Using Java APIs to manage XML Publisher Document Templates and Delivery Requests

Brian J Looman
Greybrooke Consulting, Inc.

Introduction

XML Publisher already gives customers a great deal of flexibility for building and delivering professional documents. Using the standard Java APIs, a number of additional extensions can be added to Oracle E-Business to provide an endless tool-kit. Whether the documents need to be emailed, faxed, printed, or ftp'd, all delivery methods can be handled through the same common functionality.

Challenge

Oracle Applications utilizes only a small amount of the build and delivery functionality of XML Publisher. Even with a large library of Java delivery and document template APIs within XML Publisher, only the simplest methods are built into the Concurrent Manager and Post Processor.

Solution

Using the XML Publisher APIs, developers can build a number of different solutions to build and deliver documents. These extensions can then be called from any of the following: Java, PL/SQL, the Concurrent Manager, BPEL, or even shell scripts.

Benefits

- Extend XML Publisher to accommodate for all business document and delivery needs.
- Build a generic solution that can be used for delivering any document produced by the Oracle E-Business Suite.
- Control the delivery of all XML-based documents from the Oracle Concurrent Manager.

Implementation

The first step is to build the common Java classes that would be the base for XML Publisher extensions. These classes can generate XML from Data Definitions, create PDF documents using the XML Templates, deliver documents via email or fax, and any combination of these. Here is a Java method that will produce a PDF document (as an InputStream) for XML data using the given XML Publisher Template.

```
public static InputStream createDocument
    ( AppsContext appsContext, String appName, String templateCode, String language,
      String territory, InputStream xmlData ) throws Exception {
  // Initialize an OutputStream for the document output.
  ByteArrayOutputStream document = null;

  // Build a PDF document for the XML Data using the given XML Publisher Template.
```

```
    TemplateHelper.processTemplate
    ( appsContext, appName, templateCode, language,
      territory, xmlData, TemplateHelper.OUTPUT_TYPE_PDF,
      null, (OutputStream) document );

    // Return an InputStream that can delivered via any of the delivery methods.
    return (InputStream)(new ByteArrayInputStream( document.toByteArray() ) );
  }
```

Here is another base method that can be used to process a XML Publisher Data Definition with the given parameters to produce XML data (as an InputStream):

```
  public static InputStream createXmlData
      ( AppsContext appsContext, String appName,
        String dataTemplateCode, String filePath, Hashtable parameters ) throws Exception {
    // Initialize a Data Definition object for the given JDBC Connection.
    DataTemplate datatemplate = new DataTemplate();
    datatemplate.setOracleConnection( appsContext.getJDBCConnection() );

    // Create a temporary file for storing the XML Output.
    File tmpXML = File.createTempFile( "tmpxml", null );

    // Generate the XML Data for the Data Definition with the given parameters.
    datatemplate.writeXML
    ( appName, dataTemplateCode, parameters, tmpXML.getAbsolutePath(), null );

    // Return an InputStream of XML data.
    return (InputStream)(new FileInputStream( tmpXML ) );
  }
```

As a last example, this base method will fax a document to the given fax number:

```
  import java.io.*;
  import oracle.apps.xdo.delivery.*;
  import oracle.apps.xdo.delivery.smtp.*;

  ...

  public static void faxDocument
      ( String faxHost, String faxPort, String faxPrintQueue,
        String faxNumber, InputStream document ) throws Exception {
    // Initialize XML Publisher Delivery Manager for faxing.
    DeliveryManager delivMgr = new DeliveryManager();
    DeliveryRequest request = delivMgr.createRequest(DeliveryManager.TYPE_IPP_FAX);

    // Set all properties for the given fax request.
    request.addProperty( DeliveryPropertyDefinitions.IPP_HOST, faxHost );
    request.addProperty( DeliveryPropertyDefinitions.IPP_PORT, faxPort );
    request.addProperty( DeliveryPropertyDefinitions.IPP_PRINTER_NAME, faxPrintQueue );
    request.addProperty( DeliveryPropertyDefinitions.IPP_PHONE_NUMBER, faxNumber );

    // Add the document output to be faxed.
    request.setDocument( document );

    // Submit the fax.
    request.submit();
    request.close();
  }
```

All these examples demonstrate the many capabilities of the XML Publisher APIs. But in order to fully utilize XML Publisher within Oracle Applications, these must be built upon. A common need by customers is the ability to email or fax the output of a standard Oracle Applications report without customizing or rebuilding the report. Standard Concurrent Program functionality would allow the customer to email a person setup in Oracle Applications through a notification. However, faxing the output or emailing the output to a user-defined email address is not available.

A quick solution would be to add the standard report to a request set and having a second program responsible for delivering the output. This second program would do nothing more than read the output from the standard Oracle Report and use the methods described above to fax or email the output. Here is the source for this second Concurrent Program:

```
import java.io.*;
import java.sql.*;
import oracle.apps.fnd.cp.request.*;
import oracle.apps.fnd.util.*;

public class FaxOutput implements JavaConcurrentProgram {

  // Returns the XML Output from the previous request of a request set.
  private static InputStream getDocument(CpContext cpContext)
        throws Exception {
    PreparedStatement preparedStatement = null;
    ResultSet resultSet = null;

    try {
      // Get the request_id of the current request.
      int requestId = cpContext.getReqDetails().getRequestId();

      // SQL to retrieve the output file location from the previous program.
      StringBuffer sqlSB = new StringBuffer();
      sqlSB.append( "SELECT cr1.outfile_name, " );
      sqlSB.append( "  FROM fnd_concurrent_requests cr1, " );
      sqlSB.append( "       fnd_concurrent_requests cr2 " );
      sqlSB.append( " WHERE cr1.phase_code = 'C' " );
      sqlSB.append( "   AND cr1.status_code = 'C' " );
      sqlSB.append( "   AND cr1.output_file_type = 'XML' " );
      sqlSB.append( "   AND cr1.parent_request_id = cr2.parent_request_id " );
      sqlSB.append( "   AND cr2.request_id = :1 " );
      sqlSB.append( "   AND cr1.request_id < :2 " );
      sqlSB.append( " ORDER BY cr1.request_id DESC " );

      // Execute the SQL statement using the request_id.
      preparedStatement =
        cpContext.getJDBCConnection().prepareStatement(sqlSB.toString());
      preparedStatement.setInt( 1, requestId );
      preparedStatement.setInt( 2, requestId );
      resultSet = preparedStatement.executeQuery();
      resultSet.next();

      // Get the value from the OUTFILE_NAME column.
      String outfileName = resultSet.getString( "OUTFILE_NAME" );

      resultSet.close();
      preparedStatement.close();

      // return a InputStream containing the XML Output
      return (InputStream) new FileInputStream( new File( outfileName ) );
    }
    catch( SQLException sqlException )
    { try { resultSet.close(); }
      catch( Exception e ) { }
      try { preparedStatement.close(); }
      catch( Exception e ) { }
      finally { }
      throw sqlException;
    }

  }

  // Run method for the Oracle Applications Java Concurrent Program.
  public static void runProgram(CpContext cpContext) {
    try {
      // Get the parameters for this request.
      ParameterList parameterList = cpContext.getParameterList();

      // Get all the parameter values (must be in this order)
      String host = parameterList.nextParameter().getValue();
      String port = parameterList.nextParameter().getValue();

      String faxPrinter = parameterList.nextParameter().getValue();
      String faxNumber = parameterList.nextParameter().getValue();

      // Get the Output from the previous request in the request set.
      InputStream document = getDocument( cpContext );

      // Fax the output to the given faxNumber.
      DocumentManager.faxDocument
      ( host, port, faxPrinter, faxNumber, document );

      // Set request as Completed with Success.
      cpContext.getReqCompletion().setCompletion( 0, "SUCCESS" );
    }
    catch ( Exception e ) {
      if ( cpContext.getReqCompletion() != null ) {
        // Set request as Completed with Error.
        e.printStackTrace();
        cpContext.getReqCompletion().setCompletion( 2, "ERROR" );
      }
    }
  }
```

In this example the presentable document would be ready for delivery without XML Publisher processing it. However, if the output from the program were XML data, a few additional parameters with a small change to the code would generate the final document right before delivery. Here is the addition:

```
  ...
  String appName = parameterList.nextParameter().getValue();
  String templateCode = parameterList.nextParameter().getValue();
  String language = parameterList.nextParameter().getValue();
  String territory = parameterList.nextParameter().getValue();

  // Get the XML Output from the previous request in the request set.
  InputStream xmlOutput = getDocument( cpContext );

  // Process the XML Output using the XML Template defined.
  InputStream document =
    DocumentManager.createDocument
```

```
        ( cpContext, appName, templateCode, language, territory, xmlOutput );
    ...
```

So you can see from this small example, the potential these simple XML Publisher methods have to build full solutions within the Applications. The code is very reusable on many scales and can be a quick solution for processing and delivering all customer documents.

About the Author

Brian Looman is Director of Technologies for Greybrooke Consulting, a firm specializing in Oracle Applications implementation and upgrades with principal offices in Orlando, Atlanta, Chicago, and Tampa. Brian has a unique combination of business and technical expertise that is seldom seen in the ERP consulting arena. He has both a keen sense for business and a wealth of experience as a solutions architect, functional analyst, and technical lead.

His focus is on improving client ERP systems using a combination of technical and functional expertise in both product implementations and solution designs. He has demonstrated his functional expertise for over eleven years across a variety of Oracle Applications and business processes - Process & Discrete Manufacturing, Order-to-Cash, and Procure-to-Pay. In addition, his technical expertise spans practically all Oracle development tools and technologies.

More papers and presentations written by Brian are available at www.greybrooke.com.