WFi

Using XSL within WFi for Infor ERP System21

Copyright © 2012 Infor

All rights reserved. The word and design marks set forth herein are trademarks and/or registered trademarks of Infor and/or related affiliates and subsidiaries. All rights reserved. All other trademarks listed herein are the property of their respective owners.

Important Notices

The material contained in this publication (including any supplementary information) constitutes and contains confidential and proprietary information of Infor.

By gaining access to the attached, you acknowledge and agree that the material (including any modification, translation or adaptation of the material) and all copyright, trade secrets and all other right, title and interest therein, are the sole property of Infor and that you shall not gain right, title or interest in the material (including any modification, translation or adaptation of the material) by virtue of your review thereof other than the non-exclusive right to use the material solely in connection with and the furtherance of your license and use of software made available to your company from Infor pursuant to a separate agreement ("Purpose").

In addition, by accessing the enclosed material, you acknowledge and agree that you are required to maintain such material in strict confidence and that your use of such material is limited to the Purpose described above.

Although Infor has taken due care to ensure that the material included in this publication is accurate and complete, Infor cannot warrant that the information contained in this publication is complete, does not contain typographical or other errors, or will meet your specific requirements. As such, Infor does not assume and hereby disclaims all liability, consequential or otherwise, for any loss or damage to any person or entity which is caused by or relates to errors or omissions in this publication (including any supplementary information), whether such errors or omissions result from negligence, accident or any other cause.

Trademarks

Microsoft®, Windows®, Windows Server® and Internet Explorer® are registered trademarks of Microsoft Corporation.

IBM®, WebSphere®, MQSeries®, iSeries® are registered trademarks of IBM Corporation.

Oracle® and Java® are registered trademarks of Oracle and/or its affiliates.

Publication Information

Release: WFi - Release 1.0 (Silver Copy) Revision 1.0

Publication Date: January 2012

Table of Contents

Chapter 1 Introduction 1-7
Welcome 1-8
What is XML? 1-8
What is XSL? 1-8
What is XSLT? 1-8
What is XPath? 1-8
How can I use them within WFi? 1-9

Chapter 2 Tutorials 2-10
Using an XML Activity to Send Formatted Information 2-11
Using an XML Activity to Alter Process Flow 2-22
Extending the Standard Manual Activity XML Format 2-26
Process Generated Documentation (PDF) 2-33
Why use PDF? 2-33
Converting a Manual Activity Message to PDF 2-33
Further Things to Try 2-44
Creating an RSS Feed From a Process 2-45
What is RSS? 2-45
Using RSS with WFi 2-45
Adding RSS to Order Fulfilment 2-45
Filtering Data Using Categories 2-53
Further Things to Try 2-54

Chapter 3 Performance Tips 3-56
Converting Manual Activities from XML to XSL 3-57
Saving & Retrieving Data within the Document Handler Session 3-58
Example 1: Using the dochandlerDataStore Object in a Process 3-59
Saving Data Permanently from Your Stylesheet 3-65
Example 2: Storing data permanently in a Process 3-66
Retrieving Multiple Data Fields in a Single SQL Call 3-68
Example 3: Using sqlqueryFunction to Retrieve Data Fields 3-69

Chapter 4 Using Event Data Within A Process 4-73
Overview 4-74
An Example of Using Event Data in a Process 4-76

Appendix A Finished Example A-81

Appendix B Error Handling B-87

Appendix C XSL-FO to PDF Extension Function C-91
convertFOToPDF C-95
Extension Function Definition C-95
Description C-95
Return Nodeset Definition C-95
Example C-96

Chapter 1 Introduction

1

Welcome

What is XML?

Extensible Mark-up Language (XML) is a specification developed by the W3C. XML is a pared-down version of Standard Generalised Mark-Up Language, designed especially for Web documents. It allows designers to create their own customised tags, enabling the definition, transmission, validation, and interpretation of data between applications and between organizations.

What is XSL?

Extensible Stylesheet Language (XSL) is a language for creating a stylesheet that describes how data sent over the Web using the Extensible Mark-up Language (XML) is to be presented to the user. XSL specifies the styling of an XML document by using XSLT to describe how the document is transformed into another XML document that uses the formatting vocabulary.

What is XSLT?

XSL Transformations (XSLT) is a language for transforming XML documents into other XML documents. XSLT is designed for use as part of XSL, which is a stylesheet language for XML. In addition to XSLT, XSL includes an XML vocabulary for specifying formatting. XSL specifies the styling of an XML document by using XSLT to describe how the document is transformed into another XML document that uses the formatting vocabulary.

What is XPath?

XML Path Language (XPath) is a language for addressing parts of an XML document, designed for use with XSLT. The language mainly consists of location paths and expressions.

How can I use them within WFi?

Good question! Read the rest of this document and find out…

Note: This guide is not meant as a guide to XML, XSL, XSLT or XPath and assumes the reader has basic proficiency in all these technologies as well as XHTML.

For more information, tutorials and examples see the W3C Schools web site (http://www.w3schools.com/), an excellent resource for beginners and ideal reference site for more advanced users.

Chapter 2 Tutorials

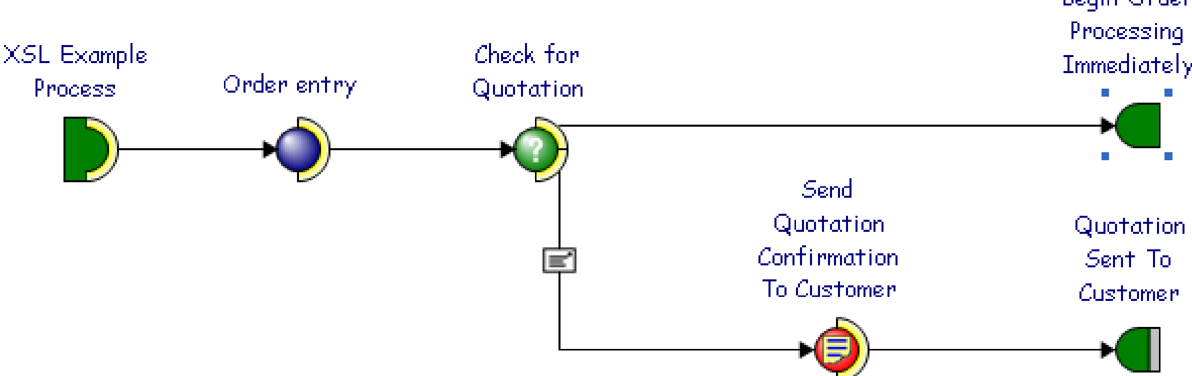
2

Using an XML Activity to Send Formatted Information

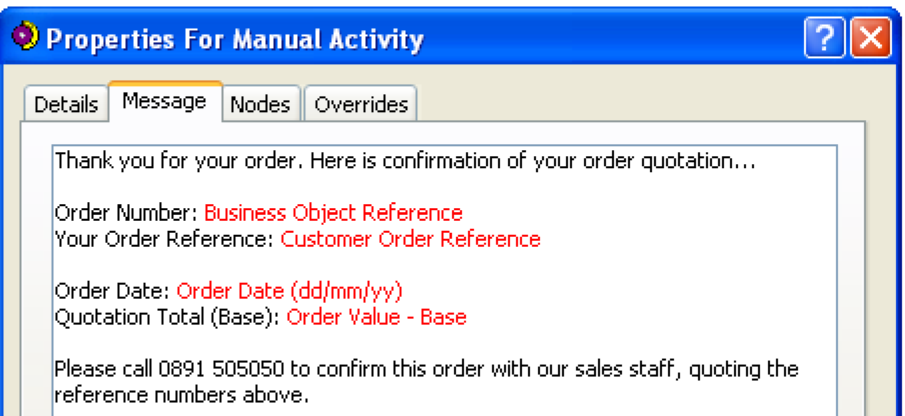
For this tutorial we are going to use the System21 Aurora Order Processing activities and data to show how to extract data from your IBM i server, using JDBC and SQL.

Note: This tutorial assumes that you have a correctly configured System21 Aurora server, WFi Engine, user profiles, WFi Components installation and WFi Modeler. Please refer to the either the provided System21 Aurora documentation or the WFi Installation and User guides for more information on specific features of these products.

To start with, here is our Business Process. The customer places an order, via phone to an operator. If the order is a quotation then the customer will be sent an email (in HTML format) containing basic information about the order, otherwise the standard order processing will begin (not dealt with in this example). Here is the process model…

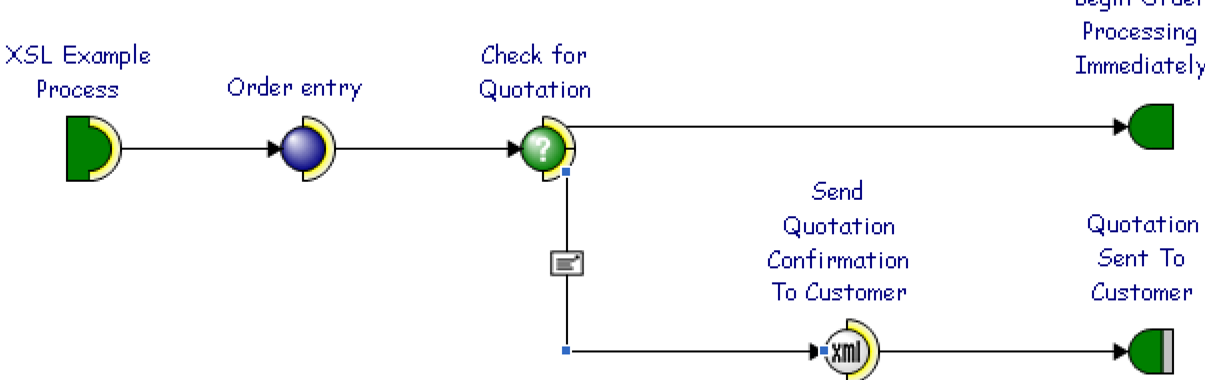


The Manual Activity message contains the following text…

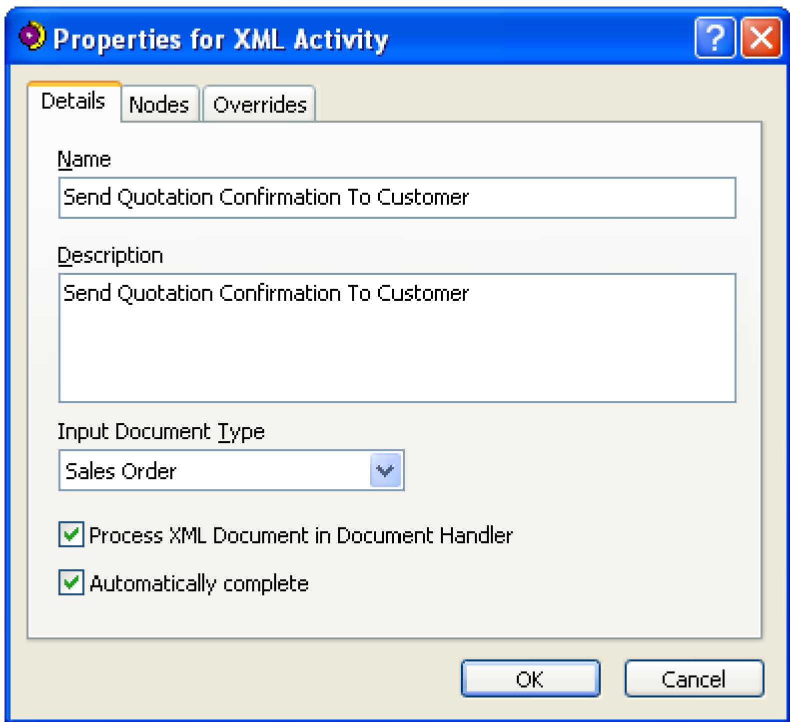


The message contains some basic order information but it does not convey the full detail of the order (items ordered, amount etc.). This information can be extracted using Data Fields but the format and presentation of the data is still somewhat limited (e.g. data cannot be tabularised).

To overcome these shortcomings we can replace the Manual Activity with an XML Activity and insert an XSL stylesheet that will create the HTML message output. Here is the process with the XML Activity inserted…



Set the XML Activity to be processed by the Document Handler…



Once created, the stylesheet can be imported into the model by selecting the *XML, Import* option from the XML Activities context menu or it can be created or edited within WFi Modeler using the inbuilt context sensitive editor.

In this tutorial we will build the XSL stylesheet up bit by bit to introduce the key features. To start with the stylesheet outline would look something like this…

```
<xsl:stylesheet version="1.0"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" media-type="text/html"/>

  <xsl:template match="process_activity_input_data">
  <HTML>
    <BODY>
       <H2 ALIGN="CENTER">Order Quotation</H2>
       <!-- Order Header -->
       <P><TABLE WIDTH="100%" BORDER="1">
```

```
            <TR>
              <TD WIDTH="50%">
                   <TABLE BORDER="0">
                     <TR><TD><B>Our Order Reference:</B></TD>
                          <TD><xsl:value-of select=
                                 "business_object_reference"/></TD></TR>
                     <TR><TD><B>Your Order Reference:</B></TD>
                          <TD></TD></TR>
                     <TR><TD><B>Order Date:</B></TD>
                          <TD></TD></TR>
                   </TABLE>
              </TD>
              <TD WIDTH="50%" VALIGN="TOP"><B>Delivery
                     Address:</B></TD>
            </TR>
           </TABLE></P>

<!-- Ordered Items -->
            <P><TABLE WIDTH="100%" BORDER="1">
                   <TR><TH ALIGN="LEFT">Item Number</TH>
                   <TH ALIGN="LEFT">Item Description</TH>
                   <TH ALIGN="CENTER">Item Quantity</TH>
                   <TH ALIGN="RIGHT">Cost</TH></TR>
                   <TR><TD></TD></TR>
            </TABLE></P>

<!-- Order Footer -->
            <P><TABLE WIDTH="100%" BORDER="0">
                   <TR><TD ALIGN="CENTER">Please call 0891 505050 to
confirm this order with our sales staff, quoting the reference
numbers above.<BR/><BR/><I>Thank you for your
business.</I></TD></TR>
                   </TABLE></P>
           </BODY>
```

```
    </HTML>
  </xsl:template>
</xsl:stylesheet>
```

If processed by the Document Handler at this time, the output would look something like this…

Order Quotation

Our Order Reference: 0000123 **Delivery Address:**

Your Order Reference:

Order Date:

Item Number	Item Description	Item Quantity	Cost

Please call 0891 505050 to confirm this order with our sales staff, quoting the reference numbers above.

Thank you for your business

The only piece of data we have is the order reference (the `business_object_reference` node, which is passed to us by the Document Handler as part of the XML documents supplied to every transformation. This is referenced directly using the `xsl:value-of` command which returns the string content of the node(S) referenced by the XPath in the `select` attribute.

One of the most difficult parts of coding XSL stylesheets is in understanding where you are within the XML document and how to reference child/parent nodes. In the stylesheet above, because we are matching on the Document Element (`process_activity_input_data`), we do not need to re-reference the Document Element to access any of its child nodes.

We now need to “fill in the blanks” by using JDBC and SQL to get the appropriate data. Let’s start with the header. The information for this order is held in a file called OEP40. The customer address details are in a file called SLP05. To retrieve the information we shall use the `sqlqueryFunction` that allows the execution of SQL scripts against any JDBC data source.

To use the `sqlqueryFunction` we need to tell the stylesheet the Java package and class to load and define a reference (namespace) for it (in this case we’ve called it `sql`). This is done within the XSL stylesheet element at the top of the document…

```
<xsl:stylesheet version="1.0"
        xmlns:sql="com.geac.xtrane.extensions.SQLQuery"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

Rather than putting the order-header code in the main template we can create a new template, called `order-header`, and call it using the code…

```
<xsl:call-template name="order-header"/>
```

As well as matching on XML documents Templates can be used as equivalents to subroutines or functions in other languages in that you can call them, pass parameters and return values from them (though that’s a little more complicated).

> **Tip**: If you write your templates to be independently called you can put them into their own file and import them into the master stylesheet. This makes them easily re-usable.

Here is the code for our `order-header` template…

```
<xsl:template name="order-header">

  <xsl:variable name="sqlOEP40">

       select CUSN40, DSEQ40, DTSO40, CUSO40 from OEP40

       where CONO40='<xsl:value-of select="company_code"/>'

      and ORDN40=

       '<xsl:value-of select= "business_object_reference"/>'
```

```
</xsl:variable>

<xsl:variable name="resultOEP40"
      select="sql:sqlqueryFunction(database_pool,
      environment_code, normalize-space($sqlOEP40), 1)"/>

<P><TABLE WIDTH="100%" BORDER="0">
<TR>
  <TD WIDTH="50%" VALIGN="TOP">
    <TABLE BORDER="0">
    <TR>
      <TD><B>Our Order Reference:</B></TD>
      <TD><xsl:value-of
           select="business_object_reference"/></TD>
    </TR>
    <TR>
      <TD><B>Your Order Reference:</B></TD>
      <TD><xsl:value-of select="$resultOEP40/resultset/row/
           column[@name='CUSO40']/@value"/></TD>
    </TR>
    <TR>
      <TD><B>Order Date:</B></TD>
      <TD><xsl:value-of select="$resultOEP40/resultset/row/
           column[@name='DTSO40']/@value"/></TD>
    </TR>
    </TABLE>
  </TD>

  <xsl:variable name="sqlSLP05">select * from SLP05 where
     CUSN05='<xsl:value-of select="$resultOEP40/resultset/
     row/column[@name='CUSN40']/@value"/>' and
     DSEQ05='<xsl:value-of select="$resultOEP40/resultset/
     row/column[@name='DSEQ40']/@value"/>'</xsl:variable>
  <xsl:variable name="resultSLP05"
      select="sql:sqlqueryFunction(database_pool,
      environment_code, normalize-space($sqlSLP05), 1)"/>
```

```
    <xsl:for-each select="$resultSLP05/resultset/row">
      <TD WIDTH="50%" VALIGN="TOP">
         <B>Delivery Address:</B><BR/><BR/>
<xsl:value-of select="column[@name='CNAM05']/@value"/><BR/>
<xsl:value-of select="column[@name='CAD105']/@value"/><BR/>
<xsl:value-of select="column[@name='CAD205']/@value"/><BR/>
<xsl:value-of select="column[@name='CAD305']/@value"/><BR/>
<xsl:value-of select="column[@name='CAD405']/@value"/><BR/>
<xsl:value-of select="column[@name='CAD505']/@value"/><BR/>
<xsl:value-of select="column[@name='PCD105']/@value"/>
<xsl:value-of select="column[@name='PCD205']/@value"/><BR/>
      </TD>
    </xsl:for-each>
  </TR>
  </TABLE></P>
</xsl:template>
```

Note: Some of the rows above have been wrapped to fit within this documents formatting.

The key points to note in the order header template are…

- The SQL is constructed within a variable using a mixture of text and XML variables. This variable is then passed to the `sqlqueryFunction`. We use the normalize-space function so that any unwanted tabs or new line characters are removed.

- The `sqlqueryFunction` takes four parameters:

    - The first is the name of the JDBC pool (defined in the `DBConnectionManager.properties` file in your WFi Components installation). We use the pool that is passed in by the Document Handler.

  - The second is the WFi environment code that is defined as being WFi enabled. We use the code that is passed in by the Document Handler.

  - The third is the SQL statement to execute. We pass our previously created variable. In XSL the dollar symbol denotes a variable.

  - The number of rows to return. In this example there should only be one row. Specify –1 to return all rows that match the passed SQL criteria.

- The XML node set returned by the `sqlqueryFunction` has a Document Element of `resultset`, which has child nodes called `row`, which has child nodes called `column`. The `column` node has an attribute called `name` containing the table's column name and an attribute called `value` containing the columns data.

- The values from a previous SQL query can be used to construct others.

- The XSL for-each statement is used purely for abbreviation. As there is only one row it will only loop once and it saves referencing the full XPath for the data we require.

If processed by the Document Handler at this time, the output would look something like this…

Order Quotation

Our Order Reference:	0000607	**Delivery Address:**
Your Order Reference:	XA65101	Reeves Chemist's 31 Lenton Boulevard Lenton Nottingham England NG7 4QP
Order Date:	1040625	

Item Number	Item Description	Item Quantity	Cost

Please call 0891 505050 to confirm this order with our sales staff, quoting the reference numbers above.

Thank you for your business.

Note: The date is its raw System21 Aurora format.

Creating a template for the items is similar to the header. The order lines are held in the file OEP55 and are indexed by order number (ORDN55) and company (CONO55). The item number is held in field CATN55, the quantity is held in the field QTOR55 and the order line value is held in ORLV55. You'll need to use a XSL for-each loop to traverse each item. The item description is held in INP35 in a field called PDES35 (the field PNUM35 should match the value in CATN55). There will only be one item description per item number.

Try and create the order-lines template for yourself!

After you have written the XSL and passed it through the Document Handler it should look something like this…

Order Quotation

Our Order Reference: 0000607

Your Order Reference: XA65101

Order Date: 1040625

Delivery Address:

Reeves Chemist's
31 Lenton Boulevard
Lenton
Nottingham
England

NG7

Item Number	Item Description	Item Quantity	Cost
5001	Sun cream Factor 4 225ml	1.000	100.00
5002	Sun cream Factor 8 225ml	3.000	23.00
5003	Total Sun Block Cream 175ml	12.000	89.00

Please call 0891 505050 to confirm this order with our sales staff, quoting the reference numbers above.

Thank you for your business.

Note: The full, commented, stylesheet can be found in Appendix A.

Some enhancements you can try are…

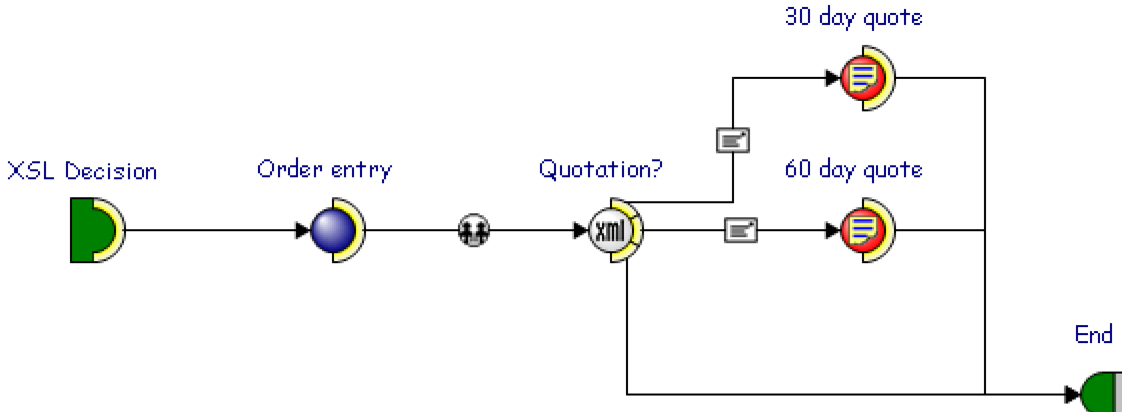
- Use the XSL `sum` function to add up all the costs to give a total.
- Try using the `sub-string` and `string-length` functions in XSL to format it correctly in dd/mm/yy format.

Using an XML Activity to Alter Process Flow

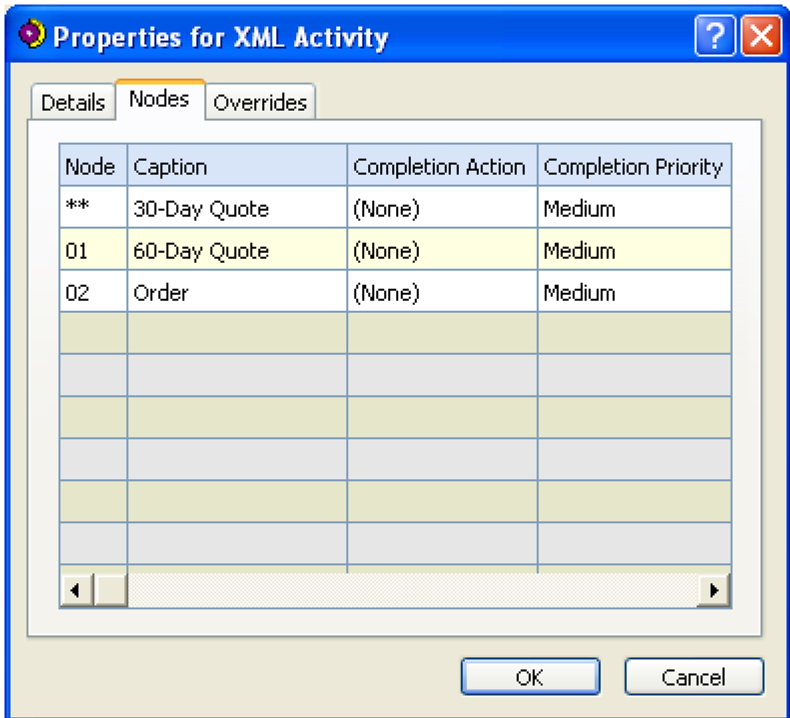
Using the WFi `setActivityStatus` extension function, the exit node, on completion of the XML Activity, can be selected, allowing the XSL stylesheet to control process flow.

For this tutorial we are going to use the Infor ERP System21 Aurora Order Processing activities and data to show how to use extracted data from your IBM i server to alter the process flow.

To start with, here is our Business Process. The customer places an order, via phone to an operator. Instead of using an Action Agent to make a decision within the process we are going to use the stylesheet within the XML Activity to decide what to do. Here is the business process…



In the XML Activity we define three exit nodes like so…



The stylesheet will use the `sqlqueryFunction` to get the order type and then use `setActivityStatus` to select the correct exit path. Here is the stylesheet to import into the XML Activity…

```
<xsl:stylesheet  version="1.0"
    xmlns:sql="com.geac.xtrane.extensions.SQLQuery"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:process-activity="com.geac.process.extensions.Activity">

      <xsl:output method="xml" media-type="text/xml"/>

      <xsl:template match="process_activity_input_data">
            <xsl:variable name="sqlOEP40">
                  select QTNO40 from OEP40
                  where CONO40='<xsl:value-of
                  select="company_code"/>'
                  and ORDN40='<xsl:value-of
                  select="business_object_reference"/>'
            </xsl:variable>
<!--
      Use the same Document Handler database pool and environment
-->
            <xsl:variable name="resultOEP40" select="
                  sql:sqlqueryFunction(database_pool,
```

```
                    environment_code, normalize-space($sqlOEP40),
                    1)"/>

            <results_from_setActivityStatus>
<!--
        Before doing a complete the activity must be released from
        pre-pending state
-->
            <xsl:copy-of select="process-activity:

                    setActivityStatus(database_pool,

                    environment_code,'release',

                    multi_thread_identifier,

                    activity_number, recipient_name, '')"/>
            <xsl:choose>
<!--
        30 day quote (QTNO40 = A)
-->
             <xsl:when test="$resultOEP40/resultset/row/column
                            [@name='QTNO40']/@value = 'A'">
               <xsl:copy-of select="process-activity:
                    setActivityStatus(database_pool,
                    environment_code, 'complete',
                    multi_thread_identifier,
                    activity_number, recipient_name,
                    '**')"/>
             </xsl:when>
<!--
        60 day quote (QTNO40 = B)
-->
             <xsl:when test="$resultOEP40/resultset/row/column
                          [@name='QTNO40']/@value = 'B'">
               <xsl:copy-of select="process-activity:
                          setActivityStatus(database_pool,
                          environment_code, 'complete',
                          multi_thread_identifier,
                          activity_number, recipient_name,
                          '01')"/>
             </xsl:when>
<!--
        Any other order type

-->
             <xsl:otherwise>
               <xsl:copy-of select="process-activity:
```

```
                        setActivityStatus(database_pool,
                        environment_code, 'complete',
                        multi_thread_identifier,
                        activity_number, recipient_name,
                        '02')"/>
            </xsl:otherwise>
        </xsl:choose>

  </results_from_setActivityStatus>

 </xsl:template>

</xsl:stylesheet>
```

After extracting the quotation type we use the `xsl:choose` construct to decide the exit route. The completion code passed by the `setActivityStatus` function matches the ones defined in the process model.

To finish, in the example process, we send the named customer contact an email, when a quote is created, reminding them of the order and providing the expiry date.

> **Tip**: In this example we use Manual Activities to send a message to the customer but you could try converting these to suitable XML Activities as we did in the first example in this chapter.

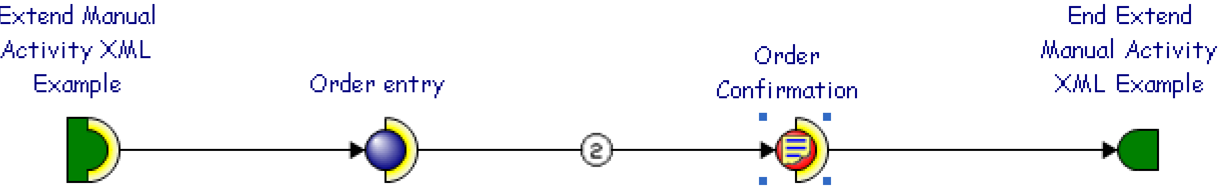
> **Tip**: The stylesheet above shows the most basic of tests to decide which route to take. To take advantage of this form of decision-making the tests can be made over several fields in the same table or even over multiple tables. As we are using JDBC, decisions can also be made using non-IBM i server data sources such as Microsoft Access or Oracle.

Extending the Standard Manual Activity XML Format

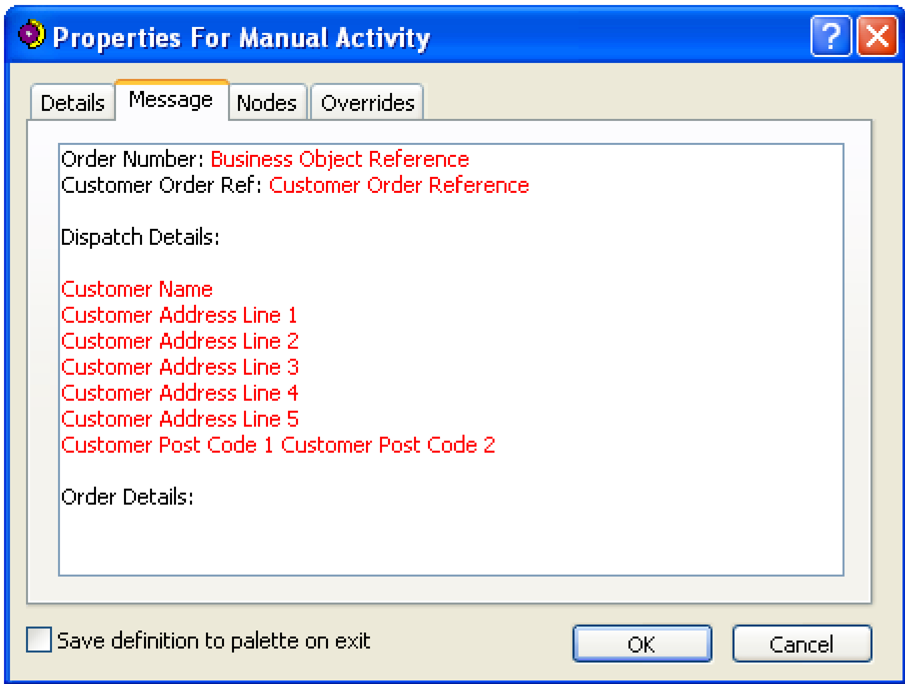
The Manual Activity format supports a fixed XML document definition that is translated into a GUI display by either stylesheets within Infor System i Workspace or the Email Writer. Using XML Activities we can easily extend this XML document format to add new elements.

For this tutorial we are going to add a table element to the document to store information returned using SQL.

First we need to create a process. For this example we will send a HTML email and a Manual Activity message to an Order Clerk role when an order has been placed in System21. The initial process should look like the following…



The Manual Activity should be setup to include a basic order confirmation message like the one defined below…



We will need to insert our new table element under the “Order Details:” caption within the message. First, we need to convert our Manual Activity to XML using the *XML, Convert To XSL* option off the Manual Activity element’s context menu. Once this is done the XML can be altered.

The order details can be read from the OEP55 using INP35 for the item details with the following SQL statement…

```
select ORDL55, PDES35, EQTY55, EUOM55, ORLV55
from OEP55, INP35
where CONO35 = {company code} and
      CONO55 = {company code} and
      ORDN55 = {order reference} and
      CONO55 = CONO35 and
      CATN55 = PNUM35
```

This will pull back the order line (ORDL55), item description (PDES35), quantity (EQTY55), unit or measurement (EUOM55) and line value (ORLV55) with which we will organise into a table with header, row and column elements.

The extensions to the Manual Activity XML document to support a table element will require the following changes to the `Manual Activity.DTD` file.

```
<!ELEMENT  BODY  (#PCDATA  |  DATA_REF  |  LINK_DATA  |  NEW_LINE  |
CHECK_BOX | RADIO_BUTTON_GROUP | LIST | INPUT_TEXT | TABLE)*>

<!ELEMENT TABLE (HEADER | ROW)*>
<!ELEMENT HEADER (COLUMN)*>
<!ELEMENT ROW (COLUMN)*>
<!ELEMENT COLUMN (#PCDATA) *>
```

Here is the extract of XSL that will retrieve the data from the IBM i server and format it…

```
<ma:NEW_LINE/>
Order Details:
<xsl:variable name="oep55Sql">
       select ordl55, pdes35, eqty55, euom55, orlv55
       from osld1f3.oep55, osld1f3.inp35
       where cono35 = '<xsl:value-of select="company_code"/>' and
       cono55 = '<xsl:value-of select="company_code"/>' and
       ordn55 = '<xsl:value-of select=
                     "business_object_reference"/>' and
       cono55 = cono35 and
       catn55 = pnum35
</xsl:variable>
<xsl:variable name="oep55Data"
 select="sql:sqlqueryFunction(database_pool,
         environment_code, normalize-space($oep55Sql), -1)"/>
<ma:TABLE>
  <ma:HEADER>
       <ma:COLUMN>Line</ma:COLUMN>
       <ma:COLUMN>Item</ma:COLUMN>
       <ma:COLUMN>Quantity</ma:COLUMN>
       <ma:COLUMN>Unit</ma:COLUMN>
       <ma:COLUMN>Value</ma:COLUMN>
  </ma:HEADER>
  <xsl:for-each select="$oep55Data/resultset/row">
       <ma:ROW>
           <ma:COLUMN><xsl:value-of
           select="column[@name='ORDL55']/@value"/> </ma:COLUMN>
           <ma:COLUMN><xsl:value-of
           select="column[@name='PDES35']/@value"/> </ma:COLUMN>
           <ma:COLUMN><xsl:value-of
```

```
            select="column[@name='EQTY55']/@value"/> </ma:COLUMN>
            <ma:COLUMN>
            <xsl:value-of select="column[@name='EUOM55']/@value"/>
            </ma:COLUMN>
            <ma:COLUMN>
            <xsl:value-of select="column[@name='ORLV55']/@value"/>
            </ma:COLUMN>
        </ma:ROW>
  </xsl:for-each>
</ma:TABLE>
```

Once the data has been retrieved using the SQL query function we simply need to extract the relevant column data by looping over all the row elements using the `xsl:for-each` element.

Insert this extract into the appropriate place within the stylesheet (after the “Order Details” caption).

> **Note**: Make sure to add the sql extension function namespace to the stylesheet as shown in the previous example.

Here is how the table element section of the Manual Activity XML might look after it has been processed within the Document Handler…

```
<ma:NEW_LINE/>
Order Details:
<ma:TABLE>
        <ma:HEADER>
                <ma:COLUMN>Line</ma:COLUMN>
                <ma:COLUMN>Item</ma:COLUMN>
                <ma:COLUMN>Quantity</ma:COLUMN>
                <ma:COLUMN>Unit</ma:COLUMN>
                <ma:COLUMN>Value</ma:COLUMN>
        </ma:HEADER>
        <ma:ROW>
```

```
                <ma:COLUMN>1</ma:COLUMN>
                <ma:COLUMN>Suncream Factor 4 225ml</ma:COLUMN>
                <ma:COLUMN>1.000</ma:COLUMN>
                <ma:COLUMN>BX</ma:COLUMN>
                <ma:COLUMN>10.00</ma:COLUMN>
            </ma:ROW>
            <ma:ROW>
                <ma:COLUMN>2</ma:COLUMN>
                <ma:COLUMN>Suncream Factor 8 225ml</ma:COLUMN>
                <ma:COLUMN>2.000</ma:COLUMN>
                <ma:COLUMN>BX</ma:COLUMN>
                <ma:COLUMN>20.00</ma:COLUMN>
            </ma:ROW>
            <ma:ROW>
                <ma:COLUMN>3</ma:COLUMN>
                <ma:COLUMN>Total Sun Block Cream
            175ml</ma:COLUMN>
                <ma:COLUMN>3.000</ma:COLUMN>
                <ma:COLUMN>BX</ma:COLUMN>
                <ma:COLUMN>30.00</ma:COLUMN>
            </ma:ROW>
        </ma:TABLE>
```

We now need to alter the stylesheets that display the data to correctly display the table using HTML. We will start with the `2 Way Email Stylesheet HTML.xsl` file that is shipped by default with your WFi Components installation.

Add the following XSL templates to process the new table element…

```
<!--
     TABLE template - Insert a table into the HTML output
-->
<xsl:template match="ma:TABLE">
     <table border="1">
```

```
            <xsl:apply-templates/>
        </table>
</xsl:template>

<!--
        HEADER template - Insert new header row
-->
<xsl:template match="ma:HEADER">
        <tr><xsl:apply-templates mode="header"/></tr>
</xsl:template>

<!--
        ROW template - Insert new row
-->
<xsl:template match="ma:ROW">
        <tr><xsl:apply-templates mode="row"/></tr>
</xsl:template>

<!--
        COLUMN template - Put columns in header mode in bold
-->
<xsl:template match="ma:COLUMN" mode="header">
        <th><b><xsl:apply-templates/></b></th>
</xsl:template>


<!--
        COLUMN template - Put columns in row mode in plain text
-->
<xsl:template match="ma:COLUMN" mode="row">
        <td><xsl:apply-templates/></td>
</xsl:template>
```

We break each part of the table element into its separate part and output the appropriate HTML table element in its place. This leads to a table similar to the following…

Order Details:

Line	Item	Quantity	Unit	Value
1	Sun cream Factor 4 225ml	1.000	BX	10.00
2	Sun cream Factor 8 225ml	2.000	BX	20.00
3	Total Sun Block Cream 175ml	3.000	BX	30.00

In System i Workspace the `format-activity.xsl` stylesheet is used to convert the Manual Activity XML into HTML. Use the above templates within this file to allow table elements to be shown within System i Workspace.

> **Tip**: You will need to match against the mode HTML for the TABLE element and pass this through via the `xsl:apply-templates` element to the HEADER and ROW elements.

Once the basic table is in place you can try adding the following...

- Detect an SQL error and display a suitable message
- Don't show the table header if no rows were retrieved
- Add an alignment attribute to say which way the column is aligned (left, centre, right)
- Add a total to the bottom of the table (use the XSL `sum` function)

Process Generated Documentation (PDF)

There are numerous 3rd party conversion packages to take XML output and change it into a different format. One example of this is FOP from the Apache Software Foundation. FOP (Formatting Objects Processor) is the world's first print formatter driven by XSL formatting objects (XSL-FO) and the world's first output independent formatter. It is a Java application that reads a formatting object tree and renders the resulting pages to a specified output. Output formats currently supported include PDF, PCL, PS, SVG, XML (area tree representation), Print, AWT, MIF and TXT. The primary output target is PDF.

FOP is part of the Open Source Project and is therefore free to use by third-party developers. For more information of FOP see http://xmlgraphics.apache.org/fop/.

For this tutorial we will look at using FOP within a process to generate Adobe PDF format documents.

Why use PDF?

PDF is rapidly becoming the de-facto format for transferring documents across the Internet. Many businesses are using PDF to store key documents for auditing purposes (including Sarbanes & Oxley Act compliance). The Adobe Acrobat reader software is free to use and the PDF format provides a compact and portable method for storing information. PDF documents can be encoded/encrypted/digitally signed for security. For more on Adobe PDF see the web site at http://www.adobe.co.uk/products/acrobat/main.html.

Converting a Manual Activity Message to PDF

For our example we are going to convert a standard Manual Activity message into a PDF document that can be emailed, as an attachment, to any desired recipient.

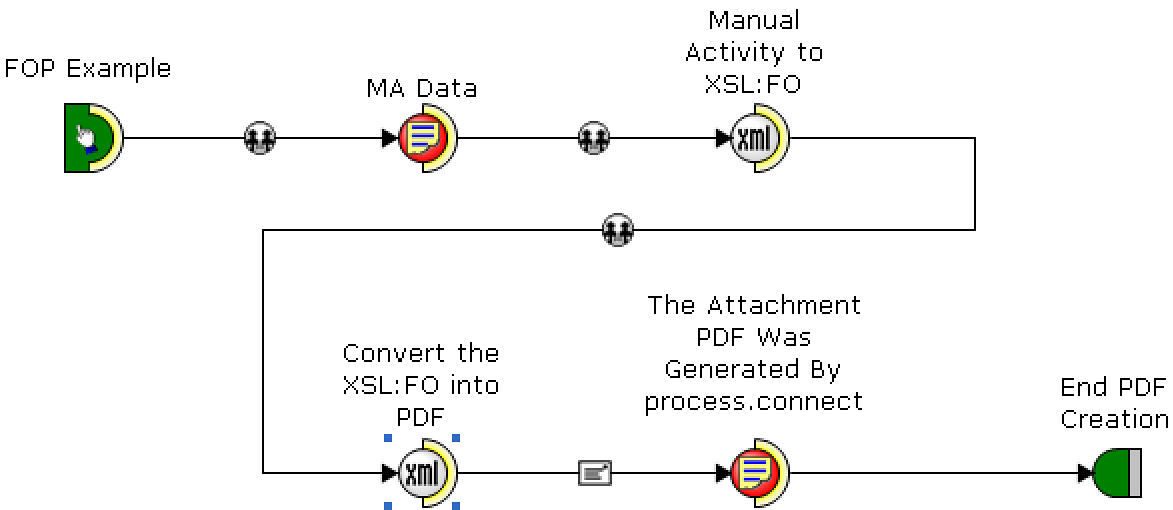
Normally, Manual Activity messages are sent to recipients as HTML (rendered by System i Workspace or by the WFi Email Writer). To convert a Manual Activity into a PDF file we need to carry out three distinct actions…

- Generate the Manual Activity message data (XML)

- Convert the Manual Activity message data to formatted data (XSL-FO)
- Convert the formatted data into PDF

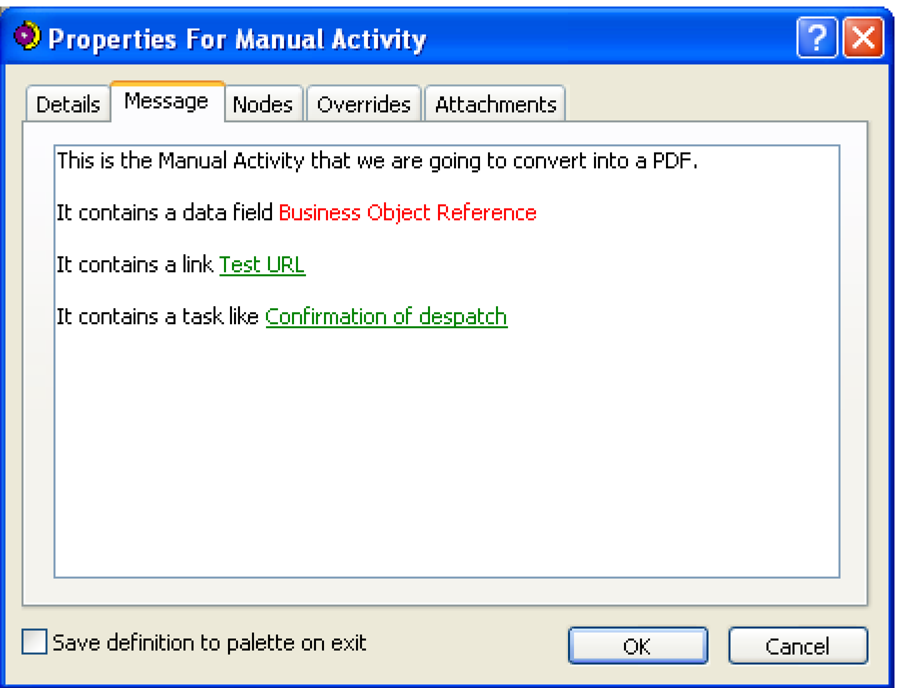
Once we have generated the PDF file, we can then send it to a recipient as an attachment.

The process to carry out this conversion may look something like this…



Tip: For this tutorial we have broken the steps into distinct activities for clarity. The four activities above could easily be combined into a single step.

The first activity contains a simple message that we want to convert. By setting the activity to auto-complete, we create the message XML within the WFi files for access later in the process.



The next activity takes the XML data and converts it to the XSL-FO format. We read in the message data using the `readActivityDocument` extension function and then apply a transformation to convert to the XSL-FO document format.

> **Note**: For this tutorial we will not be covering the details of XSL-FO. For more information see http://www.w3schools.com/xslfo/xslfo_intro.asp and http://xmlgraphics.apache.org/fop/fo.html.

Our XSL-FO document simply has a single page that contains a logo at the top, a heading and the message text (any buttons are ignored). Here is the stylesheet…

```
<xsl:stylesheet version="1.0"
     xmlns:process-activity="com.geac.process.extensions.Activity"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     xmlns:fo="http://www.w3.org/1999/XSL/Format"
     xmlns:ma="http://www.geac.com/manact"
     xmlns:du="com.geac.xtrane.extensions.DomUtils">

<xsl:output method="xml" media-type="text/xml"/>

<!-- Hard coded links to the Workspace server -->
```

```
<xsl:variable name="workSpaceServer"
 select="'http://workspace.com'"/>
<xsl:variable name="workSpaceProfile" select="'PROFILE'"/>
<xsl:variable name="roleCode" select="'ROLE'"/>

<xsl:template match="process_activity_input_data">
<!--
       Read in the manual activity data into a variable and
       transform the content.
-->
       <xsl:variable name="manactData"
          select="process-activity:readActivityDocument(
             database_pool, environment_code,
             multi_thread_identifier,
             string(number(activity_number) - 1))"/>

<xsl:apply-templates select="$manactData/read_activity_document/
         activity_document/."/>
</xsl:template>

<xsl:template match="ma:MANUAL_ACTIVITY">
  <fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
    <!-- Define the page layout -->
    <fo:layout-master-set>
     <fo:simple-page-master master-name="rest"
 page-height="29.7cm" page-width="21cm" margin-top="1cm"
 margin-bottom="2cm" margin-left="2.5cm"
        margin-right="2.5cm">
       <fo:region-body margin-top="2.5cm" margin-bottom="1.5cm"/>
       <fo:region-before extent="2.5cm"/>
       <fo:region-after extent="1.5cm"/>
    </fo:simple-page-master>

    <fo:page-sequence-master master-name="basicPSM" >
     <fo:repeatable-page-master-alternatives>
      <fo:conditional-page-master-reference
       master-reference="rest" page-position="rest" />
      <fo:conditional-page-master-reference
       master-reference="rest" />
     </fo:repeatable-page-master-alternatives>
    </fo:page-sequence-master>

    </fo:layout-master-set>

    <!-- Page 1 -->
```

```
    <fo:page-sequence master-reference="basicPSM">
<!--
        header
-->
        <fo:static-content flow-name="xsl-region-before">
          <fo:block text-align="center" font-size="10pt"
    font-family="serif" line-height="14pt" >
                <xsl:apply-templates
                       select="./ma:TITLE[namespace-uri()
                       ='http://www.geac.com/manact']"/>
         </fo:block>
        </fo:static-content>

        <!-- footer -->
        <fo:static-content flow-name="xsl-region-after">
                <fo:block text-align="center"
                          font-size="10pt" font-family="serif"
                          line-height="14pt" >
                       | Page <fo:page-number/> |
                </fo:block>
        </fo:static-content>

        <!-- body -->
        <fo:flow flow-name="xsl-region-body">
                <!-- Title block -->
                <fo:block font-size="18pt"
                           font-family="sans-serif"
                           line-height="24pt"
                           space-after.optimum="15pt"
                           background-color="blue"
                           color="white"
                           text-align="center"
                           padding-top="3pt"
                           font-variant="small-caps"
                           space-before.minimum="1em"
                           space-before.optimum="1.5em"
                           space-before.maximum="2em">

                       <xsl:apply-templates
                              select="./ma:TITLE[namespace-uri() =
                              'http://www.geac.com/manact']"/>
                </fo:block>

                <!-- Body block -->
                <fo:block font-size="12pt"
                           font-family="sans-serif"
                           line-height="15pt"
                           space-after.optimum="3pt"
```

```
                        text-align="start">
                    <xsl:apply-templates
                         select=".//ma:BODY[namespace-uri() =
                         'http://www.geac.com/manact']" />
               </fo:block>
          </fo:flow>
     </fo:page-sequence>
</fo:root>

</xsl:template>

<xsl:template match="ma:TITLE">
       <xsl:apply-templates/>
</xsl:template>

<!-- Convert links into URL's to the WorkSpace server -->
<xsl:template match="ma:LINK_DATA">

       <xsl:choose>
              <xsl:when test="@SHORTCUT_TYPE='1'">
                     <xsl:variable name="url">

                     <xsl:value-of select="$workSpaceServer"/>

                     /server/safe-launch.xsp?profile-id=

                     <xsl:value-of select="du:escape

                     ($workSpaceProfile)"/>

                     </xsl:variable>
                     <fo:inline font-family="serif">
                       <fo:basic-link color="blue">
<xsl:attribute name="external-destination"><xsl:value-of
select="$url"/></xsl:attribute>
                            <xsl:value-of select="text()" />
                       </fo:basic-link>
                     </fo:inline>
              </xsl:when>
              <xsl:when test="@SHORTCUT_TYPE='3'">
                     <xsl:variable name="sub-url">
                            <xsl:choose>
                                   <xsl:when test="@DATA_REF">
              <xsl:variable name="dfname" select="@DATA_REF" />
              <xsl:value-of select="//ma:MANUAL_ACTIVITY/
                 ma:META_DATA/ma:DATA_FIELD_DATA[@NAME=$dfname]"/>
                                   </xsl:when>
                                   <xsl:otherwise>
                            <xsl:value-of select="@SHORTCUT_DATA"/>
                                   </xsl:otherwise>
```

```
                </xsl:choose>
            <xsl:if test="./ma:FIELD_PARAMETER">?</xsl:if>
            </xsl:variable>
            <xsl:variable name="params">

              <xsl:apply-templates mode="param"/>

            </xsl:variable>
            <xsl:variable name="url">

      <xsl:value-of select="normalize-space($sub-url)"/>

      <xsl:value-of select="normalize-space($params)"/>

            </xsl:variable>
            <fo:inline font-family="serif">
              <fo:basic-link color="blue">
            <xsl:attribute name="external-destination">

                <xsl:value-of select="$url"/>

            </xsl:attribute>
            <xsl:value-of select="text()" />
              </fo:basic-link>
            </fo:inline>
      </xsl:when>
      <xsl:otherwise>
            <fo:inline font-family="serif">
              <fo:basic-link color="blue">
            <xsl:attribute name="external-destination">

              <xsl:value-of select="@SHORTCUT_DATA" />

            </xsl:attribute>
            <xsl:value-of select="text()" />
              </fo:basic-link>
            </fo:inline>
      </xsl:otherwise>
    </xsl:choose>
</xsl:template>

<!--
    Show data fields as bold text

-->

<xsl:template match="ma:DATA_REF">
    <xsl:variable name="dfname" select="@NAME" />
    <fo:inline font-weight="bold">
    <xsl:value-of select="//ma:MANUAL_ACTIVITY
/ma:META_DATA/ma:DATA_FIELD_DATA[@NAME=$dfname]"/>

    </fo:inline>

</xsl:template>
```

```
<xsl:template match="ma:FIELD_PARAMETER" mode="param">
  <xsl:variable name="param">
    <xsl:apply-templates/>
  </xsl:variable>
  <xsl:value-of select="@NAME"/>=
  <xsl:value-of select="du:escape(normalize-space($param))"/>
  <xsl:if test="following-sibling::ma:FIELD_PARAMETER">&amp;
  </xsl:if>
</xsl:template>

<xsl:template match="text()" mode="param"></xsl:template>

<xsl:template match="ma:NEW_LINE"><fo:block/></xsl:template>

</xsl:stylesheet>
```

The comments describe what each section in the stylesheet does. At the top of the document we define links to a System i Workspace server. This allows us to embed links that the user can click on to launch Infor ERP System21 Aurora tasks from the PDF document. At the top of the first page we define a link to a bitmap (you will need to set this to a valid bitmap on your system).

The output is now in XSL-FO format and will be stored within WFi. The next activity will take XSL-FO data and convert it to a PDF file using Apache FOP.

> **Note**: For this stage we need the Apache FOP distribution. This can be found at http://www.apache.org/dyn/closer.cgi/xmlgraphics/fop. For this example we have used version 0.94. Download and install the FOP directory structure onto your system.

By default, FOP does not have an interface for use in a stylesheet but it is quite easy to wrapper the FOP conversion function using Java. There is an example Java class to do this in Appendix C.

The class contains a XSL extension function called `convertFOToPDF` that will take an XSL-FO file and convert it to PDF file. The function takes two string arguments (input file and output file) and returns anything a nodeset similar to the ones produced by the standard WFi extension functions. We put this into the standard process extensions package and use this in our next activity.

Note: You will need to compile the Java source into a class file. Use the IBM SDK (1.5 or later) or some other compliant Java compiler or development environment (such as IBM Rational Application Developer). You will need to link against the `fop.jar` file from the Apache FOP distribution to resolve the import references.

Here is the stylesheet to convert the XSL-FO into a PDF file…

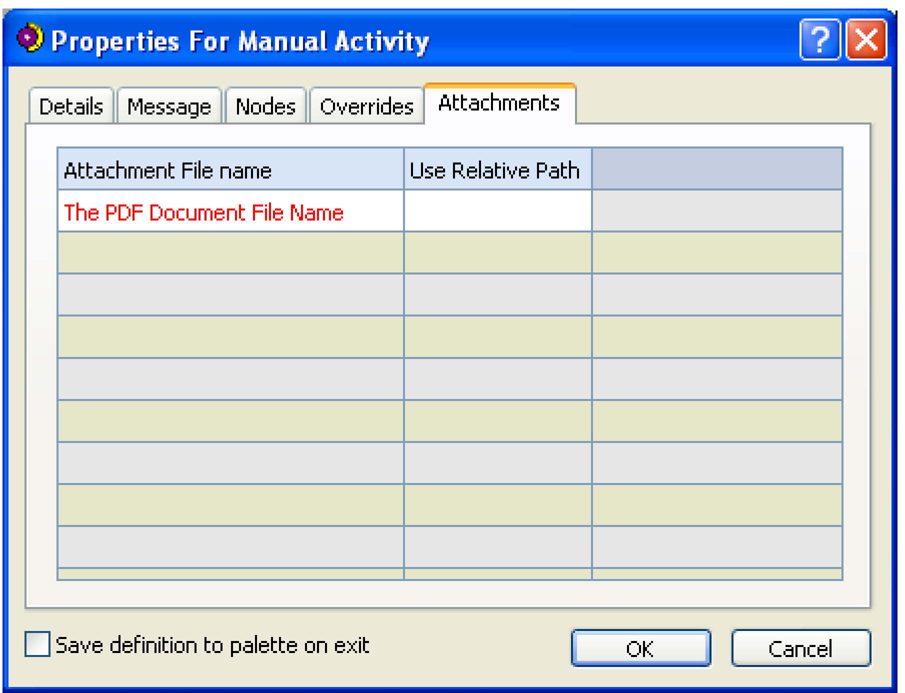
```
<xsl:stylesheet version="1.0"
     xmlns:process-activity="com.geac.process.extensions.Activity"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     xmlns:fop="com.geac.process.extensions.FOP"
     xmlns:redirect="org.apache.xalan.lib.Redirect"
     extension-element-prefixes="redirect">

 <xsl:output method="xml" media-type="text/xml"/>

 <xsl:template match="process_activity_input_data">

<!--
       Read the document into a variable
-->
       <xsl:variable name="xmlFO" select= "process-activity:
              readActivityDocument(database_pool, environment_code,
              multi_thread_identifier,
              string(number(activity_number)-1))"/>

<!--
       Create the XSL-FO input filename
-->
       <xsl:variable name="xmlFOFile">c:\temp\pdfs\<xsl:value-of
select="business_object_reference"/>.xml</xsl:variable>

<!--
       Save the input document to a file
-->
       <redirect:write select="$xmlFOFile">
              <xsl:copy-of select="$xmlFO/read_activity_document
 /activity_document/*"/>
       </redirect:write>

<!--
       Create the PDF file name
-->
       <xsl:variable name="pdfFile">c:\temp\pdfs\<xsl:value-of
           select="business_object_reference"/>.pdf</xsl:variable>
```

```
<!--
      Call the conversion routine
-->
      <xsl:variable name="pdfOut"
            select="fop:convertFOToPDF($xmlFOFile, $pdfFile)"/>


 </xsl:template>

</xsl:stylesheet>
```

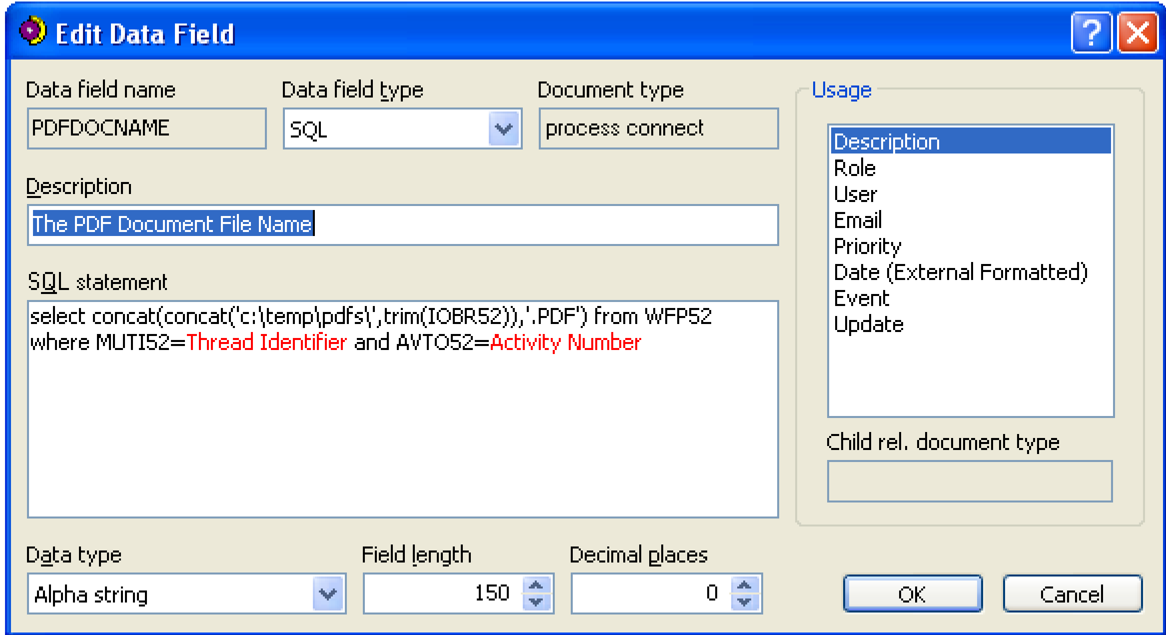
The stylesheet reads the content of the XSL-FO transformation into a variable (`xmlFO`) and saves this to a file. The filename is made up of a fixed path (`c:\temp\pdfs\`) the business object reference and the `.XML` extension. We use the Apache Xalan Redirect extension element to create the XSL-FO file. We then create a filename for the output PDF document and pass these two filenames into the `convertToPDF` function.

> **Note**: When processing the activity within the Document Handler you will need to add the class file that you created for the `ConvertFOToPDF` extension function to the class path. You will also need to add the `batik-all-1.6.jar`, `avalon-framework-4.2.0.jar`, `commons-io-1.3.1.jar`, `commons-logging-1.0.4.jar`, `xmlgraphics-commons-1.2.jar` and `fop.jar` files from the Apache FOP installation.
>
> To do this, just add the new class file and fop JARs to the `classpath` property in the Document Handler properties file and the Document Handler will do the rest!

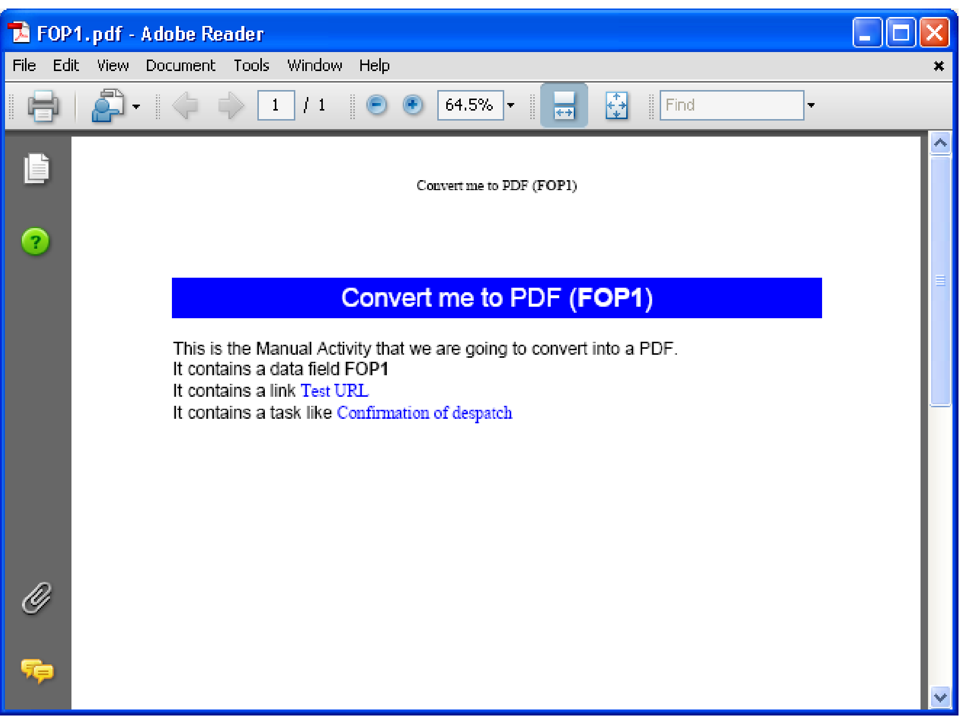
Once the PDF file is created, we can send it to our recipient. In the example we add the file as an attachment and send it via email.



The file name data field is generated into a data field using the business object reference.



The recipient should get a PDF document that looks similar to this…



Further Things to Try

This tutorial is only intended to whet your appetite as to the possibilities of process driven documentation. Here are some more things you might want to try…

- Combine the conversion to PDF into a single activity.
- Embed a table within the message (using the `sqlqueryFunction` to obtain data from Infor ERP System21 Aurora and the XSL-FO table formatting elements).
- Remove the intermediate XML file after it has been converted.
- Extend the Java function to return success/failure return document.
- Allow the user to return the document into the system with amendments.
- Try converting the sales invoice from tutorial 1 into a PDF file.

Creating an RSS Feed From a Process

What is RSS?

Really Simple Syndication (RSS) is a lightweight XML format designed for sharing headlines and other Web content. Think of it as a distributable "What's New" for web sites. Originated by UserLand in 1997 and subsequently used by Netscape to fill channels for Netcenter, RSS has evolved into a popular means of sharing content between sites.

The web site creates a RSS format file on their web-site that contains a list of items, or topics, each with a headline and a description. The user then uses a RSS news reader (also known as Aggregators) to point to the file and notify the user when updates occur. RSS feeds can also be added to various registries that poll the server file and then broadcast the occurrence of any updates.

> **Tip**: The BBC have produced a good introduction to RSS here http://news.bbc.co.uk/1/hi/help/3223484.stm.

Using RSS with WFi

So, where does this tie up with WFi? Well, think about the life of a complex process. Within the process are key sub-sections that have significance to a particular subscriber (e.g. Chief Financial Officer, customer, manager). Whilst task-to-task actions can be examined using the Infor ERP System i Workspace Action Tracker this can often be overkill for someone just wanting to know if “order for customer X is ready”, especially if the reader does not have a copy of the process model to reference. Using RSS we can define key events within a process (think of them as “milestones”), let the subscriber know when such a milestone has been achieved, and provide them with an interface to “find out more” about the event (e.g. call back to System21 Aurora, public interface).

Adding RSS to Order Fulfilment

To demonstrate how to use RSS with WFi this tutorial will use an Order Fulfilment process and insert XML Activities within it that will create the necessary RSS elements.

The following pre-requisites are required for this example…

- a copy of the Order Fulfilment Process
- a web server
- a RSS News Reader

> **Note**: The RSS feed XML file will require a web server to publish its information to the RSS News Reader. As Infor ERP System i Workspace users will already have a web server (IBM WebSphere) this shouldn't be a problem but if a web server is not available then there are free/open source solutions, such as Apache and Fnord, available.

For this example we will assume that the intended recipient of the feed is a Sales Manager, or Chief Financial Officer who wants to keep an eye on order processing within their system. The RSS feed messages we create will be aimed at providing them with an overview of the status of the order process.

The first step is to create our basic RSS feed XML file. Create a new XML file and enter the following content…

```
<?xml version="1.0"?>
<rss version="2.0">
   <channel>
      <title>Infor Order Fulfilment</title>
      <link>http://myWorkspace.com/aurora/</link>
      <description>Monitor key events within the Infor Order
      Fulfilment process.</description>
      <language>en-gb</language>
      <lastBuildDate>Thu, 17 Jan 2008 09:00:00 GMT</lastBuildDate>
   </channel>
</rss>
```

> **Note**: You can find out more about the RSS 2.0 XML format here http://blogs.law.harvard.edu/tech/rss.

The top level element of an RSS 2.0 document must be `rss` which can have a single `channel` child element. The `title`, `link` and `description` are mandatory child elements of the `channel` element. They represent…

Element	Description
title	The name of the channel. It's how people refer to your service. If you have an HTML website that contains the same information as your RSS file, the title of your channel should be the same as the title of your website.
link	The URL to the HTML website corresponding to the channel. In our example we have linked to the initial page of our System i Workspace server.
description	Phrase or sentence describing the channel.

In our example we have utilised some optional elements…

Element	Description
language	The language the channel is written in. This allows aggregators to group all Italian language sites, for example, on a single page. We are using British English (en-gb).
lastBuildDate	The last date and time the content of the channel changed. All date-times in RSS conform to the Date and Time Specification of RFC 822, with the exception that the year may be expressed with two characters or four characters (four preferred).

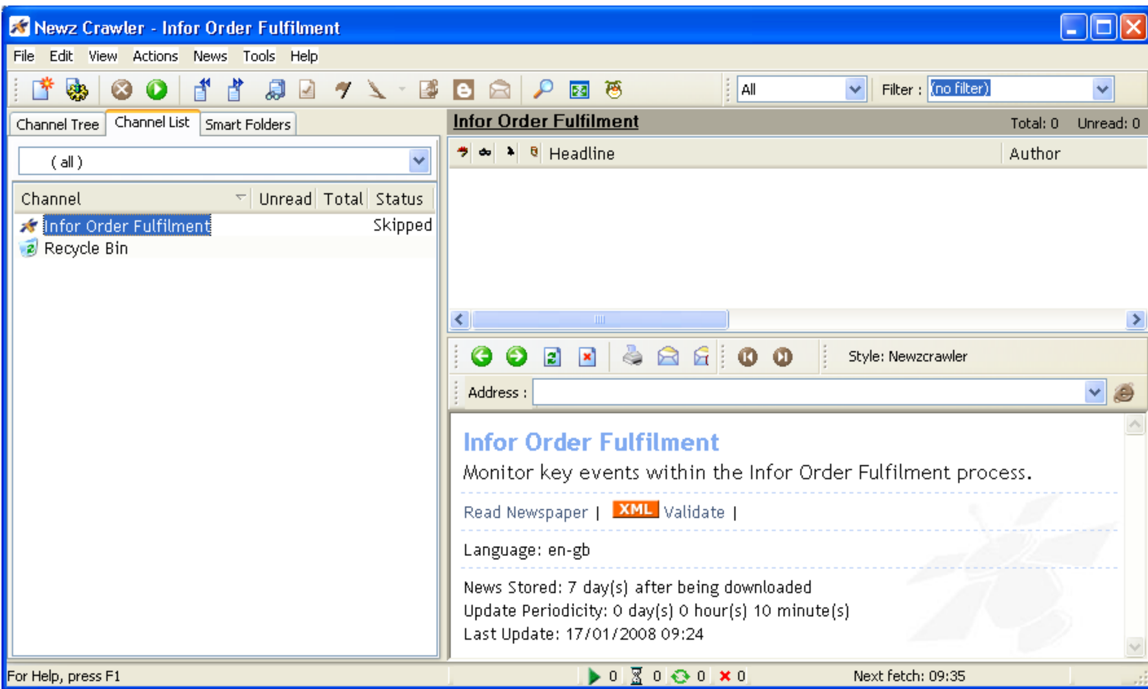
Save the file with a suitable name (e.g. `Order_Fulfilment.XML`) on your web server. You now need to create an alias within your web-server to the directory that contains the file which can be used as the URL for the RSS feed. For example, if you create an alias called RSS; your URL would be `http://myserver/RSS/Order_Fulfilment.XML`.

You should now be able to register and open the file within a RSS News Reader and see the initial page (no items will be available).

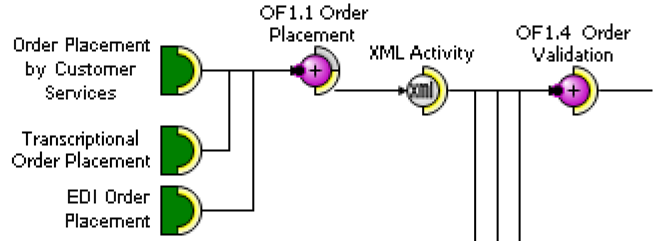
Note: For this example, we are using the Newz Crawler from ADC Software (http://www.newzcrawler.com/). This is a Microsoft Windows based RSS news reader that has many useful features for configuring and obtaining RSS feeds. A trial version is available for download (the full version requires a license fee).

Numerous RSS news readers are available (including one built into Internet Explorer 7) for most operating systems. You can also get RSS news readers that run on the web and on mobile devices.

Here is how it may look in your RSS news reader…



We are now ready to embed “milestone” events within our Order Fulfilment process. The first will mark the creation of a Sales Order. Open WFi Modeler and open the **OF Order Fulfilment (FMCG)** process. Drill down into the **OF1 Order Placement and Maintenance** process and drag a new XML Activity onto the link between the **OF1.1 Order Placement** and **OF1.4 Order Validation** activities; E.g.



To notify the RSS feed of a new order we create an `item` element as child of the `channel` element. The `item` element can have various child elements, but in our example we are only going to use the `title`, `description` and `pubDate` elements. The `item` element will take the following form…

```
<item>
   <title>Sales Order [reference] Created For Customer
   [customer name]</title>
   <description>A new Sales Order [reference] has been created
    for Customer [customer name].</description>
    <pubDate>Tue, 30 Jun 2005 12:00:00 GMT</pubDate>
</item>
```

Our stylesheet will need to read the existing RSS XML document and add our element into it. To do this, edit the XML source and insert the following XSL…

```
<xsl:stylesheet version="1.0"
     xmlns:process-data="com.geac.process.extensions.ProcessData"
     xmlns:lxslt="http://xml.apache.org/xslt"
     xmlns:javascript="javascript"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     xmlns:redirect="org.apache.xalan.lib.Redirect"
     extension-element-prefixes="redirect">

<xsl:output method="xml" media-type="text/xml"/>

<!--
       Create a date in the correct format
-->
<lxslt:component prefix="javascript" functions="gmtDate">
       <lxslt:script lang="javascript">
              function gmtDate()
```

```
            {
                   var objDate = new Date();
                   return objDate.toGMTString();
            }
       </lxslt:script>
</lxslt:component>

<!-- Get the information required for the message -->
<xsl:variable name="cusn"
       select="normalize-space(process-data:getProcessDataField
       (/process_activity_input_data/database_pool,
        /process_activity_input_data/environment_code,
        /process_activity_input_data/multi_thread_identifier,
        /process_activity_input_data/activity_number,
        'CUSTNAME')/get_process_data_field/data_field_data)"/>
<xsl:variable name="date" select="javascript:gmtDate()"/>
<xsl:variable name="ref" select="normalize-space
       (/process_activity_input_data/business_object_reference)"/>
<!--
       Process the default document handler input document
-->
<xsl:template match="process_activity_input_data">
       <!-- Read in the existing RSS XML file into a variable -->
       <xsl:variable name="rssOrig"
         select="document('c:/temp/RSS/Order_Fulfilment.xml')"/>
<!—
       Create and save the new item within the existing RSS XML
       file into a variable
-->
       <xsl:if test="$rssOrig">
              <redirect:write select=
                     "'c:/temp/RSS/Order_Fulfilment.xml'">
                     <xsl:apply-templates select="$rssOrig"/>
              </redirect:write>
       </xsl:if>

</xsl:template>
<!--
       This template processes the existing RSS document and adds
       a new item element
-->
```

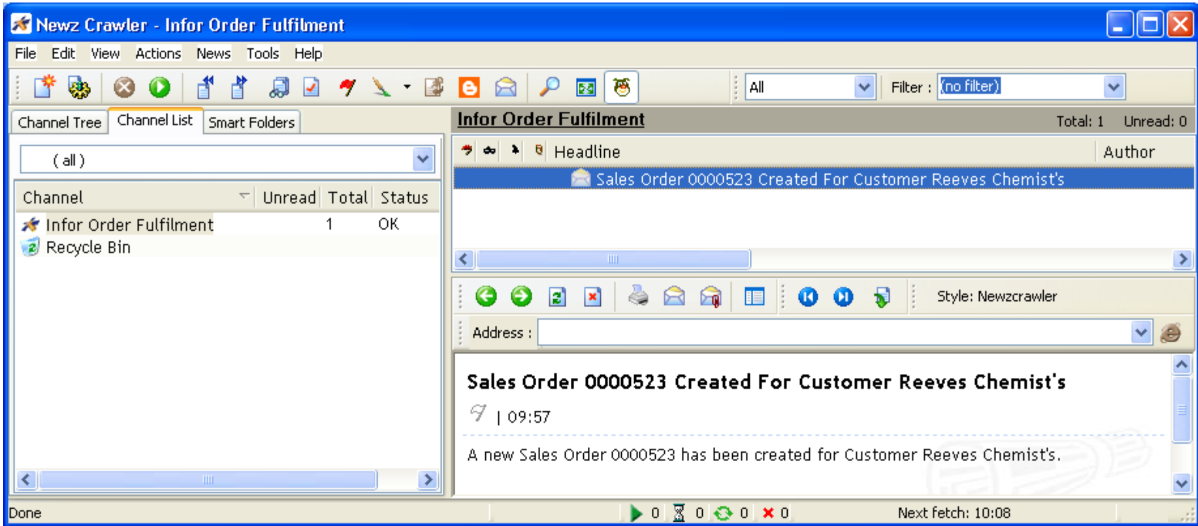
```
<xsl:template match="rss">
        <xsl:element name="rss">
                <xsl:attribute name="version">2.0</xsl:attribute>
                <xsl:apply-templates/>
        </xsl:element>
</xsl:template>

<xsl:template match="channel">
        <xsl:element name="channel">
                <xsl:apply-templates/>
                <!-- Create the new item -->
                <xsl:call-template name="create-item"/>
        </xsl:element>
</xsl:template>

<xsl:template match="title|link|description|language|item">
        <xsl:copy-of select="."/>
</xsl:template>


<xsl:template match="lastBuildDate">

        <xsl:element name="lastBuildDate"><xsl:value-of
        select="$date"/> </xsl:element>

</xsl:template>
<!--
        This template creates the new RSS item
-->
<xsl:template name="create-item">
        <xsl:element name="item">
                <xsl:element name="title">Sales Order <xsl:value-of
select="$ref"/> Created For Customer <xsl:value-of
select="$cusn"/></xsl:element>
                <xsl:element name="description">A new Sales Order
<xsl:value-of select="$ref"/> has been created for Customer
<xsl:value-of select="$cusn"/>.</xsl:element>
                <xsl:element name="pubDate"><xsl:value-of
select="$date"/></xsl:element>
        </xsl:element>
</xsl:template>

</xsl:stylesheet>
```

Note: In the example, we have assumed that the Document Handler is running on the same server that the RSS file is stored in, so it can be read and written to using a UNC file path (i.e. it is in the folder c:/temp/RSS). You could read and write via the web server URL but this may require extra configuration to allow a fixed user read/write access (adding write access for all users would be a security risk).

Note: In the example, we have assumed that only a single Document Handler instance will be accessing the file. If you are running multiple Document Handlers you will need to create some sort of file locking system so the file is not accessed by multiple processes, which could cause data loss.

The stylesheet uses the XSL `document` function to read the existing RSS file into a variable as a nodeset. We then process this and intercept any templates we are interested in. Once we have processed all the existing items, we add our new `item` element. The stylesheet updates the `lastBuildDate` element with the time stamp. The new document is re-directed back to the original file using the Xalan `redirect` extension element.

Once you have entered the XSL, save it and set the XML Activities properties to have a useful name/description. Save the process and activate it. Depending on your RSS news reader, on entering a new order (that isn't a quote) you should get an update. In the Newz Crawler we get a popup to notify us of the new event. On selecting the item the RSS news reader is launched, displaying the information…



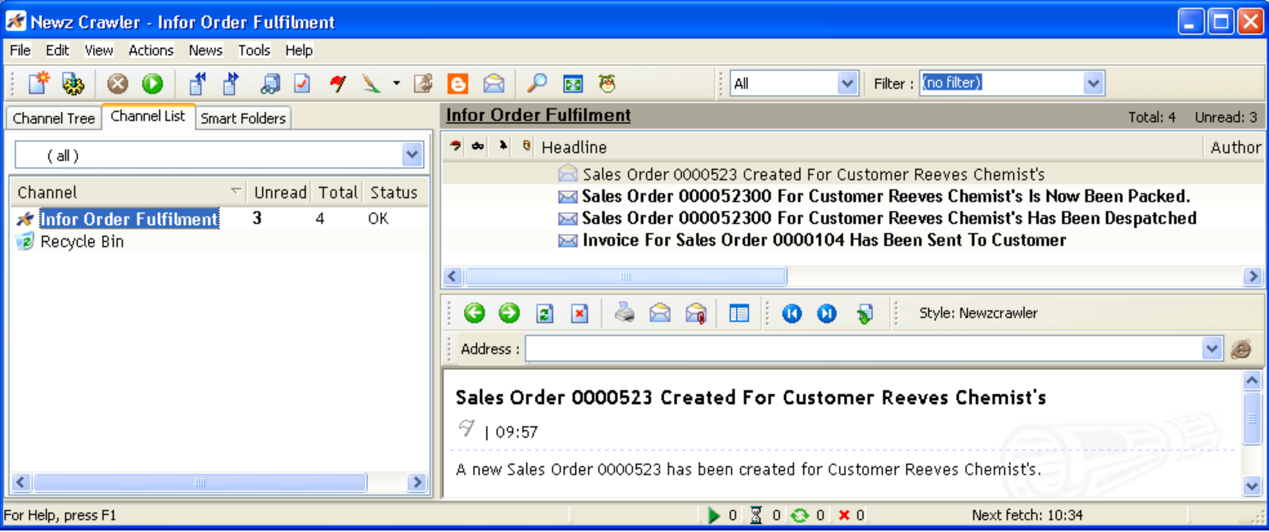
To add more events, all you need to do is duplicate the XML Activity to other parts of the process, altering the content of the `create-item` template to suit the event.

Tip: Why not filter the "static" parts of the stylesheet that do not alter into a separate document and import it using `<xsl:import>`.

The other parts of the stylesheet that you can add events to are…

- in **OF 1.2 Order Amendment** when all or part of an order is cancelled
- in **OF 2 Order Delivery** when an order is sent for picking or is despatched
- in **OF 3 Order Invoicing** when an invoice is sent to the customer

So for a complete process, the RSS feed recipient may see a list of new items similar to the one below…



Filtering Data Using Categories

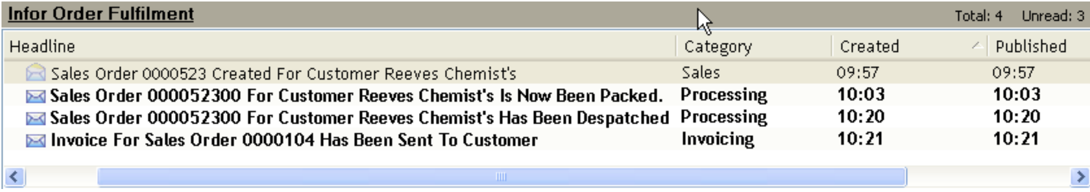
In our example above, every RSS feed recipient gets every message that is created, which again could be information overkill for some recipients. The RSS XML standard provides an extra layer of filtering called “categories”.

When creating an item you can add one or more categories using the `category` child element. The text of this element defines the category so it is up to the designer to create appropriate category names.

So, in our example, we will alter the items to include categories called **Sales** (when a sales order is created or amended), **Processing** (when an order is picked or despatched) and **Invoicing** (when financial information is created). Here is the updated create-item template.

```
<!--
      This template creates the new RSS item
-->
<xsl:template name="create-item">
      <xsl:element name="item">
            <xsl:element name="title">Sales Order
              <xsl:value-of select="$ref"/> Created For Customer
              <xsl:value-of select="$cusn"/>
            </xsl:element>
            <xsl:element name="description">A new Sales Order
              <xsl:value-of select="$ref"/> has been created for
              Customer <xsl:value-of select="$cusn"/>.
            </xsl:element>
            <xsl:element name="pubDate">
              <xsl:value-of select="$date"/>
            </xsl:element>
            <xsl:element name="category">Sales</xsl:element>
      </xsl:element>
</xsl:template>
```

When applied to our process we can see the different in our RSS feed…



Note: Not all RSS news readers support categories or category filtering. The alternative is to create an individual XML file for each category and then it is up to the recipient to subscribe to the ones they require. This may mean that some RSS XML Activities within the process write to more than one file. This is how the BBC site supplies its RSS feeds.

Further Things to Try

Being XML based, the RSS standard is very flexible, and provides a number of facilities you can leverage in your processes. Why not try some of the following…

- Use the `link` element (child of `item`) to create a URL that when selected will open a task within System i Workspace (e.g. for a Sales Order message link to **Whole Order Enquiry**).

- Create a SVG model of your process (using the tools provided with WFi) and put this on your web server and associate it with your processes RSS feed, so that when a recipient clicks on the channel overview they get to see the process it relates to.

- Use the `docs` element of the `channel` to connect to any process specific documentation.

- Create user and/or customer specific channels that are updated by multiple processes. Dynamically select the user/customer file based on the current data being processed.

Chapter 3 Performance Tips

3

Converting Manual Activities from XML to XSL

In WFi there are two types of activities that can be dispatched to a user/role/customer, Manual and XML. In basic terms you should use Manual Activities for simple messaging and use XML Activities when you want to do custom/advanced processing within your process.

To ease the move of an existing Manual Activity to the same function, but as an XML Activity, there is an option to "Convert to XSL". This will transform the Manual Activity XML into a XSL stylesheet. This is actually done using the Microsoft XML Parser. The output is generated by taking the Manual Activity XML along with the `manactTransform.XSL` stylesheet that can be found in your installation directory. By default this will….

- Convert XML into XSL.
- Replace any WM namespace elements to use the `getProcessDataField` extension function.

The Manual Activity will be converted into a XML Activity, with the "Process in Document Handler flag set" and the resultant stylesheet of the conversion will be automatically loaded in, ready for activation to your WFi host.

The `manactTransform.XSL` file can be freely altered to add any special site-specific customizations to the output. For example, the stylesheet uses the same JDBC database pool as the Document Handler for retrieving data fields (i.e. by passing the `database_pool` input document element to the `getProcessDataField` function). This could be converted to use a fixed pool name that is company/process specific.

Saving & Retrieving Data within the Document Handler Session

There is a special parameter within the Document Handler that can be used to store data to for use later on in the current process or other processes. To use this you need to add the following line to your stylesheet…

```
<xsl:param name="dochandlerDataStore"/>
```

This parameter gives you a link to a Java Hashtable object, and all the public methods it provides. The key methods are `put` (to put data into the object for later use using a unique lookup key), `get` (to remove data from the object using the unique key) and `remove` (deletes the key and its related value from the table).

The namespace declaration required for these methods is….

```
xmlns:java="http://xml.apache.org/xslt/java"
extension-element-prefixes="java"
```

Here is an example of the `put` method….

```
<xsl:copy-of select="java:put(
             $dochandlerDataStore,'passedData','Hello')"/>
```

The first parameter is the handle to the passed in object, which should always be the same. The second parameter is a unique reference key. The third parameter is the data. This can be any basic type (string, numeric etc.) but cannot be a node set or document fragment or some other complex types.

Here is an example of the `get` method….

```
<xsl:copy-of select="java:get(
             $dochandlerDataStore,'passedData')"/>
```

The parameters are the same as the first two of the `put` method. The data can be brought back “as is” or back into a XSL variable.

Here is an example of the `remove` method….

```
<xsl:copy-of select="java:remove(
                $dochandlerDataStore,'passedData')"/>
```

The parameters are the same as the first two of the `put` method.

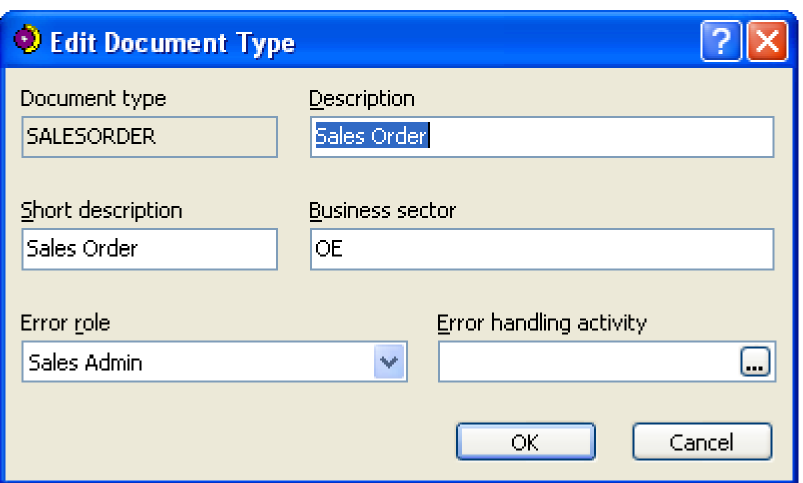
For more information on the methods available see
http://java.sun.com/j2se/1.5/docs/api/java/util/Hashtable.html

For more information about using Java from XSL see
http://xml.apache.org/xalan-j/extensions.html#java-namespace-declare

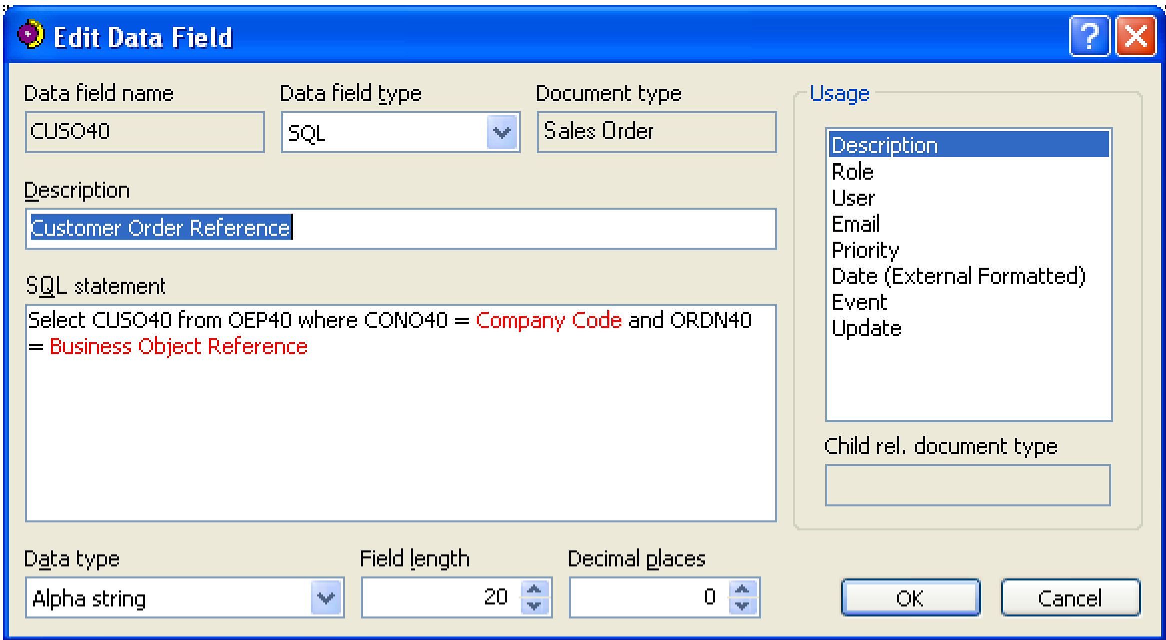
> **Note**: By storing the data on the Document Handler you potentially add a performance benefit, as the data field does not have to be re-evaluated every time, thus removing some of the work from the IBM i host. However, care should be taken to ensure the Document Handler object does not get overloaded with old data that has not been removed.

Example 1: Using the dochandlerDataStore Object in a Process

Start WFi Modeler and choose *Configure, WFi, Document Types*. Create a new document with the following properties (unless you already have created or imported a Sales Order document previously)…

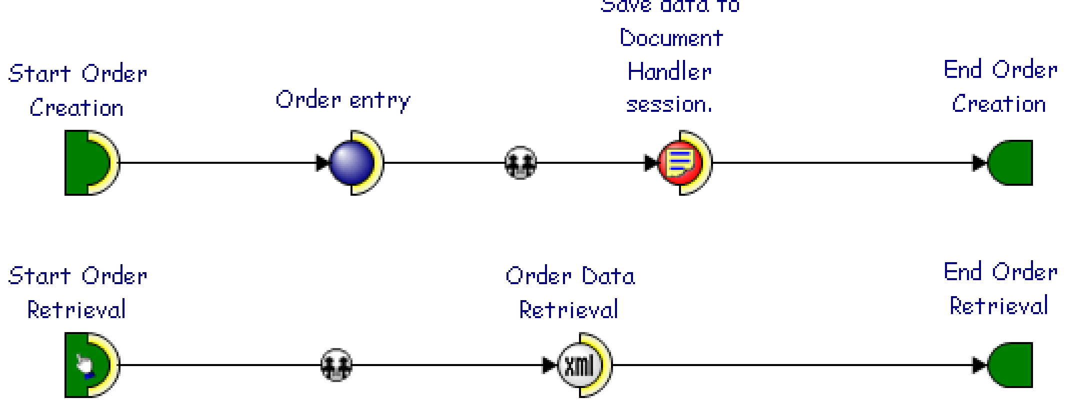


Choose *Configure, WFi, Data Fields*. Add a new data field with the following properties…

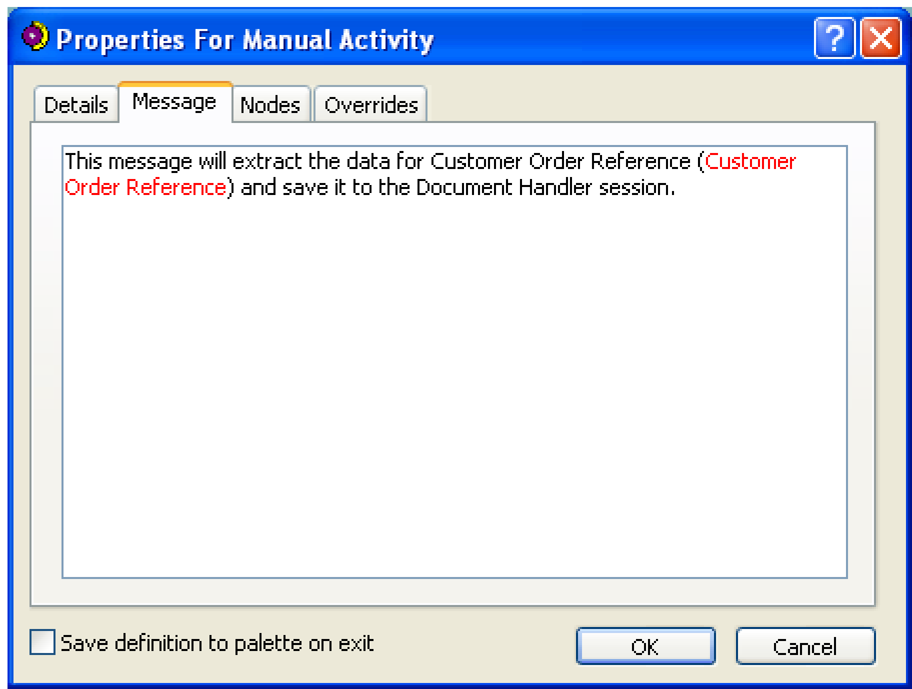


Note: "Company Code" and "Business Object Reference" are global data fields (use Insert Data Field option on right click menu and select "Show global data fields" option to see them in the list).

We can now create our test process. Select *File, New, Business Process* and call the process "Example 1". Drag the icons from the control palette and change the properties so your process looks like the screen shot below….



Edit the link after “Order Entry” to send recipient to the “Order Entry Clerk” role. Edit “Save data to Document Handler session” and change the input document type to “Sales Order” and the message to….



Select OK then right click on the object and select *XML, Convert to XSL*. Once the conversion is complete you can edit the stylesheet within WFi Modeler (by selecting *XML, Edit* from the context menu).

Make the following changes (Note: The new file will not have any line breaks. It has been re-formatted below for readability with the changes to the original highlighted in bold.)…

```
<xsl:stylesheet  version="1.0"
      xmlns:ma="http://www.geac.com/manact"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns:process-data=
             "com.geac.process.extensions.ProcessData"
      xmlns:java="http://xml.apache.org/xslt/java"
      extension-element-prefixes="java">

  <xsl:output method="xml" media-type="text/xml"/>

  <xsl:param name="dochandlerDataStore"/>

  <xsl:template match="process_activity_input_data">
```

```
<xsl:variable name="custRef"
      select="process-data:getProcessDataField(
            database_pool, environment_code,
            multi_thread_identifier,
            activity_number, 'CUSTREF')
            /get_process_data_field/data_field_data
            "/>
<xsl:variable name="putResult"
      select="java:put($dochandlerDataStore,
            'storedRef',string($custRef))"/>

<ma:MANUAL_ACTIVITY
      xmlns:wm="http://www.geac.com/workman">
  <ma:META_DATA>
      <ma:DATA_FIELD_DATA NAME="CUSTREF"
            DESCRIPTION="Customer Order Reference"
            DATA_TYPE="A" LENGTH="20"
            DECIMAL_PLACES="0">
            <xsl:value-of select="$custRef"/>
      </ma:DATA_FIELD_DATA>
  </ma:META_DATA>
  <ma:TITLE>Save data to Document Handler session
  </ma:TITLE>
  <ma:BODY>This message will extract the data for
Customer Order Reference (<ma:DATA_REF NAME="CUSTREF"/>) and save
it to the Document Handler session.
  </ma:BODY>
  <ma:COMPLETION_DETAILS>
      <ma:NODE COMPLETION_CODE="**">OK</ma:NODE>
      <ma:NODE ACTION="F3">Defer</ma:NODE>
      <ma:NODE ACTION="F10">Set WIP</ma:NODE>
  </ma:COMPLETION_DETAILS>
</ma:MANUAL_ACTIVITY>
</xsl:template>
</xsl:stylesheet>
```

The stylesheet uses the `docHandlerDataStore` object to put the customer reference data into, using a key of “storedRef”. In a real-world process you would need to make sure that this value is a unique string for this process/document type by using the business object, business object reference etc.

Once the changes are made, save and close the XML Activity edit window.

We now need to edit the second process path. Open the properties for the “Start Order Retrieval” node and set the document type to “Sales Order” and the start mode to “User requested”. Edit the link between the start and XML Activity and set the recipient to the “Order Entry Clerk” role.

Edit the “Order Data Retrieval” node and set the “Process XML document in Document Handler” check box and change the document type to “Sales Order” then close and use *XML, Edit* to create the following stylesheet (which you must manually enter yourself)…

```
<xsl:stylesheet version="1.0"
       xmlns:ma="http://www.geac.com/manact"
       xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
       xmlns:java="http://xml.apache.org/xslt/java"
       extension-element-prefixes="java">

  <xsl:output method="xml" media-type="text/xml"/>

  <xsl:param name="dochandlerDataStore"/>

  <xsl:template match="/">

       <ma:MANUAL_ACTIVITY
              xmlns:wm="http://www.geac.com/workman">
         <ma:META_DATA>
              <ma:DATA_FIELD_DATA NAME="CUSTREF"
                     DESCRIPTION="Customer Order Reference"
                     DATA_TYPE="A" LENGTH="20"
                     DECIMAL_PLACES="0">
                     <xsl:value-of select="java:get(
                            $dochandlerDataStore,
                            'storedRef')"/>
                     <xsl:variable name="dummy"
                            select="java:remove(
                            $dochandlerDataStore,
                            'storedRef')"/>
              </ma:DATA_FIELD_DATA>
         </ma:META_DATA>
         <ma:TITLE>
Retrieve data from Document Handler session
         </ma:TITLE>
         <ma:BODY>
This message will retrieve the data for Customer Order Reference (
<ma:DATA_REF NAME="CUSTREF"/>) from the Document Handler session.
         </ma:BODY>
         <ma:COMPLETION_DETAILS>
              <ma:NODE COMPLETION_CODE="**">OK</ma:NODE>
```

```
                <ma:NODE ACTION="F3">Defer</ma:NODE>
                <ma:NODE ACTION="F10">Set WIP</ma:NODE>
            </ma:COMPLETION_DETAILS>
        </ma:MANUAL_ACTIVITY>
    </xsl:template>
</xsl:stylesheet>
```

This stylesheet uses the `get` and `remove` methods to get the data back and then clear the object from the session.

All that now needs to be done is to give the process an activation code, validate, save and activate to the IBM i WFi system.

Make sure that you start off a copy of the Document Handler and do not shut it down between processing the two process paths.

You can start the first process by using the Advanced Order Entry program in Infor ERP System21 Aurora. Once the “Save data to Document Handler session” has been processed in the Document Handler you can kick off the second process using the “Business Process Launcher” in System i Workspace (use business object SALESORDER and any reference then select the “Start Order Retrieval” start node.

Saving Data Permanently from Your Stylesheet

The Document Handler storage object is very useful for passing data around but it does have one drawback in that if the Document Handler was to crash or be closed then the data on the session is lost. There are extension functions within XSL that can be used to save data permanently to a file for retrieval later.

To write data to a file you can use the redirect package that comes with Xalan.

The namespace declaration for this package is…

```
xmlns:redirect="org.apache.xalan.lib.Redirect"
```

Here is how you would use the function in your stylesheet….

```
<redirect:write select="'c:/temp/store_data.xml'">
        <xsl:copy-of select="$some-data"/>
</redirect:write>
```

This will save any data between the redirect:write elements to the named file. The select parameter must contain a string value so if you hard-code the path make sure to enclose the string in single quotes.

There are several methods to reload data into your stylesheet but the easiest to use is the document function that comes as part of the standard XSL function set…

```
<xsl:variable name="myData"
        select="document('c:/temp/store_data.xml')"/>
```

This will load the contents of the file into the variable `myData` that will be a nodeset.

Example 2: Storing data permanently in a Process

Taking the process that was created for example 1 we only need to change the stylesheets to show how to use the redirect functions. For this example we will take the original output from the “Save data to Document Handler session” XML Activity as a whole and save it to a file. This is the stylesheet required….

```
<xsl:stylesheet  version="1.0"
       xmlns:ma="http://www.geac.com/manact"
       xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
       xmlns:process-data=
              "com.geac.process.extensions.ProcessData"
       xmlns:redirect="org.apache.xalan.lib.Redirect"
       extension-element-prefixes="redirect">

  <xsl:output method="xml" media-type="text/xml"/>

  <xsl:template match="process_activity_input_data">
       <xsl:variable name="myData">
        <ma:MANUAL_ACTIVITY
              xmlns:wm="http://www.geac.com/workman">
         <ma:META_DATA>
              <ma:DATA_FIELD_DATA NAME="CUSTREF"
                     DESCRIPTION="Customer Order Reference"
                     DATA_TYPE="A" LENGTH="20"
                     DECIMAL_PLACES="0">
              <xsl:value-of select="process-data:
                     getProcessDataField(database_pool,
                     environment_code,
                     multi_thread_identifier,
                     activity_number, 'CUSTREF')
                     /get_process_data_field/data_field_data
              "/>
                     </ma:DATA_FIELD_DATA>
         </ma:META_DATA>
         <ma:TITLE>
Save data to Document Handler session
         </ma:TITLE>
         <ma:BODY>
This message will extract the data for Customer Order Reference (
<ma:DATA_REF NAME="CUSTREF"/>) and  save it to the Document
Handler session.
         </ma:BODY>
         <ma:COMPLETION_DETAILS>
              <ma:NODE COMPLETION_CODE="**">OK</ma:NODE>
              <ma:NODE ACTION="F3">Defer</ma:NODE>
              <ma:NODE ACTION="F10">Set WIP</ma:NODE>
```

```
            </ma:COMPLETION_DETAILS>
         </ma:MANUAL_ACTIVITY>
       </xsl:variable>
<!—
       Make sure to copy the variable out so this activity has
       some output
-->
       <xsl:copy-of select="$myData"/>

       <redirect:write select="'c:/temp/store_data.xml'">
            <xsl:copy-of select="$myData"/>
       </redirect:write>

      </xsl:template>

     </xsl:stylesheet>
```

Create a new file and enter the data above then save. Import this data into the “Save data to Document Handler session” XML Activity.

The stylesheet to re-load the data is much simpler….

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet  version="1.0"
       xmlns:ma="http://www.geac.com/manact"
       xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
       <xsl:template match="/">
              <xsl:copy-of select=
                     "document('c:/temp/store_data.xml')"/>
       </xsl:template>
</xsl:stylesheet>
```

Create the file and import this into the “Order Data Retrieval” XML Activity.

At runtime you should see the output from the first XML Activity duplicated in the second process.

Retrieving Multiple Data Fields in a Single SQL Call

If you use a large amount of data fields that come from the same file the default system will have to issue a complete SQL enquiry for each one. By using some Infor ERP System21 Aurora domain knowledge and the `sqlqueryFunction` extension the amount of SQL access can be reduced.

The namespace declaration for this package is…

```
xmlns:sql="com.geac.xtrane.extensions.SQLQuery"
```

Here is how you would use the function in your stylesheet….

```
<xsl:variable name="sqlQuery"
       select="select * from X where Y='Z'"/>
<xsl:variable name="sqlResult"
       select="sql:sqlqueryFunction($sqlQuery)"/>
```

The only parameter of this function is a string containing the SQL statement to execute. The function will return a nodeset containing details about the table and the results from the SQL statement (if it was successful).

There is also a variant of the sqlqueryFunction that can control the number of lines returned…

```
<xsl:variable name="sqlResult"
       select="sql:sqlqueryFunction(string(database_pool),
               $sqlQuery, 10)"/>
```

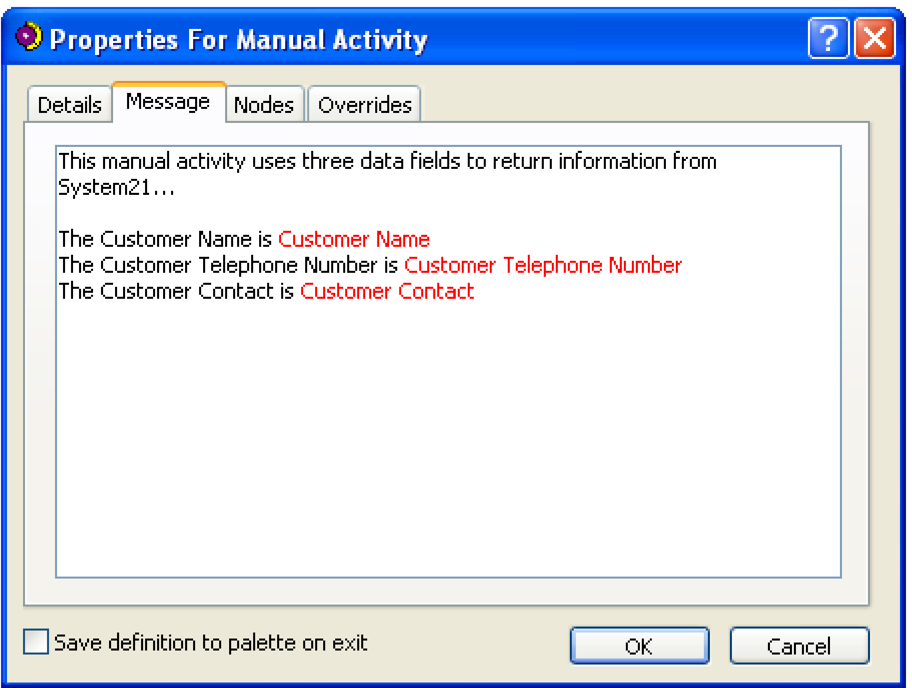
The first parameter is the name of the pool. This has to be specified so we use the same one that the Document Handler is using. The second parameter is the SQL query and the last parameter is the number of records to return.

Example 3: Using sqlqueryFunction to Retrieve Data Fields

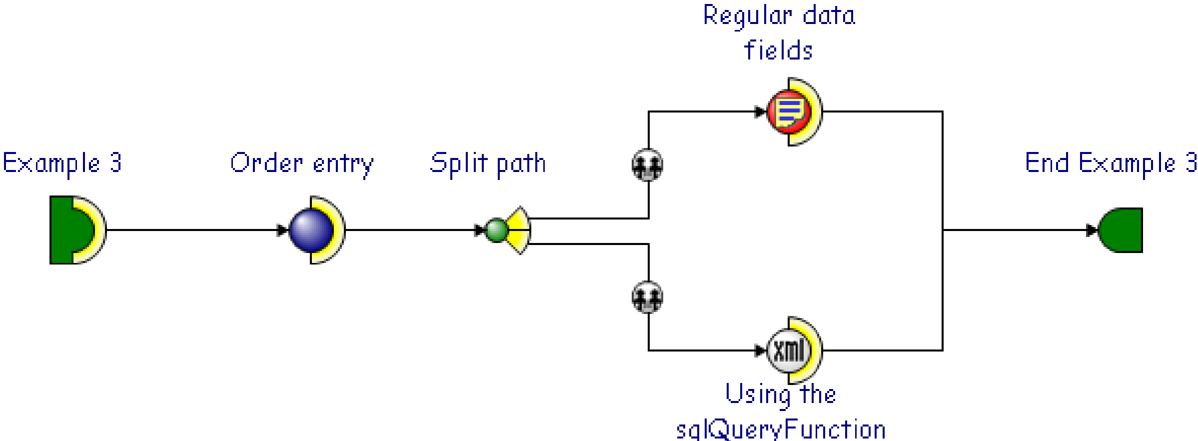
In WFi Modeler go into the Configure->WFi->Data Fields dialog and add the new data fields as defined below…

Data field name	Description	Length	SQL
CUSTNAME	Customer Name	35	Select CNAM05 from OEP40,SLP05 where CONO05 = CONO40 AND CUSN05 = CUSN40 AND DSEQ05 = DSEQ40 AND CONO40 = Company Code and ORDN40 = Business Object Reference
CUSTTEL	Customer Telephone Number	20	Select PHON05 from OEP40,SLP05 where CONO05 = CONO40 AND CUSN05 = CUSN40 AND DSEQ05 = DSEQ40 AND CONO40 =Company Code and ORDN40 = Business Object Reference
CUSTCONTACT	Customer Contact	35	Select CNTN05 from OEP40,SLP05 where CONO05 = CONO40 AND CUSN05 = CUSN40 AND DSEQ05 = DSEQ40 AND CONO40 = Company Code and ORDN40 = Business Object Reference

Each one should be type *SQL*, document type *Sales Order*, data type *Alpha* and usage *Description*. Now create a new business process and drag a Manual Activity control onto the palette, set its title to “Regular data fields”, document type to “Sales Order” and set its message details as defined below…



Drag the icons from the control palette and change the properties so your process looks like the screen shot below….



The recipients for the two activities should be redirected to the “Order Entry Clerk” role (as in example 1).

Edit the properties of the XML Activity and change the document type to “Sales Order” and check the “Process XML Document in Document Handler”. Now create the following stylesheet in the XML Activity editor…

```
<xsl:stylesheet version="1.0"
      xmlns:ma="http://www.geac.com/manact"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns:sql="com.geac.xtrane.extensions.SQLQuery"
      extension-element-prefixes="sql">

  <xsl:template match="process_activity_input_data">

      <xsl:variable name="sqlData">
                    select CNAM05,PHON05,CNTN05
                    from OSLD1F3.OEP40,OSLSLF3.SLP05
                    where CONO05 = CONO40 AND CUSN05 = CUSN40 AND
                    DSEQ05 = DSEQ40 AND CONO40 =
             '<xsl:value-of select="company_code"/>'
                    AND ORDN40 =
             '<xsl:value-of select="business_object_reference"/>'
      </xsl:variable>

      <xsl:variable name="sqlResult"
                    select="sql:sqlqueryFunction(
                    normalize-space($sqlData))"/>
      <xsl:variable name="rowResult"
                    select="$sqlresult/resultset/row[1]"/>

      <ma:MANUAL_ACTIVITY>
        <ma:META_DATA>
             <ma:DATA_FIELD_DATA NAME="CUSTNAME"
                    DESCRIPTION="Customer Name" DATA_TYPE="A"
                    LENGTH="35" DECIMAL_PLACES="0">
                    <xsl:value-of select="$rowResult/
                                  column[@name='CNAM05']/@value"/>
             </ma:DATA_FIELD_DATA>
             <ma:DATA_FIELD_DATA NAME="CUSTTEL"
                    DESCRIPTION="Customer Telephone Number"
                    DATA_TYPE="A" LENGTH="20" DECIMAL_PLACES="0">
                    <xsl:value-of select="$rowResult/
                                  column[@name='PHON05']/@value"/>
             </ma:DATA_FIELD_DATA>
             <ma:DATA_FIELD_DATA NAME="CUSTCONTACT"
                    DESCRIPTION="Customer Contact" DATA_TYPE="A"
                    LENGTH="35" DECIMAL_PLACES="0">
                    <xsl:value-of select="$rowResult/
                                  column[@name='CNTN05']/@value"/>
             </ma:DATA_FIELD_DATA>
        </ma:META_DATA>
        <ma:TITLE>Regular data fields</ma:TITLE>
        <ma:BODY>
```

```
This manual activity uses three data fields to return information
from System21...
                <ma:NEW_LINE/>
                <ma:NEW_LINE/>
                The Customer Name is <ma:DATA_REF NAME="CUSTNAME"/>
                <ma:NEW_LINE/>
                The Customer Telephone Number is
                <ma:DATA_REF NAME="CUSTTEL"/><ma:NEW_LINE/>
                The Customer Contact is
                <ma:DATA_REF NAME="CUSTCONTACT"/>
          </ma:BODY>
          <ma:COMPLETION_DETAILS>
                <ma:NODE COMPLETION_CODE="**">OK</ma:NODE>
                <ma:NODE ACTION="F3">Defer</ma:NODE>
                <ma:NODE ACTION="F10">Set WIP</ma:NODE>
          </ma:COMPLETION_DETAILS>
        </ma:MANUAL_ACTIVITY>
  </xsl:template>
</xsl:stylesheet>
```

The text in bold highlights the important parts of this stylesheet. We first construct our SQL statement and then use the `sqlqueryFunction` to retrieve the data. The first `row` element in the returned document is stored into a variable for faster access. Then, instead of calling the `evaluateDataField` function (as we did in example 1) we just remove the required element from the results nodeset of the `sqlqueryFunction`.

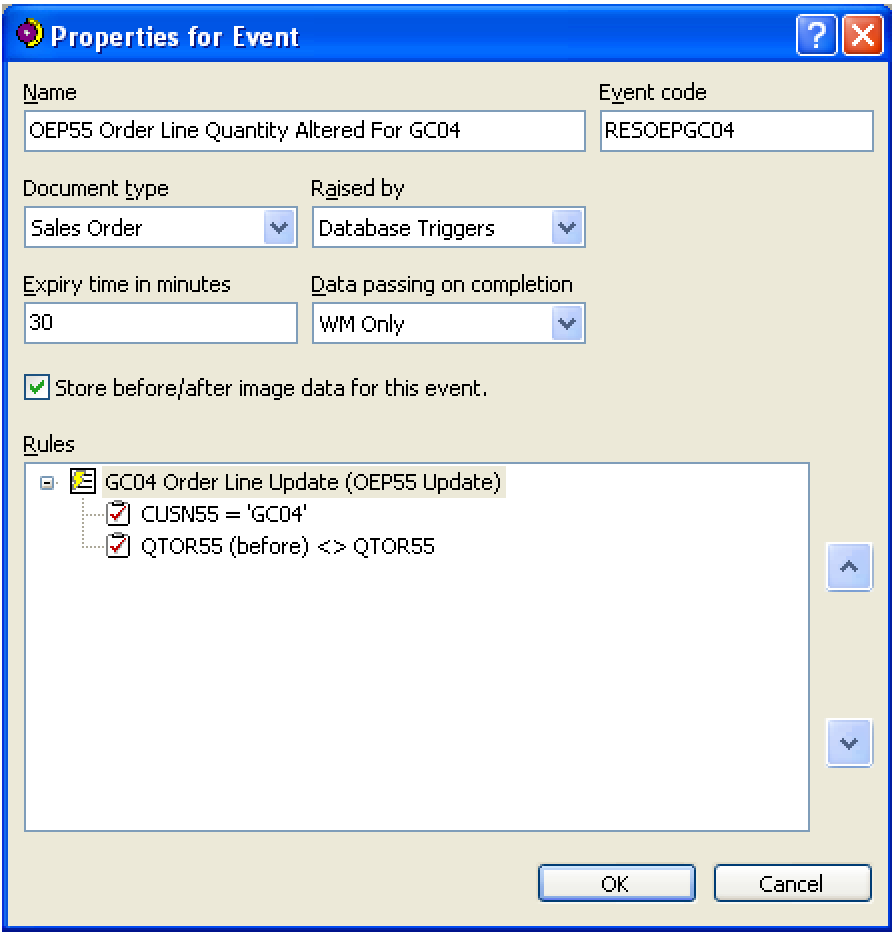
Import this into the XML Activity and save, validate and activate your process to your IBM i server. You should then see that the output from both activities is identical.

Chapter 4 Using Event Data Within A Process

4

Overview

When an event is created/edited in WFi Modeler there is a checkbox on the Event properties dialog called “Store before/after image data for this event”…



This checkbox can only be set when the “Data passing on completion” option includes WFi as one of its output types. By default the checkbox is off.

Turning this option on will cause the WFi Trigger Handler to write an XML document (representing the changed record) to the WFi file WFP72, but only when all the event rules are satisfied and passed.

The file for storing the before/after images is called WFP72. Its structure is…

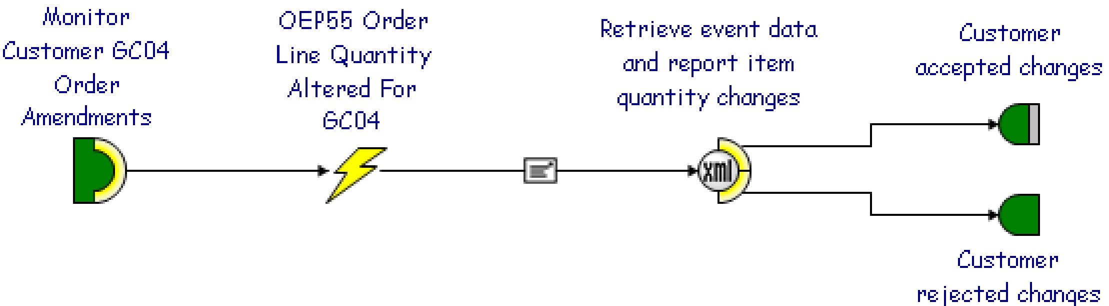
Name	Type
Company Code (CONO72)	Character * 2
Business Object Reference (IOBR72)	Character * 50
Business Object (IOBJ72)	Character * 10
Event Code (EVNT72)	Character * 10
Date created (DATE72)	Character * 8
Time created (TIME72)	Character * 9
Before/After Image (XML172)	CLOB

The content of this file is created by the WFi Trigger Handler and the retrieval and removal of records is controlled by XSL extension functions. The WFi Engine does not access this file.

The extension functions to retrieve and maintain the content of WFP72 are called `readEventDocuments` and `deleteEventDocuments`.

An Example of Using Event Data in a Process

Here is an example of a simple process that uses an XML Activity to read the event data and use it to generate a HTML email…



The database trigger event is defined over the Order Lines (OEP55) file (part of Customer Services and Logistics in System21 Aurora). It is defined so that when an order lines item quantity is changed, for an order for customer code GC04, the process is initiated. The "Store before/after image data for this event" is set on so that the Trigger Handler saves the event data into WFP72.

The XML activity is set to process inside the Document Handler and has two exit routes, accept and reject. The following stylesheet was imported into the XML activity…

```
<xsl:stylesheet version="1.0"
       xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
       xmlns:process-event="com.geac.process.extensions.Event">

  <xsl:output method="xml" encoding="utf-8"/>

  <xsl:template match="process_activity_input_data">
<!--
      Read the event document associated with the database change
-->
      <xsl:variable name="event-data-nodeset"
            select="process-event:readEventDocuments(
```

```
                database_pool, environment_code,'RESOEPGC04',
                business_object_reference, company_code,
                'SALESORDER')"/>
<html>
  <body>
       <h1>Order Update</h1>
       <xsl:choose>
<!--
       If the event data document could not be received then
       send an error message out
-->
       <xsl:when test="$event-data-nodeset/read_event_documents
                      /function_error">
             <p>
We were unable to automatically retrieve the details of your order
amendment due to problems with our system. Please contact our
support desk immediately on 0891 505050
             </p>
       </xsl:when>
<!--
       Use the event data to create an e-mail
-->
       <xsl:otherwise>
             <p>
An item that is part of the order number <b>
                    <xsl:value-of select="$event-data-nodeset/
                           read_event_documents/event_document/
                           SALESORDER/tables/update_table/fields/
                           field[./name='ORDN55']/new"/>
             </b> has been altered.
             </p><p>
The quantity ordered for item <b>
                    <xsl:value-of select="$event-data-nodeset/
                             read_event_documents/event_document/
                             SALESORDER/tables/update_table/fields/
                             field[./name='CATN55']/new"/>
             </b> was changed from <b>
                    <xsl:value-of select="$event-data-nodeset/
                           read_event_documents/event_document/
                            SALESORDER/tables/update_table/fields/
```

```
                          field[./name = 'QTOR55']/old"/>
              </b> to <b>
                     <xsl:value-of select="$event-data-nodeset/
                          read_event_documents/event_document/
                          SALESORDER/tables/update_table/fields/
                          field[./name='QTOR55']/new"/>
              </b>.
              </p><p>
Please respond as to whether this change should be processed by
choosing one of the buttons below.
              </p>
<!--
       Display the completion route buttons
-->
       <table border="0">
         <tr>
              <td style="background-color:lime;
                          padding:5; width:50;">
                <center><b>
                     <a STYLE="cursor:hand;
                          color:white; text-decoration:none">
                           <xsl:attribute name="href">
                     &lt;@PM:CC=**&gt;
                          </xsl:attribute>Accept
                     </a>
                </b></center>
              </td>
              <td style="padding:5;"/>
              <td style="background-color:red;
                       padding:5; width:50;">
                <center><b>
                     <a STYLE="cursor:hand; color:white;
                                 text-decoration:none">
                       <xsl:attribute name="href">lt;@PM:CC=01&gt;
                     </xsl:attribute> Reject
                     </a>
                </b></center>
              </td>
         </tr>
       </table>
```

```
<!--
       Remove the event document from WFP72
-->
       <xsl:variable name="delete-result-nodeset"
              select="process-event:deleteEventDocuments
                     (database_pool, environment_code,'RESOEPGC04',
                      business_object_reference, company_code,
                      'SALESORDER')"/>
       </xsl:otherwise>
       </xsl:choose>
  </body>
</html>
</xsl:template>
</xsl:stylesheet>
```

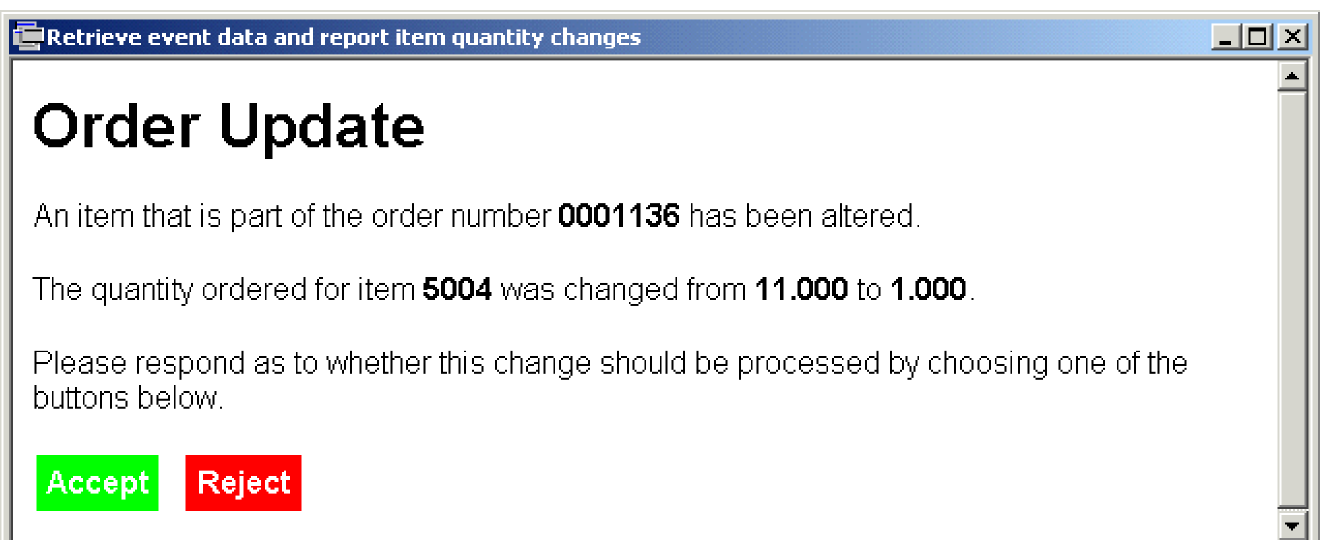
This stylesheet retrieves the data associated with the event using the `readEventDocuments` function. The event data will contain a complete copy of the database record that was altered including all its fields and the before and after values. The stylesheet uses the data to construct a HTML format message informing the user of the item that has changed and showing the old and new quantity values.

Additionally, it puts two completion buttons on the bottom of the HTML that are converted, by the Email Writer, at runtime into completion paths so that the correct information is returned to the process.

If, for some reason, the event document cannot be read an error message is sent instead.

The HTML is then sent, via Email, to the customer, allowing them to decide whether to confirm or reject the order (the reply is read by the Email Reader).

Here is an example of what the stylesheet will generate…


Retrieve event data and report item quantity changes
Order Update
An item that is part of the order number 0001136 has been altered.
The quantity ordered for item 5004 was changed from 11.000 to 1.000.
Please respond as to whether this change should be processed by choosing one of the buttons below.
Accept
Reject

Appendix A Finished Example

A

Here is the finished stylesheet for the order confirmation message tutorial in chapter 2.

Note: Some of the rows have been wrapped to fit within this documents formatting.

```
<xsl:stylesheet  version = "1.0"
                 xmlns:sql="com.geac.xtrane.extensions.SQLQuery"
                 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" media-type="text/html"/>

  <xsl:template match="process_activity_input_data">
    <HTML>
      <BODY>
        <!-- Title -->
        <H2 ALIGN="CENTER">Order Quotation</H2>
        <!-- Order Header -->
        <xsl:call-template name="order-header"/>
        <!-- Ordered Items -->
        <xsl:call-template name="order-lines"/>
        <!-- Ordered Footer -->
        <P>
          <TABLE WIDTH="100%" BORDER="0">
            <TR>
              <TD ALIGN="CENTER">Please call 0891 505050 to confirm this
order with our sales staff, quoting the reference numbers
above.<BR/><BR/><I>Thank you for your business.</I></TD>
            </TR>
          </TABLE>
        </P>
      </BODY>
    </HTML>
  </xsl:template>


  <xsl:template name="order-header">

    <!--
      Get the order header detail from System21 Aurora
    -->
```

```
<xsl:variable name="sqlOEP40">
  select CUSN40, DSEQ40, DTSO40,CUSO40 from OEP40
  where CONO40='<xsl:value-of select="company_code"/>'
  and ORDN40='<xsl:value-of select="business_object_reference"/>'
</xsl:variable>

<!--
  Use the same Document Handler database pool and environemnt
-->
<xsl:variable name="resultOEP40"
    select="sql:sqlqueryFunction(database_pool,
           environment_code, normalize-space($sqlOEP40), 1)"/>
<P>
<TABLE WIDTH="100%" BORDER="0">
  <TR>
    <TD WIDTH="50%" VALIGN="TOP">
      <TABLE BORDER="0">
        <TR>
          <TD><B>Our Order Reference:</B></TD>
          <TD><xsl:value-of select="business_object_reference"/></TD>
        </TR>
      <TR>
        <TD><B>Your Order Reference:</B></TD>
        <TD><xsl:value-of select="$resultOEP40/resultset/row/
                                  column[@name='CUSO40']/@value"/></TD>
      </TR>
      <TR>
        <TD><B>Order Date:</B></TD>
        <TD><xsl:value-of select="$resultOEP40/resultset/row/
                                  column[@name='DTSO40']/@value"/></TD>
      </TR>
    </TABLE>
  </TD>

  <!--
    Get the delivery address from System21 Aurora
  -->
  <xsl:variable name="sqlSLP05">
    select * from SLP05
    where CUSN05='<xsl:value-of select="$resultOEP40/resultset/row/
```

```
      column[@name='CUSN40']/@value"/>'
        and DSEQ05='<xsl:value-of select="$resultOEP40/resultset/row/

      column[@name='DSEQ40']/@value"/>'
      </xsl:variable>

      <xsl:variable name="resultSLP05"
                    select="sql:sqlqueryFunction(database_pool,
                       environment_code, normalize-space($sqlSLP05), 1)"/>

      <xsl:for-each select="$resultSLP05/resultset/row">

        <TD WIDTH="50%" VALIGN="TOP">
          <B>Delivery Address:</B><BR/><BR/>
          <xsl:value-of select="column[@name='CNAM05']/@value"/><BR/>
          <xsl:value-of select="column[@name='CAD105']/@value"/><BR/>
          <xsl:value-of select="column[@name='CAD205']/@value"/><BR/>
          <xsl:value-of select="column[@name='CAD305']/@value"/><BR/>
          <xsl:value-of select="column[@name='CAD405']/@value"/><BR/>
          <xsl:value-of select="column[@name='CAD505']/@value"/><BR/>
          <xsl:value-of select="column[@name='PCD105']/@value"/>
          <xsl:value-of select="column[@name='PCD205']/@value"/><BR/>
        </TD>

      </xsl:for-each>
      </TR>
    </TABLE>
    </P>
  </xsl:template>

  <xsl:template name="order-lines">

    <!--
      Get the item lines from System21 Aurora for this order
    -->
    <xsl:variable name="sqlOEP55">
      select CATN55, QTOR55, ORLV55 from OEP55
      where CONO55='<xsl:value-of select="company_code"/>'
      and ORDN55='<xsl:value-of select="business_object_reference"/>'
    </xsl:variable>
```

```
<xsl:variable name="resultOEP55"
    select="sql:sqlqueryFunction(database_pool, environment_code,
            normalize-space($sqlOEP55), -1)"/>
<P>
<TABLE WIDTH="100%" BORDER="1">
  <TR>
    <TH ALIGN="LEFT">Item Number</TH>
    <TH ALIGN="LEFT">Item Description</TH>
    <TH ALIGN="CENTER">Item Quantity</TH>
    <TH ALIGN="RIGHT">Cost</TH>
  </TR>

  <!--
    Store these values in local variables because when we go into
    the for-each loop we can no longer access them as the document
    reference is altered to the one in the select statement
  -->
  <xsl:variable name="pool" select="database_pool"/>
  <xsl:variable name="env" select="environment_code"/>
  <xsl:variable name="company" select="company_code"/>

  <!--
    Output 1 table row for each item
  -->
  <xsl:for-each select="$resultOEP55/resultset/row">

    <!--
      Get each item description from System21 Aurora
    -->
    <xsl:variable name="sqlINP35">
      select PDES35 from INP35
      where CONO35='<xsl:value-of select="$company"/>'
      and PNUM35='<xsl:value-of
      select="column[@name='CATN55']/@value"/>'
    </xsl:variable>

    <xsl:variable name="resultINP35"
            select="sql:sqlqueryFunction($pool, $env,
                    normalize-space($sqlINP35), 1)"/>
    <TR>
```

```
                <TD ALIGN="LEFT">
                    <xsl:value-of select="column[@name='CATN55']/@value"/></TD>
                <TD ALIGN="LEFT"><xsl:value-of select="$resultINP35/resultset/
                    row/ column[@name='PDES35']/@value"/></TD>
                <TD ALIGN="CENTER">
                    <xsl:value-of select="column[@name='QTOR55']/@value"/></TD>
                <TD ALIGN="RIGHT">
                    <xsl:value-of select="column[@name='ORLV55']/@value"/></TD>
              </TR>
            </xsl:for-each>
          </TABLE>
          </P>
        </xsl:template>
      </xsl:stylesheet>
```

Appendix B Error Handling

B

When calling any extension function, especially ones that use JDBC and IBM WebSphere MQ, you should always check the returned nodeset for errors and act accordingly on those errors. One way of doing this is by putting the function calls within templates and calling them. The template can then have an inbuilt amount of retry attempts.

Here is an example of using the `sqlqueryFunction` with error handling.

```
<xsl:stylesheet version = "1.0"
                xmlns:sql="com.geac.xtrane.extensions.SQLQuery"
                xmlns:xalan="http://xml.apache.org/xalan"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml" media-type="text/xml"/>

  <xsl:template match="process_activity_input_data">
    <!--
          Select from WFP52
    -->
    <xsl:variable name="sql">select * from WFP52</xsl:variable>

    <xsl:variable name="sqlRTF">
       <!--
             Call the query up to 5 times
        -->
        <xsl:call-template name="sqlqueryFunction">
          <xsl:with-param name="pool"        select="database_pool"/>
          <xsl:with-param name="environment" select="environment_code"/>
          <xsl:with-param name="statement"   select="$sql"/>
          <xsl:with-param name="records"     select="10"/>
          <xsl:with-param name="retries"     select="5"/>
        </xsl:call-template>
    </xsl:variable>

    <xsl:variable name="sqlResult" select="xalan:nodeset($sqlRTF)"/>

    <xsl:choose>
      <xsl:when test="$sqlResult/resultset/error">
          <error>Could not execute SQL statement.</error>
      </xsl:when>
```

```
    <xsl:otherwise>
        <xsl:copy-of select="$sqlResult"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<!--
      The sqlqueryFunction template (re-usable)
-->
<xsl:template name="sqlqueryFunction">
  <xsl:param name="pool"         select="'sql'"/>
  <xsl:param name="environment"  select="'AUL'"/>
  <xsl:param name="statement"/>
  <xsl:param name="records"      select="0"/>
  <xsl:param name="retries"      select="0"/>

  <xsl:variable name="sqlResult"  select="sql:sqlqueryFunction($pool,
                                   $environment, string($statement),
                                   number($records))"/>
 <!--
       Check for errors
  -->
  <xsl:choose>
    <!--
        If there is an error node then the sql has failed
    -->
    <xsl:when test="$sqlResult/resultset/error and number($retries)">
      <xsl:message>
          Call to sqlQueryFunction failed:
          <xsl:value-of select="$sqlResult/resultset/error/message"/>
      </xsl:message>
      <!--
          Call the function again
      -->
      <xsl:call-template name="sqlqueryFunction">
        <xsl:with-param name="pool"         select="$pool"/>
        <xsl:with-param name="environment"  select="$environment"/>
        <xsl:with-param name="statement"    select="$statement"/>
        <xsl:with-param name="records"      select="$records"/>
        <xsl:with-param name="retries"      select="$retries - 1"/>
```

```
        </xsl:call-template>
      </xsl:when>
      <xsl:otherwise>
          <xsl:copy-of select="$sqlResult"/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
</xsl:stylesheet>
```

The above stylesheet uses XSL's ability to recursively call templates to call itself when an error occurs. At the point when an error is detected a pause could be introduced (using the Java API) to delay the time between SQL queries (in case the fault is due to network interruption).

Appendix C XSL-FO to PDF Extension Function

The following Java class file can be compiled to create an XSL extension function within the WFi extensions package called `convertFOToPDF`.

```
package com.geac.process.extensions;

// Java
//
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStream;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.transform.Result;
import javax.xml.transform.Source;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.sax.SAXResult;
import javax.xml.transform.stream.StreamSource;

import org.apache.fop.apps.FOUserAgent;
import org.apache.fop.apps.Fop;
import org.apache.fop.apps.FopFactory;
import org.apache.fop.apps.FormattingResults;
import org.apache.fop.apps.MimeConstants;
import org.apache.fop.apps.PageSequenceResults;
import org.w3c.dom.Document;
import org.w3c.dom.Element;

public class FOP
{
    public static Document convertFOToPDF(String xslFOFile,
                                          String pdfFile)
    {
        // Create the result document factory
        //
        Document doc = newDocument();

        // Root node of the resultset fragment
        //
        Element eRoot = doc.createElement("convert_fo_to_pdf");
        doc.appendChild(eRoot);

        // Load files
        //
        File fo = new File(xslFOFile);
        File pdf = new File(pdfFile);
        OutputStream out = null;

        try
        {
                // Create the return DOM
                //
                Element eParm = doc.createElement("extension_information");
                Element eName = doc.createElement("function_name");
                eName.appendChild(doc.createTextNode("convertFOToPDF"));
                eParm.appendChild(eName);

                Element eInput = doc.createElement("input_file");
                eInput.appendChild(doc.createTextNode(xslFOFile));
                eParm.appendChild(eInput);

                Element eOutput = doc.createElement("output_file");
                eOutput.appendChild(doc.createTextNode(pdfFile));
                eParm.appendChild(eOutput);
```

```
            eRoot.appendChild(eParm);

            // Construct FOP driver
            //
            FopFactory fopFactory = FopFactory.newInstance();
            FOUserAgent foUserAgent = fopFactory.newFOUserAgent();

            // Setup output stream (buffered for performance)
            //
            out = new FileOutputStream(pdf);
            out = new BufferedOutputStream(out);

            // Construct fop with desired output format
            //
            Fop fop = fopFactory.newFop(MimeConstants.MIME_PDF,
                                        foUserAgent, out);

            // Setup JAXP using identity transformer
            //
            TransformerFactory factory =
                    TransformerFactory.newInstance();
            Transformer transformer =
                    factory.newTransformer();

            // Setup input stream
            //
            Source src = new StreamSource(fo);

            // Resulting SAX events (the generated FO) must be
            // piped through to FOP
            //
            Result res = new SAXResult(fop.getDefaultHandler());

            // Start XSLT transformation and FOP processing
            //
            transformer.transform(src, res);

            // Result processing
            //
            FormattingResults foResults = fop.getResults();
            java.util.List pageSequences =
                    foResults.getPageSequences();

            for (java.util.Iterator it = pageSequences.iterator();
                 it.hasNext();)
            {
                    PageSequenceResults pageSequenceResults =
                            (PageSequenceResults) it.next();
                    System.out.println("PageSequence " +
                            (String.valueOf(
                            pageSequenceResults.getID()).length() > 0 ?
                            pageSequenceResults.getID() : "<no id>")
                            + " generated " +
                            pageSequenceResults.getPageCount() + "
                            pages.");
            }

            System.out.println("Generated " + foResults.getPageCount()
                                    + " pages in total.");

            // Remove the input file
            //
            out.close();
            fo.delete();

        }
        catch (Exception e)
        {
```

```
                Element eError = doc.createElement("error");
                Element eDesc = doc.createElement("message");

                eDesc.appendChild(doc.createCDATASection(e.getMessage()));
                eError.appendChild(eDesc);

                Element eCode = doc.createElement("code");
                eCode.appendChild(doc.createTextNode("E8"));
                eError.appendChild(eCode);
                eRoot.appendChild(eError);
        }
        return doc;
    }

    // XML Helper method
    //
    public static Document newDocument()
    {
        DocumentBuilderFactory factory =
                DocumentBuilderFactory.newInstance();

        factory.setNamespaceAware(true);

        try
        {
                DocumentBuilder builder = factory.newDocumentBuilder();
                return builder.newDocument();
        }
        catch (ParserConfigurationException ex)
        {
                throw new RuntimeException(ex.getMessage());
        }
    }
}
```

convertFOToPDF

Extension Function Definition

```
convertFOToPDF(string xslFOFile, string pdfFile)
```

Description

The function will take an input file in XSL-FO form and create a PDF file in the passed destination location.

The 'xslFOFile' parameter is the file name and path to the XSL-FO. The file must be in valid XSL-FO format and be accessible from the calling application.

The 'pdfFile' parameter is the file name and path to the destination PDF file that the function will create. The file does not need to exist.

Return Nodeset Definition

```
<convert_fo_to_pdf>

        <extension_information>
                <function_name>convertFOToPDF</function_name>
                <input_file>{…}</input_file>
                <output_file>{…}</output_file>
        </extension_information>

<!--
    Optional - returned when the function cannot complete successfully
-->
        <error>
                <message>{…}</message>
                <code>{…}</code>
        </error>

</convert_fo_to_pdf>
```

Example

The following stylesheet shows how to use the PDF conversion class within the Document Handler.

```
<xsl:stylesheet version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
        xmlns:fop="com.geac.process.extensions.FOP">

  <xsl:output method="xml" indent="yes" media-type="text/xml"/>

  <xsl:template match="process_activity_input_data">

        <xsl:copy-of select="fop:convertFOToPDF('c:\temp\fop.xml',
                              'c:\temp\fop.pdf')"/>

  </xsl:template>

</xsl:stylesheet>
```