Using the BioCatalogue API in Java

XmlBeans

I have been using XmlBeans version 2.4.0, which was the latest available. At the moment of writing this document a later version (2.5.0) is already available.

Information about XmlBeans is available here:

- http://xmlbeans.apache.org/index.html
- http://xmlbeans.apache.org/docs/2.0.0/guide/conGettingStartedwithXMLBeans.html

Setting up Java Project

There is a detailed XmlBeans installation guide available:
http://xmlbeans.apache.org/documentation/conInstallGuide.html

This page gives a good explanation of which JARs must be on the classpath:
http://www.javabeat.net/qna/279-what-jars-do-i-need-on-the-classpath-to-use-x/

In essence, Java project must have (at least) the following XmlBeans JARs on the classpath in order to build properly:

- xbean.jar
- jsr173_1.0_api.jar
- (if XPath / XQuery support is required in the project, then some additional libraries need to be included – some are included within XmlBeans distribution, but some need to be downloaded separately).

If Maven is used to build the project, the following should be included into the project *.pom file (I believe this dependency doesn't require any specific repository to be included, as it is a commonly used library and appears to be available from default Maven repositories):

```
<dependency>
  <groupId>org.apache.xmlbeans</groupId>
  <artifactId>xmlbeans</artifactId>
  <version>2.4.0</version>
</dependency>
```

Compiling API Binding Classes

This is achieved by using utility called “scomp.exe” that can be found in “bin” folder within the installation directory of XmlBeans.

The schema compiler has a range of various options (including the choice of version of Java compiler to use and whether the classes should only be compiled or also bundled as a JAR file).

“-dl” option is very important, because it allows to traverse includes and actually use all referenced schemas in the compilation process.

This first example shows a Windows batch file that simply compiles all classes from a specified XSD schema (i.e. the BioCatalogue schema):

```
@echo off

REM -src . -- put source files here
REM -srconly -- only sources, no compiling of java classes, no jar bundling
REM -compiler -- where to find javac
REM -javasource -- which JAVA version to aim for (1.5 uses generics)
REM -dl -- allows download of referenced schemas

scomp -src . -srconly -compiler "%JAVA_HOME%\bin\javac.exe" -javasource 1.5 -dl
http://sandbox.biocatalogue.org/2009/xml/rest/schema-v1.xsd
```

The second example does a similar thing, but after compilation of classes it also creates a JAR file which includes the freshly-compiled classes:

```
@echo off

REM -src . -- put source files here
REM -compiler -- where to find javac
REM -javasource -- which JAVA version to aim for (1.5 uses generics)
REM -out -- specifies the name of the target JAR file
REM -dl -- allows download of referenced schemas

scomp -src . -compiler "%JAVA_HOME%\bin\javac.exe" -javasource 1.5 -out
biocatalogue_api_classes.jar -dl
http://sandbox.biocatalogue.org/2009/xml/rest/schema-v1.xsd
```

Using Compiled Binding Classes Within the Project

Various combinations of usage of compiled classes – directly as source files or as a compiled JAR – are possible depending on the development environment.

I have made use of both:

- sources of compiled classes are linked into the build path in Eclipse, so that it is easily possible to access the inner workings of the classes (has proven to be very useful – some well-written automatic comments are made by XmlBeans and it may be necessary/more convenient to review those not just through Javadoc, but directly in the source code) from within the project;
- generated JAR file is on the classpath of the standalone version of the BioCatalogue plugin – the only reason for using it was to speed-up testing of certain scenarios (using the two doesn't cause any confusion, given that both generated source files and the JAR file are in sync – which is easy to achieve by using the build scripts from the earlier section).

Examples of Using the BioCatalogue API

The simplest example of making a GET request

```
// user-agent can be anything (or not even specified at all), but it's better
// to set one, so that it is possible to perform usage analysis on the server
// side – if necessary
final String USER_AGENT = "[any string describing the API caller]";

String urlToConnectTo = "http://sandbox.biocatalogue.org/services";
URL url = new URL(urlToConnectTo);

// open the connection and set various request headers;
// setting the Accept header makes sure that the API returns XML data, not
// HTML data – many URLs overlap with those used in HTTP interface, so using
// this header is a must
HttpURLConnection conn = (HttpURLConnection) url.openConnection();
conn.setRequestProperty("User-Agent", USER_AGENT);
conn.setRequestProperty("Accept", "application/xml");

int iResponseCode = conn.getResponseCode();
InputStream serverResponse = null;

switch (iResponseCode)
{
  case HttpURLConnection.HTTP_OK:
    // regular operation – can directly use the response data input stream
    serverResponse = conn.getInputStream();

  case HttpURLConnection.HTTP_BAD_REQUEST:
    // this was a bad XML request - need full XML response to read the error
    // message from that the server has provided (if any);
    // Java throws IOException if getInputStream() is used when non HTTP_OK
    // response code was received -
    // hence can use getErrorStream() straight away to fetch the error document
    serverResponse = conn.getErrorStream();

  case HttpURLConnection.HTTP_UNAUTHORIZED:
    // this content is not authorised for current user – i.e. the application
    // may need to prompt the current user to provide their login credentials
    /* decide what to do here */

  case HttpURLConnection.HTTP_NOT_FOUND:
    // nothing was found at the provided URL – may try to use error stream
    // (or just do nothing)
    serverResponse = conn.getErrorStream();

  default:
    // unexpected response code - raise an exception
    throw new IOException("Received unexpected HTTP response code (" +
      iResponseCode + ") while fetching data at " : urlToConnectTo);
}

/*
 * now need to check if “serverResponse” is not null – if yes, can proceed to
 * reading the data received from the server
 */
```

Using the Response Data Directly

The regular way of using the response data would be to convert it into Java objects with XmlBeans' generated API binding classes, but before that let's look at an alternative.

In this example we just use the InputStream provided by the HttpURLConnection to read the data directly as text:

```
// assuming that we continue from the previous example and “serverResponse”
// is the InputStream that we obtained from the API request in the first
// example
final StringBuilder text = new StringBuilder();
try {
  BufferedReader br = new BufferedReader(new InputStreamReader(serverResponse));
  String str = "";

  while ((str = br.readLine()) != null) {
    text.append(str + "\n");
  }

  br.close();
}
catch (Exception e) {
  System.err.println("An error has occurred, details:\n" + e.getMessage());
}

/*
 * “text” variable now contains the whole of the server's response -
 * could be used as “text.toString()” or in any other required way
 */
```

Converting Server's Response Data with XmlBeans

In this section we look at the regular way of using the BioCatalogue API's response data.

First of all a little explanation of how XmlBeans handles the conversion procedure. I will use the following example to help with this:

if we accessed *http://sandbox.biocatalogue.org/services* (with “Accept: application/xml” header set in the request OR by using the URL in the web browser, but appending “.xml” at the end of it, the API output data looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<services xlink:href="http://sandbox.biocatalogue.org/services"
      resourceType="Services"
      xsi:schemaLocation="http://www.biocatalogue.org/2009/xml/rest
      http://sandbox.biocatalogue.org/2009/xml/rest/schema-v1.xsd"
      xmlns:xlink="http://www.w3.org/1999/xlink"
      xmlns:dc="http://purl.org/dc/elements/1.1/"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns="http://www.biocatalogue.org/2009/xml/rest"
      xmlns:dcterms="http://purl.org/dc/terms/"
>
  <parameters>
    ...
  </parameters>
```

```
  <statistics>
    ...
  </statistics>
  <results>
    <service … />
    ...
  </results>
  <related>
    ...
  </related>
</services>
```

There are various elements in this XML document (`<services>`, `<parameters>`, `<statistics>`, `<results>`, `<service>`, `<related>`). For each element type in this document (and indeed for any element type defined in the BioCatalogue API XSD schema) XmlBeans creates a corresponding Java with an identical name (i.e. for `<service>` element `Service` class is generated) during the XSD schema compilation process that was described earlier.

Classes that are generated can essentially be seen as *Java beans*, which simply hold all the contents of the corresponding XML element and allow easy read and write access to its fields. Names of the automatically-generated getters and setters for all fields are intuitive and can be easily predicted simply by inspecting a sample of the API response data.

Some XML element types defined in the BioCatalogue XSD schema may act as a top-level elements in the API response XML documents. Generally, the name of the top-level element type can be derived from the API endpoint that is accessed -

- *http://sandbox.biocatalogue.org/services* - `<services>`
- *http://sandbox.biocatalogue.org/services/filters* - `<filters>`
- *http://sandbox.biocatalogue.org/soap_operations* - `<soapOperations>`
- `etc.`

In the current example the top-level element is `<services>...</services>`. XmlBeans generates an additional class for each of these top-level element types – these classes share a naming convention where the name of the XML element type is taken as-is with a suffix `"Document"` appended to it. For instance, for the `<service>` object has an associated `ServiceDocument` class. All "document" classes are used to perform the actual parsing of the API data and to retrieve the corresponding underlying object.

The following example provides the details:

```
/*
 * Assume that we have made a request to the BioCatalogue API
 * (accessed endpoint: http://sandbox.biocatalogue.org/services)
 * identically to the way it was done in the first example.
 *
 * We then have and instance of InputStream "serverResponse" initialised
 * properly and ready for reading the input data.
 */

// First of all we use ServicesDocument to get the underlying Services object
```

```
Services servicesData =
ServicesDocument.Factory.parse(serverResponse).getServices();

// now “servicesData” may be used to get hold of any of its children elements
Parameters servicesParameters = servicesData.getParameters();
Statistics servicesStatistics = servicesData.getStatistics();

// a more interesting case is when we get hold of the collection of Service
// elements held within the <results> child of <services>
List<Service> individualServiceObjectList =
servicesData.getResults().getServiceList();

/*
 * Obtained objects may be used to drill-down to the contents of those elements
 * and so on recursively – this technique provides an easy way to get hold of
 * all contents of the API response.
 */
```