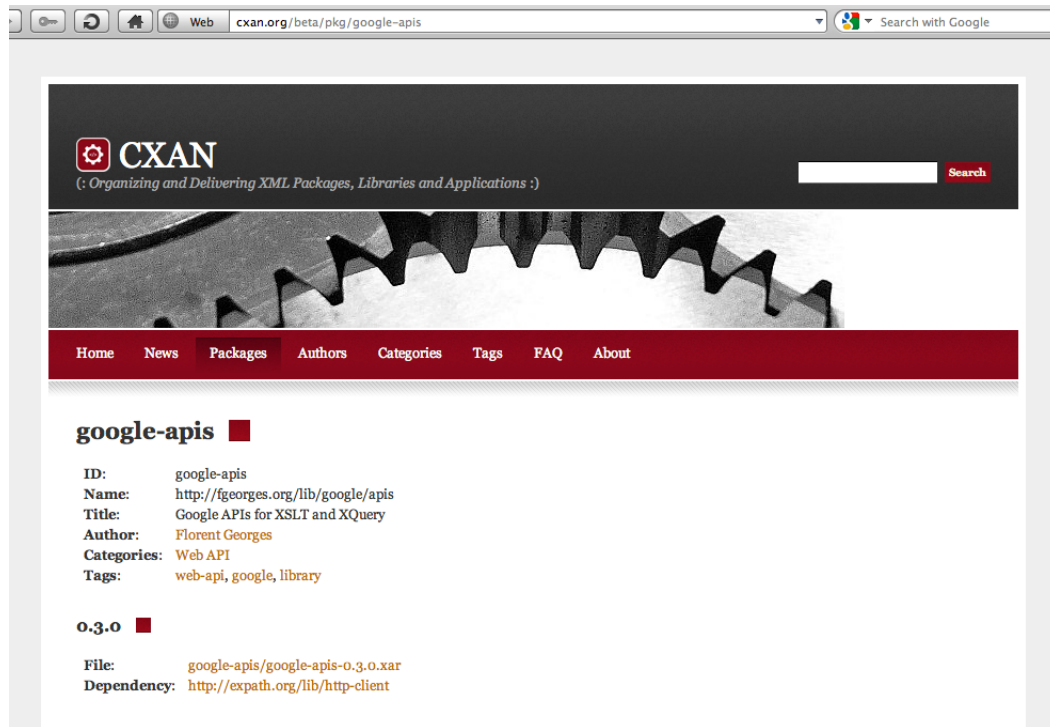
This package format makes it possible to distribute XML libraries and applications in a standard way, using a format supported by several processors. All the library author needs to do is to provide such a package, created using standard tools. The user just downloads the package and gives it to his/her repository manager, or directly to his/her processor, in order to install it automatically.

CXAN

The Packaging System makes it possible for a library author to put the package on his/her website in order for its user to download it and install it automatically. But still, a user has to find the website, find the package, download it, and invoke the repository manager with this package file to install it locally. And if the package depends on another, the user has to find the dependencies, and install them also. Recursively. All that process could be automated.

CXAN tries to solve that problem by providing two complementary components. The first component is the website. The CXAN website is aimed at gathering all known XML packages, at organizing them in a stable distribution, and at maintaining that distribution over the time. Every package in CXAN is given a unique ID, a abbreviation string. The second component is the CXAN client. The client is a program that manages parts of this stable distribution in a local repository. The client can install packages on the local machine by downloading them directly from the CXAN website, and resolving automatically the dependencies. There is a command-line client to maintain a standard on-disk repository, but every processor can define its own client, or an alternate client (for instance to provide a graphical interface in an XML IDE).

The website is organized as a large catalog of XML libraries and applications, that you can navigate through tags, authors and categories, or that you can search using some keywords or among the descriptions. It is located at <http://cxan.org/>. The following screenshot shows the details of the package `google-apis`, an XSLT and XQuery library to access Google APIs over HTTP:



The client is invoked from the command-line (although a graphical or web front-end could be written). It understands a few commands in order to find a package, install it, or remove it in the local repository. The following screenshot shows how to look for packages with the tag `google`. There is one, the package with the ID `google-apis`. We then display the details for that package. We also search for an HTTP Client implementation, then install it before installing the Google APIs. All informations and packages are retrieved directly from the CXAN website:

```
> cxan tag google
tags: google
subtags: library, web-api

google-apis - Google APIs for XSLT and XQuery

> cxan show google-apis
ID: google-apis
Package: http://fgeorges.org/lib/google/apis
Description: Google APIs for XSLT and XQuery
Author: fgeorges
Categories: web-api
Tags: web-api, google, library
Version: 0.3.0
Available: 0.2.0, 0.3.0
Depends on: http://expath.org/lib/http-client

> cxan search http
expath-http-client-exist - EXPath HTTP Client for eXist
expath-http-client-saxon - EXPath HTTP Client for Saxon

> cxan install expath-http-client-saxon

> cxan install google-apis

> 
```

Besides those two tools, the website and the client, the most valuable part of CXAN is the collection of packages itself. CXAN is not a brand-new idea, and is similar in spirit to systems like Debian's APT system (and its famous `apt-get` command), CTAN for TeX and LaTeX, or CPAN for Perl (also with a website at <http://cpan.org/> and a client to look up and install packages locally).

Webapp and Servlex

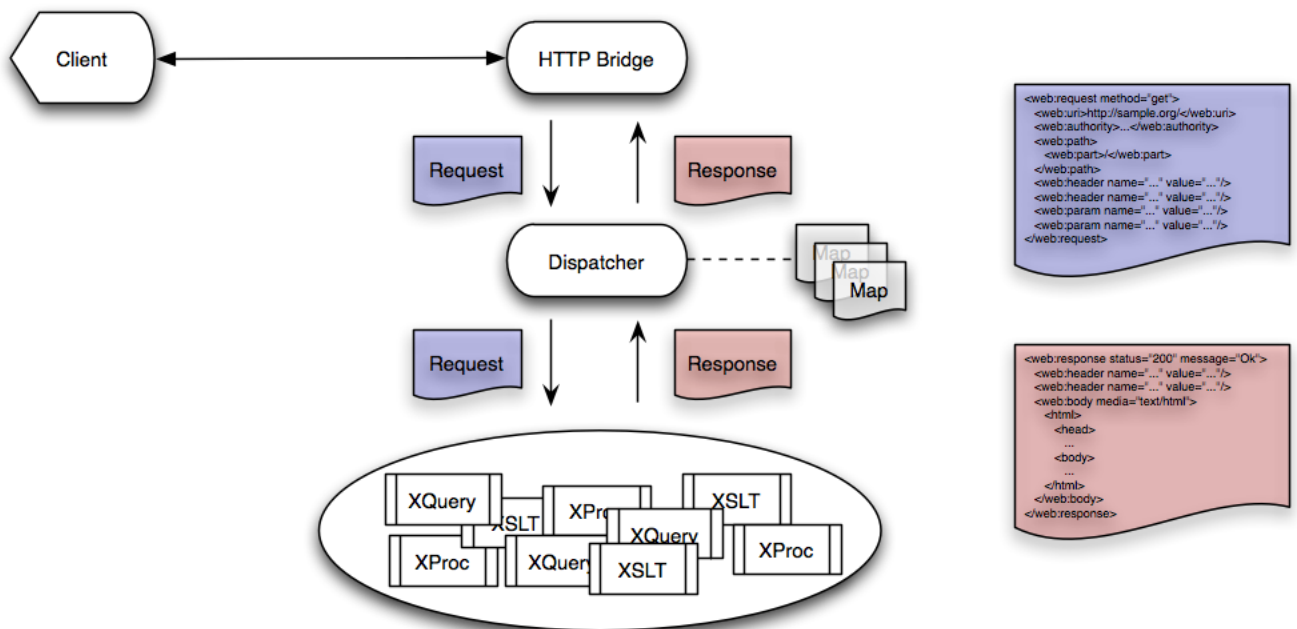
The EXPath Webapp Module defines a web container, using XSLT, XQuery and XProc to implement web applications. It defines how the HTTP requests are dispatched to those components based on a mapping between the request URI and the components. It also defines how the container communicates with the components (basically by providing them with an XML representation of the HTTP request, and by receiving in turn an XML representation of the HTTP response to send back to the client).

The purpose of this module is to provide the developer with a low-level, yet powerful way to map HTTP requests to XML components, without need for any other technology. It is defined independently on any processor, and can actually be implemented by all kind of processors. Most XML databases provide such a feature (usually trying to provide an API at a slightly higher level, sacrificing the power of a full HTTP support). Its place in the XML eco-system is similar to the place of the Servlet technology in the Java eco-system: quite low-level, but providing the ability to build more complex systems on top of it, entirely in XML.

Servlex is an open-source implementation of the Webapp Module, based on Saxon and Calabash as its XSLT, XQuery and XProc processors, and on the Java Servlet technology for its networking needs. It can be installed in any servlet container, like Tomcat, Glassfish, or Jetty. It is available on Google Code at <http://code.google.com/p/servlex/>.

The Webapp Module

The overall treatment of an in-bound HTTP request is as follows in the Webapp Module:



That is, the client sends a request. It is received by the webapp container. It is translated to an XML representation by the HTTP Bridge. This XML representation is a simple XML vocabulary giving information about the HTTP verb, the request URI, the URI parameters, the HTTP headers, and the entity content (e.g. in case of a PUT or a POST). Based on the request URI and on a set of maps, the Dispatcher finds the component to call in order to handle the request.

Once the correct component is found, it is called with the request as parameter. For instance, if the component is an XQuery function, the request is passed as a function parameter; if the component is an XSLT stylesheet, the request is passed as a stylesheet parameter. The result of the evaluation of the component must be the XML representation of the HTTP response to send back to the client. The HTTP request looks like the following:

```
<web:request servlet="package" path="/pkg/google-apis" method="get">
  <web:uri>http://cxan.org/pkg/google-apis?extra=param</web:uri>
  <web:authority>http://cxan.org</web:authority>
  <web:context-root></web:context-root>
  <web:path>
    <web:part>/pkg/</web:part>
    <web:match name="id">google-apis</web:part>
  </web:path>
  <web:param name="extra" value="param"/>
  <web:header name="host" value="cxan.org"/>
  <web:header name="user-agent" value="Opera/9.80 ..."/>
  ...
</web:request>
```

As you can see, this XML document contains all information about the HTTP request. The HTTP method of course (GET, POST, etc.) and everything related to the request URI: the full URI but also its authority part, the port number, the context root and the path within the web application. The webapp map can identify some parts in the URI using a regex and give them a name, so they can be easily retrieved from within the component. In the above example, the map says that everything matching the wildcard in `/pkg/*` must be given the name `id`, so it can be accessed in the request by the XPath `/web:request/web:path/web:match[@name eq 'id']`. The URI query parameter and the HTTP request headers are also easily accessible by name.

The entity content (aka the request body), if any, is also passed to the component. The bodies though are passed a bit differently. Instead of being part of the request document, the bodies are passed in a separate sequence (I say bodies, because in case of a multi-part request we can have several of them). They are parsed depending on their content type, so a textual body is passed as a string item, an XML content is parsed as a document node, an HTML content is tidied up and parsed in a document node, and everything else is passed as a base 64 binary item. A *description* of each body is inserted in the `web:request` though, describing its content type and a few other infos.

The component is called with the request document, and must provide as result a response document. The same way the request document represents the HTTP request the client sent, the response document represents the HTTP response to send back to the client. It looks like:

```
<web:response status="200" message="Ok">
  <web:header name="Extra-Header" value="..."/>
  <web:body content-type="text/html">
    <html>
      <head>
        <title>Hello</title>
      </head>
      <body>
        <p>Hello, world!</p>
      </body>
    </html>
  </web:body>
</web:response>
```

The response includes a status code and the status message of the HTTP response. It can also (and usually do) contain an entity content, the body of the response. In this case this is an HTML page, with the content type `text/html`. Optionally, the response document can set some headers on the HTTP response.

Besides components in XSLT, XQuery and XProc, a webapp can contain resources. They are also identified using a regex, but then the path is resolved in the webapp's directory and the webapp container

returns them as is from the filesystem. The map can set their MIME content type, and can also use a regex rewrite pattern to rewrite a resource URI path to a path in the filesystem. This is useful to have the webapp container serving directly paths like `/style/*.css` and `/images/*.png`, without actually calling any component and without having to generate the request document.

A web application is thus a set of components, along with a map (mapping request URIs to components). It is packaged using the Packaging System format, with the following structure:

```
expath-pkg.xml
expath-web.xml
the-webapp/
  component-one.xsl
  two.xproc
  tres.xqm
  any-dir/
    more-components.xproc
  images/
    logo.png
  style/
    layout.css
```

Because this is a standard package (with the addition of the webapp descriptor, aka the webapp map, `expath-web.xml`), all public components are associated an import URI. The webapp map can then use those absolute URIs to reference components, making it independent on the physical structure of the project. The webapp descriptor looks like the following:

```
<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>
    <xproc uri="http://example.org/ns/my-home.xproc"/>
    <url pattern="/" />
  </servlet>

  <servlet>
    <xquery function="app:other-page"/>
    <url pattern="/other"/>
  </servlet>

  <servlet>
    <xslt uri="http://example.org/ns/servlets.xsl" function="app:yet-page"/>
    <url pattern="/yet/(.+)"/>
      <match group="1" name="id"/>
    <url>
  </servlet>

  <servlet>
    <xslt uri="http://example.org/ns/catch-all.xsl"/>
    <url pattern="/.*"/>
  </servlet>

</webapp>
```

Besides some metadata like the webapp name and its title, the webapp descriptor is basically a sequence of components, each associated with a URL pattern. A URL is a regex. When a request is received by the webapp container, the webapp is identified by the context root (that is, the first level of the URI). Then

all the patterns in the corresponding webapp descriptor are tried to be matched against the request path, in order. The request is dispatched to the first one that matches (either a resource or a component). The components can be anything among the following types:

Table 1.

Language	Kind of component
XProc	Step
	Pipeline
XQuery	Function
	Main module
XSLT	Function
	Named template
	Stylesheet

Each kind of component defines the exact way it is evaluated, how the request is passed to the component, and how the component gives back the response. For instance, an XQuery or an XSLT function must have exactly two parameters: the first one is the `web:request` element, and the second one is the sequence (possible empty) of the entity content, aka the request bodies. The result of calling such a function must in turn give an element `web:response`, and possibly several subsequent items representing the response body. An XProc pipeline is evaluated the same way, but the specification defines instead specific port names for the request and the response.

Servlex

Servlex is an implementation of the Webapp Module. It is open-source and available on Google Code at <http://code.google.com/p/servlex/>. Under the hood, it is written in Java, it uses the Java Servlet technology for the link to HTTP, and it uses Saxon and Calabash as its XSLT, XQuery and XProc processors.

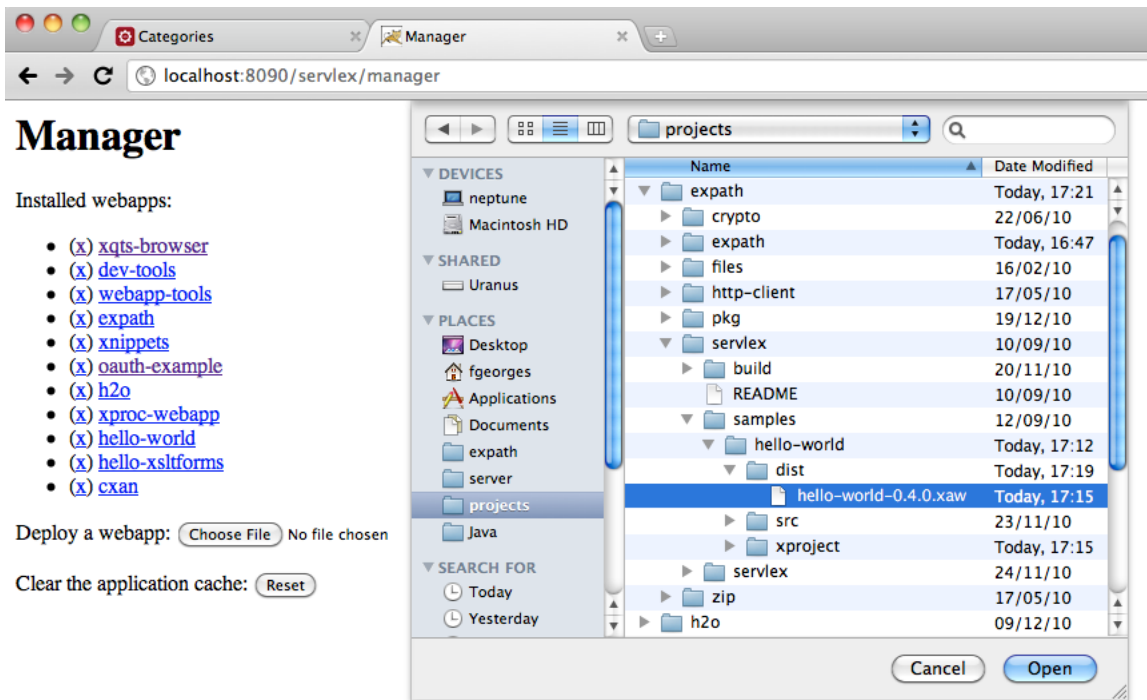
To install Servlex, you first need to get a Servlet container. The easiest is to install a container like Tomcat or Jetty. Then follow the instructions to deploy the Servlex WAR file in the container: go to your container admin console, select the WAR file on the disk and press the deploy button. As simple as that. The only option to configure is the location of the webapp repository. For instance in Tomcat, you can add the following line in `conf/catalina.properties`: `org.expath.servlex.repo.dir=/usr/share/servlex/repo`. This must be a standard EXPath package repository.

At startup, Servlex looks into that repository. Every package with a web descriptor, aka the file `expath-web.xml`, is considered a webapp. The descriptor is read, and the Servlex application list is initialized. Each webapp has an abbreviation used to plug it in the container URI space. For instance, let us assume Servlex has been deployed on a local Tomcat instance at `http://localhost:8080/servlex/`. When Servlex receives a request at `http://localhost:8080/servlex/myapp/some/thing`, it uses `myapp` as an ID of the application. Once it knows the application, it can retrieve its map. It then uses the path, here `/some/thing`, to find a component in the map, by trying to match the path against components URL regex.

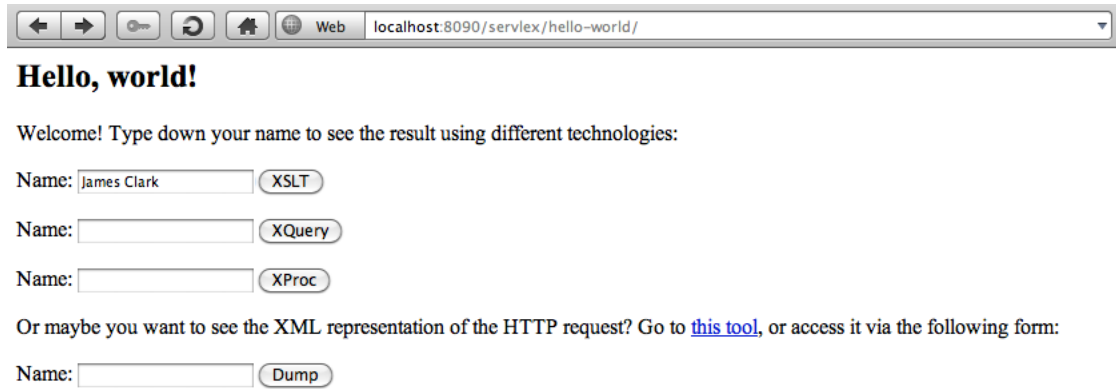
An interesting particularity of Servlex is its ability to have a read-only repository that does not use the filesystem. Thanks to the open-source implementation in Java of the Packaging System and its repository layout, Servlex can look instead in the classpath. Instead of using a directory on the filesystem, it uses the name of a Java package. For instance, let us say we have a JAR file with a Java package `org.example.repo`, and within this Java package, sub-packages and resources follow the same structure as an on-disk repository, but instead in the classpath. We can then use the name of this Java package to configure the repository of Servlex, instead of using a directory on the disk. This is particularly interesting

to deploy Servlex in disk-less environments like Google Appengine and Amazon Cloud EC2. Of course, a repository in the classpath is read-only, so you cannot install and remove webapps on the move, this is fixed at Servlex deployment.

Now that we have a Servlex instance up and running, let us have a look at a real sample of webapp. Servlex distribution comes with a simple example, called hello-world. The source and the compiled package are both included in the distribution. All you need to do in order to install the sample webapp is to go the Servlex Manager at <http://localhost:8080/servlex/manager>, select the file `hello-world-0.4.0.xaw` from the distribution (this is the webapp package), and press the deploy button. The package is read by Servlex, and added to the on-disk repository, so it will be available even after you restart Tomcat. The manager lists the installed applications, and allows you to remove them or to install new ones:



After you installed the webapp, you can directly access it at the address <http://localhost:8080/servlex/hello-world/>. This is a very simple application. The home page contains 4 forms. The first form asks for our name, then sends it to a page implemented in XSLT. The second form sends it to a page implemented in XQuery, and the third one in XProc. The last form sends you to an online tool that dumps the XML request document (representing the HTTP request in XML), located at <http://h2oconsulting.be/tools/dump>. If you fill in the first form with "James Clark" and press the button, you see a simple page with the text "Hello, James Clark! (in XSLT)":



Hello, world!

Welcome! Type down your name to see the result using different technologies:

Name:

Name:

Name:

Or maybe you want to see the XML representation of the HTTP request? Go to [this tool](#), or access it via the following form:

Name:

When you press the button "XSLT", the HTML form sends a simple HTTP GET request to the URI `http://localhost:8080/servlex/hello-world/xslt?who=James+Clark`. When Servlex receives this request, it first extracts the context root to determine which application is responsible for handling this request. The string `hello-world` helps it identifying the application, and finding its webapp descriptor. In this descriptor, it looks for a component matching the path `/xslt`. It finds the following match:

```
<servlet>
  <xslt uri="http://expath.org/ns/samples/servlex/hello.xml"
        function="app:hello-xslt"/>
  <url pattern="/xslt"/>
</servlet>
```

This component represents a function `app:hello-xslt`, which is defined in the stylesheet with the public URI `http://expath.org/ns/samples/servlex/hello.xml`. Servlex constructs then the following request document:

```
<web:request servlet="xslt" path="/xslt" method="get">
  <web:uri>http://localhost:8080/servlex/hello-world/xslt?who=James+Clark</web:uri>
  <web:authority>http://localhost:8080</web:authority>
  <web:context-root>/servlex/hello-world</web:context-root>
  <web:path>
    <web:part>/xslt</web:part>
  </web:path>
  <web:param name="who" value="James Clark"/>
  <web:header name="host" value="localhost"/>
  <web:header name="user-agent" value="Opera/9.80 ..."/>
  ...
</web:request>
```

Servlex then calls the component with this request document. In this case, this is the XSLT function `app:hello-xslt`. An XSLT function used as a servlet must accept two parameters: the first one is the element `web:request`, the second one is the sequence of bodies (here empty as this is a GET request). In this example, this function simply has to get the query parameter value from `$request/web:param[@name eq 'who']/@value`, and to format a simple HTTP response document and a simple HTML page to return to the client. The function looks like the following:

```
<xsl:function name="app:hello-xslt">
  <!-- the representation of the http request, given by servlex -->
  <xsl:param name="request" as="element(web:request)"/>
  <xsl:param name="bodies" as="item()*"/>

  <!-- compute the message, based on the param 'who' -->
  <xsl:variable name="who" select="$request/web:param[@name eq 'who']/@value"/>
  <xsl:variable name="greetings" select="concat('Hello, ', $who, '!')"/>

  <!-- first return the description of the http response -->
  <web:response status="200" message="Ok">
```

```
<web:body content-type="application/xml" method="xhtml"/>
</web:response>

<!-- then return the body of the response, an html page -->
<html>
  <head>
    <title>
      <xsl:value-of select="$greetings"/>
    </title>
  </head>
  <body>
    <p>
      <xsl:value-of select="$greetings"/>
      <xsl:text> (in XSLT)</xsl:text>
    </p>
  </body>
</html>
</xsl:function>
```

The sequence returned by the function (here an element `web:response` and a HTML element) is used by Servlex to send a response back to the client, with the code 200 Ok, the content type `application/xml` and the HTML page as payload. The application source code is structured as follows, but this is up to the developer:

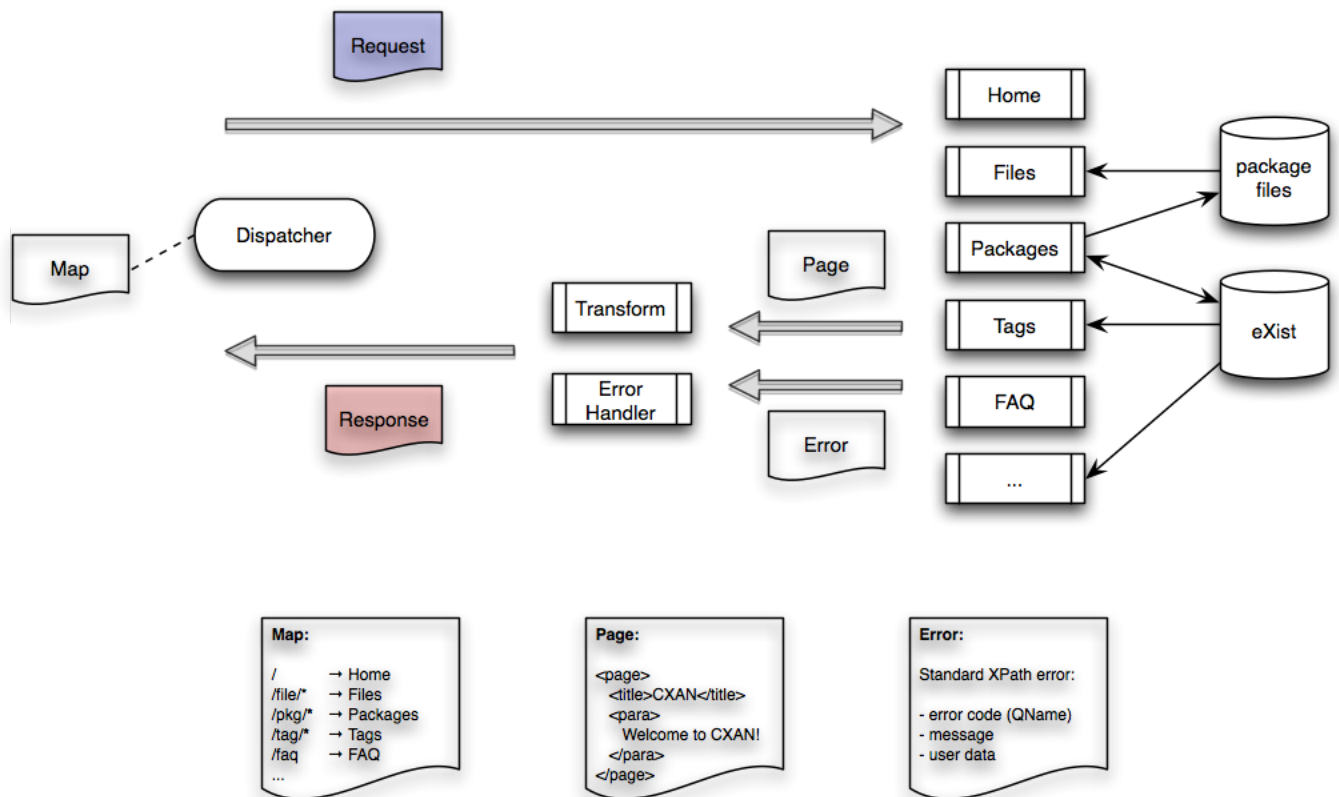
```
hello-world/
  src/
    hello.xproc
    hello.xql
    hello.xsl
    index.html
  xproject/
    expath-web.xml
    project.xml
```

The CXAN website

Now that we have met the basics of the framework, let us have a look at a real-world example: the CXAN website. The purpose of this website, has stated in the introduction, is double: first to be an online catalog of packages, to be browsed by human, with a graphical environment, and secondly to be a central server for CXAN clients to communicate to, through HTTP and XML, in order to maintain a local install of CXAN packages, by allowing searching and installing packages.

Both aspects are actually very similar. The website has to respond to HTTP requests sent to structured URIs. In both cases, the responses carry the same semantics. The difference is that in one case, the format of the information is a HTML page aimed at human beings, and in the other case, the format is an XML message aimed at a computer program. This section will first focus on the HTML part of the website, then will show how both parts are actually implemented in common.

So the main business of the website is to maintain a set of packages and to provide a way to navigate and to search them, as well as to display their details and downloading them. There must also be a way to upload a new package. So the overall architecture is pretty straight-forward: a plain directory on the filesystem to save the package files, an XML database to maintain the metadata about packages (here `eXist`, but that could be any one usable from XProc, like `Qizx`, `MarkLogic`, and many others), a set of XProc pipelines, each one implementing a particular page of the website, and a site-wide transform to apply a consistent layout accross all pages:



What happens when the user points his/her browser to the address `http://cxan.org/`? The HTTP request hits Servlex, which knows this request is aimed at the CXAN web application. In the general case, Servlex can host several applications, each of them with its own name. In the case of CXAN, the Servlex instance is dedicated to the single CXAN application. It gets the web descriptor for CXAN, and looks for a component matching the URI `/`. It finds the Home component, based on the following element in the descriptor:

```

<servlet name="home">
  <xproc uri="http://cxan.org/website/pages/home.xproc"/>
  <url pattern="/" />
</servlet>

```

Servlex then builds the `web:request` element, with the relevant information: the headers, the request URI (the raw URI as well as a *parsed* version presenting the domain name, the port number, the parameters, etc.) It uses Calabash to call the corresponding pipeline, connecting the `web:request` document to the pipeline port `request`. As you can see, the pipeline is identified by an absolute URI. The packaging support configured on Calabash with the Servlex own repository takes care of resolving this URI correctly within the repository. The case of the Home component is very simple: it simply returns the following abstract page description:

```

<page menu="home">
  <title>CXAN</title>
  <image src="images/cezanne.jpg" alt="Cezanne"/>
  <para>CXAN stands for <italic>Comprehensive XML Archive Network</italic>.
    If you know CTAN or CPAN, resp. for (La)TeX and Perl, then you already
    understood what this website is all about: providing a central place
    to collect and organize existing libraries and applications written in
    XML technologies, like XSLT, XQuery and XProc.</para>
</page>

```

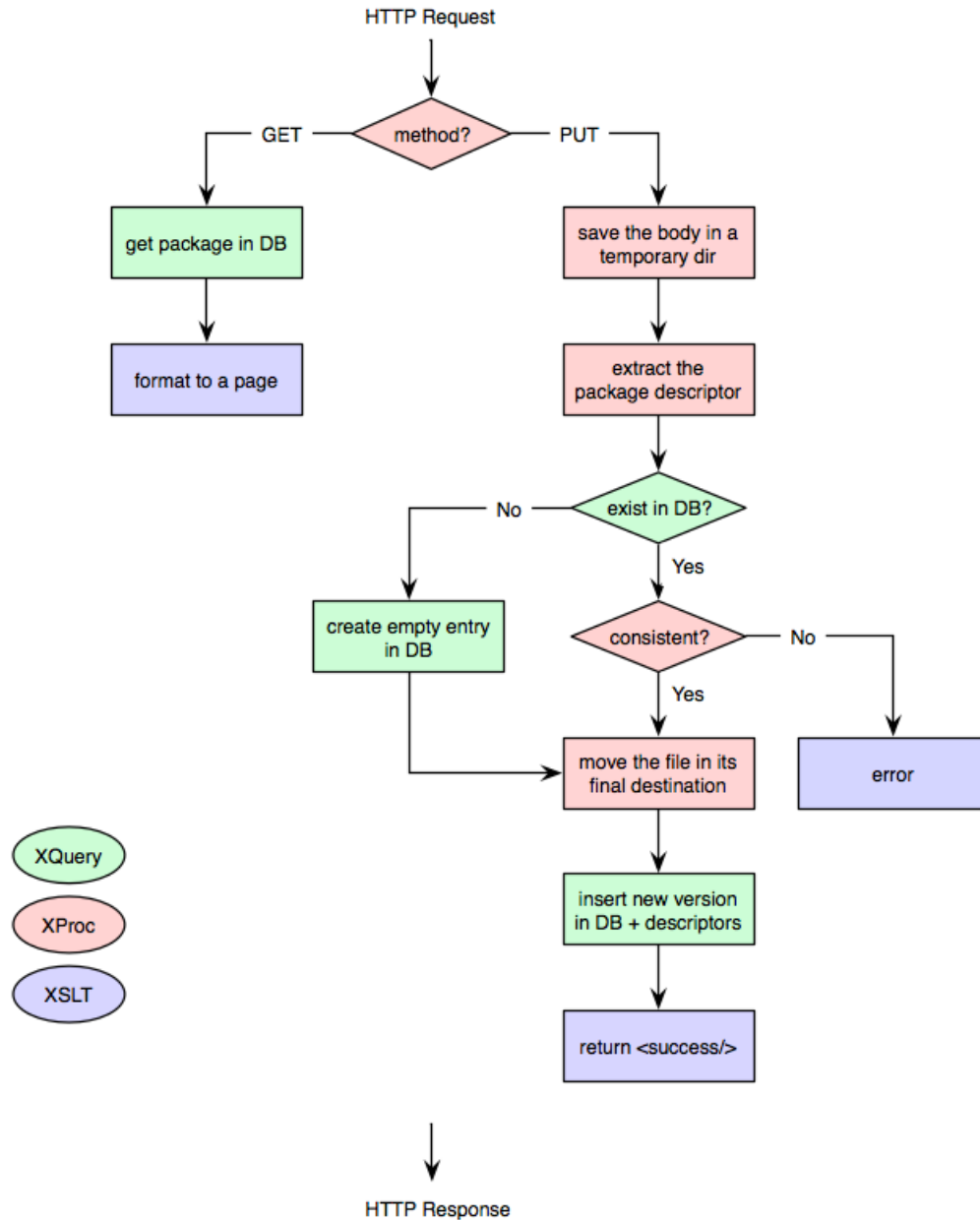
In the basic case, a component must return a full description of the HTTP request to return to the user, using an element `web:response`. But for all webpages, the HTTP response will always be the same (except the content of the returned body, that is the page itself): a status code `200 Ok`, and a content type header with `text/html`. Besides, all pages share the same structure and a consistent layout. So instead of repeating this information for every pages, in every pipeline, the CXAN website application defines an abstract representation of a page: an element `page`, with a `title`, some `para`, `image`, `italic text`, `code snippets`, `list` and some `table` elements. Each pipeline has to focus only on building such a simple description of the page to display to the user. In the web descriptor, the application sets a transformer for all pages, which is an XSLT stylesheet. This stylesheet generates the `web:response` element Servlex is expecting, including the HTML version of the page, transformed from its abstract representation.

Of course, when the browser receives the HTML page, it displays it. When doing so, it finds that the page refers to some images, CSS stylesheets and Javascript files. So the same cycle starts again: it sends as much HTTP request as resources to retrieve. But because those resources are static files, they are handled differently. When Servlex receives the request, it looks in the web descriptor for a component matching the URI. It then finds a specific kind of component: `resource` components. The resource components specify also a URL pattern, and set the content type of the resource (they can also contain a rewriting rule based on the third parameter of `fn:replace()`). The resource is then read from within the package, based on its name, and returned directly to the user:

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

So far, so good. But what happens when the component encounters an error? For instance if the user sented wrong parameters, or the database was not reachable. In that case the component just throw an error (using the standard XPath function `fn:error()` or the XProc step `p:error`). This error is caught by Servlex, which passes it to an error handler configured in the web descriptor, which formats a nice error page for the user, as well as the corresponding `web:response` (in particular the HTTP status code of the response, for instance 404 or 500). Thanks to the standard XPath error mechanism, the error itself can carry all the information needed: the CXAN application allows the error item to contain the HTTP status code to use, as well as a user-friendly message.

All the logic is thus implemented by XProc components. They talk to eXist using its REST-like API and `p:http-request`. The following is an abstract view of the most complex pipeline in the website, the Package component:



A flowchart is very handy to represent an XProc pipeline. The component handles two different processing: on a GET request, it gets the package information from eXist, and builds the corresponding abstract page; on a PUT, the request must contain, at least, the package itself (the XAR file). The webapp analyzes it, saves it on the disk and update the information in eXist. Of course, that means doing some checks: is the uploaded content a proper EXPath package, does the package already exist in the database, if yes is the new package consistent with the existing information? The advantage of XProc is that the developer can use whatever technology that best suits his/her needs: XProc itself for the flow control, XSLT to transform documents, and XQuery to query or update a database.

We have seen so far how the website serves HTML pages to a browser. But as we saw earlier, this is only one of the two goals of the CXAN website. The second goal is to enable tools to communicate with it in order to search for packages, download them, display information about them and upload new packages, for instance from the command line.

Indeed, semantically this is exactly what it already does with webpages. Technically, instead of serving HTML pages, it just has to serve XML document carrying the same information. Because the same set of URI is used in both cases, it uses the REST principle based on the HTTP header `Accept` to differentiate between both. Internally, the pipelines use a simple XML format to flow between the steps (describing packages, tags, categories, etc.) The last step in most pipelines checks the value of the `Accept` header: if it is `application/xml`, it sends the XML back to the user as is, if not it first transforms it to an abstract page.

This way, the webapp provides, almost for free, a REST-like API in addition to the traditional HTML website. A client XProc application uses this API to provide a command-line utility to the user, in order to maintain packages in a local repository, automatically installed and upgraded from the CXAN website (there is a screenshot of this `cxan` command-line utility in the introduction).

The development project

How is organized the source code of this web application? The project directory structure is as follows:

```
cxan/website/  
  dist/  
    cxan-website-1.0.0.xar  
    cxan-website-1.0.0.zip  
  src/  
    images/  
    ...  
    pages/  
      home.xproc  
    ...  
    lib/  
      tools.xpl  
    ...  
    page.xsl  
  xproject/  
    expath-web.xml  
    project.xml
```

The overall structure follows the EXPath project directory layout. The `xproject` directory contains information about the project, as well as the web descriptor (in case of a webapp project), the `src` directory contains the source of the project, and `dist` is the directory where final packages are placed. In addition, components must contain the public URI to use in the package (e.g. for an XQuery library this is its namespace URI, for an XSLT stylesheet this is set using the `/xsl:stylesheet/pkg:import-uri` element). When the developer invokes the command `xproj build` from within the project directory, it uses those informations to build automatically the package descriptor and the package itself, and put the result in the `dist` directory. The package is then ready to be directly deployed in Servlex.

I will not discuss here the details of the `xproj` program (written in XProc, of course), this could be the subject of paper on its own. The idea is to use some kind of *annotations* in order to configure the public import URIs within each component instead of having to maintain an external package descriptor (as needed to build a proper EXPath package). While that is not supported yet, it is expected to create the same kind of mechanism for the web descriptor. Using some annotations, it is easily possible to maintain the URL mapping and the parameters accepted or required by a component, directly within the source file (the query, stylesheet or pipeline). By doing so, the developer will not have anymore to maintain the `expath-web.xml` descriptor manually, it will be generated based on those annotations.

Conclusion

The main goal of the Webapp Module is to be the glue between XML technologies and HTTP on the server-side. The design choice is to provide the full power and flexibility of HTTP. This choice can make the module a bit low-level, but as we have seen, this is very easy with technologies like XSLT to create an

intermediary layer of abstraction in order to be able to write the web components at a higher level of abstraction. Because it provides a full, consistent mapping of HTTP natively oriented towards XML, it never locks its users in some restrictions because of some handy abstraction which does not fit all use cases.

Because it provides the full HTTP protocol information to the XML technologies, it can be used to easily create websites, REST web services, SOAP/WSDL web services, and everything you can do on a HTTP server. And thanks to Servlex, such applications can be hosted for free on Google Appengine or other cloud services like Amazon's.