XML Messaging with the SAP Business Connector Part 2:

Outbound IDoc-to-XML Data Mapping with XSLT and Java Services, and Inbound XML-to-IDoc Data Mapping

Robert Chu

Robert Chu joined SAP at the end of 1996. He currently works for the Integration and Certification Center at SAP Labs. His current focus is the SAP integration technologies. Robert has been regularly teaching classes in this area at SAP training centers, and is the main author of the BIT531 training course and a few other internal workshops.

(complete bio appears on page 102)

As most of us know all too well, a smooth information flow between your business partner's system and your own can have a large impact on the success of your business processes. The SAP Business Connector (BC) solution enables you to easily and reliably exchange IDocs with your business partners via XML messaging over the Internet.[1]

The first article of this two-part discussion ("XML Messaging with the SAP Business Connector Part 1: Direct IDoc-XML Data Exchange and Outbound IDoc-to-XML Data Mapping with Flow Services") demonstrated how you can use the SAP IDoc-XML format to exchange data directly with your business partner, if your business partner is able to support this format. However, this is not always something your business partners will be able to do, so you need to be able to exchange documents in formats other than SAP IDoc-XML. As you learned in the previous article, there are three data-mapping options available to you for such cases:

- Perform data mapping using a custom BC flow service.
- Perform data mapping using XSLT transformation with a custom XSLT stylesheet.
- Perform data mapping using a custom Java service.

In the discussion of outbound data mapping with non-SAP XML in the first article, I focused on the first option, which is widely adopted in Business Connector projects. Here in this article, I will discuss the

[1] At the time of writing, the SAP Exchange Infrastructure (XI) is still under restricted piloting.

remaining two options for sending outbound communication to your business partners (which require additional knowledge of XSLT and Java, respectively), and I will also walk you through how to receive *inbound* communication from your business partners.

To illustrate the points of this article, sample BC services, along with Java source code and files, are available for download at **www.SAPpro.com**.

Before diving right in to the XSLT transformation technique for outbound data mapping, let's take a step back and review some XSLT basics.

An Introduction to XSLT

Extensible Stylesheet Language Transformation (XSLT) is designed to transform XML documents into other XML documents. XSLT is a W3C recommendation as of November 1999, and is a natural choice for XML-to-XML transformation.

The following are the main characteristics of XSLT:

- It is XML-based.
- Like SQL, XSLT is declarative. SQL utilizes SELECT statements while XSLT utilizes XPath expressions (more on XPath in an upcoming section).
- It is a functional language and has no side effects (i.e., no variable updates take place).
- At runtime, the XSLT processor both constructs a tree representation of the source document and produces a tree representation of the results document.
- It is rule-based. Processing takes place when patterns in the stylesheet are matched with elements in the source document.
- With built-in functions, it is capable of performing selection, aggregation, and grouping as well as string and arithmetic data manipulation.

To perform an XSLT transformation, an XSLT processor (a.k.a. an XSLT engine) is required. An XSLT processor is the piece of software that reads both the source XML document and the XSLT stylesheet, performs the actual transformation according to the instructions in the stylesheet, and produces the resulting XML document. There are many open source and commercial XSLT processors available on the market. SAP also has its own XSLT processor implementation available as part of the SAP XML Toolkit for Java.[2]

An XSLT stylesheet document contains template rules. A template rule has a pattern specifying the nodes it matches, along with the template body to be instantiated and output when the pattern is matched. When an XSLT processor transforms an XML document using an XSL stylesheet, it walks through the source XML document tree, looking at each node in turn. As each node in the XML document is read, the processor compares it with the pattern of each template rule in the stylesheet. When the processor finds a template rule with a pattern that best matches the node, it instantiates and outputs the rule's body. The template rule's body generally includes some markup, some literal data, and some data copied from the source XML document.

XSLT Elements

XSLT defines many elements that can be used in the XSLT stylesheet.

The *<xsl:stylesheet>* element is the root element for XSLT stylesheets. It specifies the XSL namespace prefix as well as the version, as you can see in **Listing 1**.

The *<xsl:template>* element defines a template rule. The *match* attribute of the *<xsl:template>*

[2] Formerly known as the InQMy XML Toolkit.

Listing 1: The <xsl:stylesheet> Element Specifies the Namespace and Version

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/999/
  XSL/Transform">
…
</xsl:stylesheet>
```

Listing 2: The <xsl:template> Element Defines the Template Rule

```
<xsl:template match="BOMMAT03">
…
</xsl:template>
```

Listing 3: The <xsl:call-template> Element Calls the Template for Further Processing

```
<xsl:template name="convert-date">
   <xsl:param name="instring"/>
…
</xsl:template>
…
<xsl:call-template name="convert-date">
   <xsl:with-param name="instring" select="E1STZUM/DATUV/text()"/>
</xsl:call-template>
```

element contains a pattern against which nodes from the source tree are compared as they are processed (see **Listing 2**). If the pattern is the best match for a node, then the content of the template rule is instantiated and inserted into the results tree.

The *<xsl:apply-templates>* element causes the immediate children of the source element to be processed further recursively.

The *<xsl:template>* element can also be used to define a named template, which can be called by the *<xsl:call-template>* element. As the example in **Listing 3** shows, first, a named template *convert-date* with parameter *instring* is defined. Then the template is called with the actual parameter value taken from the source document.

An arbitrary element (not in the XSL namespace)

Listing 4: An Element Is Directly Inserted into the Results Tree

```
<Plant>
  <xsl:value-of select="E1STZUM/E1MASTM/WERKS/text()"/>
</Plant>
```

Listing 5: The <xsl:attribute> Element Adds an Attribute

```
<BillOfMaterial>
  <xsl:attribute name="MaterialNumber">
      <xsl:value-of select="E1STZUM/E1MASTM/MATNR/text()"/>
  </xsl:attribute>
  …
</BillOfMaterial>
```

will be inserted into the results tree directly, like the *<Plant>* element in the example shown in **Listing 4**. The *<xsl:value-of>* element extracts a specific value from the source XML document and copies it to the results document. The *select* attribute may be a full or relative (to the current node) path expression.

The *<xsl:attribute>* element adds an attribute with the given name to an element in the results tree. In the example shown in **Listing 5**, the resulting element will look like the following:

```
<BillOfMaterial MaterialNumber
  ="…">
```

The *<xsl:for-each>* element allows for looping through the nodes that match the *select* expression (see the example shown in **Listing 6**). For each node in the matching node set, the body of the *<xsl:for-each>* will be instantiated once.

The *<xsl:choose>* element, together with the *<xsl:when>* element, can be used for multiple condition tests. The example in **Listing 7** checks the value of the variable named *key*. If its value is *K*, then *Class item* will be chosen as the result; if its value is *L*, then *Stock item* will be chosen.

XPath

XSLT utilizes XPath to query the source tree, which is analogous to SQL's utilization of SELECT statements to retrieve data from a database. The following are examples of XPath expressions:

- The *match* expression of the *<xsl:template>* element
- The *select* expression of the *<xsl:value-of>* element
- The *select* expression of the *<xsl:with-param>* element
- The *select* expression of the *<xsl:for-each>* element

XPath is a W3C recommendation. XPath provides syntax for locating a node in an XML

Listing 6: The <xsl:for-each> Element Enables Looping

```
<xsl:for-each select="E1STZUM/E1STPOM">
  <xsl:call-template name="process-e1stpom"/>
</xsl:for-each>
```

Listing 7: The <xsl:choose> and <xsl:when> Elements for Condition Tests

```
<xsl:choose>
  <xsl:when test="$key = 'K'">Class item</xsl:when>
  <xsl:when test="$key = 'L'">Stock item</xsl:when>
</xsl:choose>
```

document tree. In XPath, a location path describes the location of something in an XML document. It takes its inspiration from the syntax used to denote paths in file systems such as Unix.

Everything you do in XPath is interpreted with respect to the *context* — a concept similar to that of the current directory in a file system. Most of the time, you can think of the context as the node in the tree from which an expression is evaluated. The context for an XPath expression is the current node being processed. For example, take a look at the following code sample:

```
<xsl:template match="para">
…
</xsl:template>
```

The context of the XPath expressions found in the body of this template rule will be the selected *para* elements.

Here is an example XPath expression:

```
E1STZUM/E1MASTM/WERKS/text()
```

This expression selects the text of the element *WERKS* in the element *E1MASTM* in the element *E1STZUM* in the current context node.

XSLT and XPath also define many functions[3] that can be used inside XPath expressions. These functions fall into the following categories:

- Data conversion
- String manipulation
- Node name/identifier
- Context-related
- Processor description
- Arithmetic
- Aggregation
- Node search

[3] The details of these functions are beyond the scope of this article. Please refer to the book *XSLT Programmer's Reference* by Michael Kay (ISBN 1861005067). In my opinion, this is one of the best XSLT books available.

So, now that you have a basic understanding of XSLT, let's roll up our sleeves and examine the details of mapping your IDoc data to the desired XML format using this technique.

Data Mapping Using XSLT Transformation

In the 4.6 Business Connector, the SAP XML Toolkit for Java is plugged in as the XSLT processor, which enables the BC to offer native XSLT support.

The service *pub.sap.xslt.transformation: XSLTransformation* takes as its input parameters an XML source, a stylesheet name, and optional global parameters to be used in the transformation. The service executes the necessary transformations based on the XSL stylesheet and produces the output plus any output parameters.

You can use the *xmlString* input parameter to pass the source XML string to the service. Alternatively, you can use the *xmlRecord* parameter to pass the source XML in a record structure, and use parameter *inRecordDefName* to specify the record definition to use for parsing the source XML record. With input parameter *stylesheetFilename*, you specify the file name of the stylesheet to use in the transformation. You can use either an absolute path starting from the root of the file system, or a relative path starting from the BC installation directory (e.g., *C:\sapbc46\server*).[4]

Input parameter *outRecordDefName* specifies the name of the record definition to use when building the resulting record structure. If provided, the result will be a record containing the transformed XML results in output parameter *transformedXmlRecord*. If *outRecordDefName* is not provided, then the results will be in the form of a string, in the output parameter *transformedXmlString*.

Using XSLT to perform data transformation in the BC has the following advantages:

- XSLT is the emerging[5] standard for XML transformations.
- No proprietary technologies are needed.
- The XSLT stylesheet is portable across different XSLT processors and platforms. This means that the stylesheet can be reused in completely different environments and your investments will not be lost.[6]

Potential disadvantages with XSLT-based data mapping include the following:

- The upfront learning curve associated with adopting XSLT can be steep if the XSLT knowledge is not readily available to you.
- The XSLT stylesheets have to be carefully designed and tested to optimize transformation performance. Otherwise, potential performance issues may arise.

Since its introduction in Business Connector 4.6, XSLT-based data mapping has been garnering more and more interest. For example, version 3.0 of the SAP MarketSet Connector, which connects mySAP backends with MarketSet-based marketplaces, uses mainly XSLT transformation for data mapping.

Mapping IDoc data to an XML format using XSLT transformation typically involves these steps:

1. Develop and test the XSLT stylesheet.

[4] Using the relative path is usually a good idea. You can put the XSLT stylesheets in the file system subfolder (e.g., the *resources* subfolder) of the corresponding BC package and use the relative path to address them. Thus you can achieve portability of the path.

[5] XSLT indeed is already a W3C recommendation. Many people consider it an established standard for XML transformation and I would agree with that sentiment. The term "emerging" is used here in a relative sense.

[6] This is one of the most attractive benefits of using XSLT transformation. For example, the new SAP Exchange Infrastructure (XI) supports XSLT transformation, which means your current XSLT stylesheets can be reused in future XI-based projects.

Listing 8: Command Line for Testing the XSLT Stylesheet with the SAP XSLT Processor

```
java -cp C:\sapbc46\Server\packages\SAP\code\jars\static\
  inqmyxml.jar;. com.inqmy.lib.xsl.xslt.CommandLine -xml <sourceXML>
  -xsl <stylesheet>
```

2. Create a BC service to execute the XSLT transformation.

Let's take a closer look at these two steps.

Step 1: Develop and Test the XSLT Stylesheet

Using XSLT to perform data transformation is quite easy, once you've done the work to develop the XSLT stylesheet to perform the desired transformation. This can be done outside the BC, even with the help of some third-party XSLT stylesheet design tools.[7]

To facilitate stylesheet development and testing, you can capture sample instances of the source XML document. For example, to capture a sample IDoc-XML document instance, follow these steps:

1. In the BC Administrator, select *Routing* → *Transactions*.

2. Click on the transaction ID (TID) of the relevant IDoc.

3. Right-click on *View As XML* and select *Open in New Window*.

4. In the new browser window, select *File* → *Save As* and then specify the file name for the captured sample.

Now you can focus on the development of the stylesheet. Using the sample source XML document that you captured (as described in the first article), you can also test the stylesheet outside the BC by executing the XSLT transformation and then checking the resulting XML document. The XSLT processor delivered with the 4.6 BC can be executed standalone,[8] using a command line like **Listing 8**.

The <*sourceXML*> element in Listing 8 is the source XML document file name, and <*stylesheet*> is the XSLT stylesheet file name.

Step 2: Create a BC Service to Execute the XSLT Transformation

Once you have developed and tested the XSLT stylesheet, you can create a BC service to execute the XSLT transformation as follows:

1. Prepare the source XML to be used as the input to the transformation service.

[7] XML Spy, by Altova, is one of the best XML IDEs available. It provides an XSL editor and debugger, among other features.

[8] You can also use other XSLT processors. However, since the SAP XSLT processor will be used by the BC to execute the XSLT transformation at runtime, it is a good idea to also use it to test your stylesheet at design time.

Listing 9: An Example Template Rule for Handling Multiple IDOC Elements

```
<xsl:template match="IDOC">
  <xsl:call-template name="process-bommat03idoc"/>
</xsl:template>
```

2. Invoke the transformation service, specifying the source XML and the XSLT stylesheet file names.

Depending on the desired input XML format of the XSLT transformation service, you need to prepare the source XML accordingly. Let's assume that you want to transform an SAP IDoc-XML document into another XML format, and that you have developed the XSLT stylesheet to take an IDoc-XML document as input. Before invoking the XSLT transformation with the stylesheet, you need to prepare the IDoc-XML document. The built-in BC service *pub.sap.idoc:encode* can be used for this purpose. It converts a flat-structured IDoc to an XML string in accordance to the SAP IDoc-XML specification.

You can then feed the output argument *xmlData* of the *pub.sap.idoc:encode* service as source XML to the XSLT transformation service *pub.sap.xslt.transformation:XSLTransformation*, and specify the XSLT stylesheet file name.

After the execution, the resulting XML is available in the pipeline. You can then retrieve the resulting XML and transmit it to the desired destination.

Additional Considerations for XSLT Transformation

When you design the XSLT stylesheet for the data mapping, you often need to answer these questions:

- How can I handle an IDoc packet with XSLT?
- How can I perform value lookups with XSLT?

I will address these considerations next.

Handling an IDoc Packet with XSLT

When performing mapping with XSLT, you must also take into account that in some cases the SAP system will send multiple IDocs in one packet. In this case, the XML document produced by the *pub.sap.idoc:encode* service will contain multiple *IDOC* elements, and the XSLT stylesheet will need to be designed to handle these multiple *IDOC* elements.

This can be easily done in XSLT using a template rule similar to the one shown in **Listing 9**. Here, you can see that for each occurrence of the *IDOC* element, the named template *process-bommat03idoc* will be executed once. Within the template, you only need to process a single *IDOC* element.

Performing Value Lookups with XSLT

There are two options when it comes to performing value lookups in XSLT transformation: you can use <*xsl:choose*> and <*xsl:when*> elements, or you can use Java extension functions.

The <*xsl:choose*> and <*xsl:when*> Elements

The <*xsl:choose*> element, together with the <*xsl:when*> element, can be used for multiple condition tests, as discussed earlier. If the possible entries for the value lookup are limited, you can build the value lookup into the XSLT stylesheet, as you saw back in Listing 7 (shown again in **Listing 10** for your convenience).

Obviously, this only works with a small number of possible lookup entries. It requires modifications to the XSLT stylesheet whenever the lookup entries need to be changed, and it will be tedious to write the stylesheet for a large lookup.

Java Extension Functions

The current quasi-standard of XSLT, the W3C XSLT 1.1 working draft, defines a mechanism for the XSLT processor to support invoking extension functions from the XSLT stylesheet. The extension functions can be written in a number of languages, with Java being a popular choice. The XSLT processor used in the 4.6 BC, as part of the SAP XML Toolkit for Java, supports this extension function mechanism.

You can first write the Java code separately from the BC, performing value lookup using a text file or database table. After compiling the Java code and making the Java classes accessible to the BC's XSLT processor, you can invoke these Java extension functions from within the XSLT stylesheet during the XSLT transformation.

In the first article, I showed you a Java code sample using a properties file for value lookup. Now let's look at some sample code for performing database lookup using JDBC (**Listing 11**).

Listing 10: Building the Value Lookup into the XSLT Stylesheet

```
<xsl:choose>
  <xsl:when test="$key = 'K'">Class item</xsl:when>
  <xsl:when test="$key = 'L'">Stock item</xsl:when>
</xsl:choose>
```

Listing 11: Database Lookup via JDBC

```
 1 package com.spj.sbcarticle;
 2
 3 import java.sql.*;
 4
 5 public class Lookup {
 6   public static String lookup(String key) {
 7     String value = "not found";
 8
 9     // use JDBC to lookup value
```

(continued on next page)

(continued from previous page)

```
10     try {
11       Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
12       Connection con =
           DriverManager.getConnection("jdbc:odbc:SbcLookup", "", "");
13       Statement stmt = con.createStatement();
14       String queryStr = "select value from SbcLookupTable where
           key = '" + key + "'";
15       ResultSet rs = stmt.executeQuery(queryStr);
16       if (rs != null) {
17         rs.next();
18         value = rs.getString("value");
19       }
20       rs.close();
21       stmt.close();
22       con.close();
23     } catch (Exception ex) {
24       ex.printStackTrace();
25     }
26     return value;
27   }
28 }
```

Let's take a closer look at what's happening in Listing 11. (Note that I've added line numbers for your convenience.)

As you can see, the Java class is called *Lookup*, and it resides in the *com.spj.sbcarticle* package. First we import the necessary Java packages (line 3), including *java.sql* for JDBC. In the class, we define a *static* method called *lookup()*, which takes a string argument *key*, and then returns a string (line 6). Within the *try* block of the method, we first load the JDBC driver class — *sun.jdbc.odbc.JdbcOdbcDriver* in this case (line 11). You can use any supported JDBC driver for your database, as long as it is accessible in the classpath. We then use the *getConnection()* factory method of the class *DriverManager*, specifying the connection properties as arguments, to instantiate a *Connection* object named *con* (line 12). Then the *createStatement()* method of the *Connection* object is invoked to instantiate a *Statement* object *stmt* (line 13). We then compose the SQL query string by concatenating the method argument *key* to the SQL *select* statement (line 14). Next, we execute the SQL query by invoking the *executeQuery()* method of the *Statement* object, passing the query string as the argument. The results will be in a *ResultSet* named *rs* (line 15). If the database table lookup is successful, there should be one row in the *ResultSet* object; otherwise, the *ResultSet* object will be *null*. So, we first check that there is data returned as the result of the query (line 16), then we move the cursor to the first row of the *ResultSet* (line 17) and retrieve the value of the appropriate field (in this case *value*) as a string (line 18), and finally return it as the result of the *lookup()* method (line 26). In case the SQL query did not return anything, we simply return the default value *not found* as the result of the method (lines 7 and 26).

After compiling and testing this JDBC-based lookup class, you can create a JAR file containing the

Listing 12: Calling an Extension Function in XSLT

```
<xsl:stylesheet … xmlns:lookup="java:com.spj.sbcarticle.Lookup">
  <xsl:script implements-prefix="lookup" language="java"
      src="java:com.spj.sbcarticle.Lookup"/>
  …
    <xsl:value-of select="lookup:lookup(string($key))"/>
  …
</xsl:stylesheet>
```

class and all the necessary supporting classes. This JAR file then must be placed in the directory *<sbcserver>\packages\SAP\code\jars* (e.g., *C:\sapbc46\Server\packages\SAP\code\jars*).[9]

After restarting the BC server, the JAR file will be loaded automatically and the Java classes and methods contained within will be accessible to the BC's built-in XSLT processor.

Now we can follow the XSLT 1.1 extension function rules to invoke the extension functions from the XSLT stylesheet (see **Listing 12**).

First, we declare an XML namespace *java:com.spj.sbcarticle.Lookup* with the namespace prefix *lookup*. In the *<xsl:script>* element, we indicate that the namespace *java:com.spj.sbcarticle.Lookup* associated with the namespace prefix *lookup* is actually implemented by Java extension functions contained in Java class *com.spj.sbcarticle.Lookup*. In the *select* expression of the *<xsl:value-of>* element, we invoke the Java extension function, calling the static method *lookup()* in the *com.spj.sbcarticle.Lookup* class, passing the current value of the XSLT variable *key* as the argument. The return value of the *lookup()* method will then become the result of the *<xsl:value>* element, as part of the XSLT transformation result.

Using the XSLT extension function mechanism invoking Java extension functions, among other things, allows you to implement value lookup using Java code accessing a text file or a database table. This gives you the flexibility to change the lookup entries without touching the stylesheet itself. Also, the database-table-based lookup approach scales well and offers good performance.

An Example XSLT Transformation

The BC service *spjexamples.idoc.outbound:processBommat03PacketWithXslt* contained in the *SPJ* package (which is included in the download available at **www.SAPpro.com**) is provided as an example of data mapping with XSLT transformation. It uses the XSLT stylesheet file *Bommat03Packet_ExtBomXml.xslt*, which resides in the *resource* directory of the *SPJ* package (e.g., *C:\sapbc46\Server\packages\SPJ\resources*).

The source code of a Java extension function is provided in file *com\spj\sbcarticle\Lookup.java*. The compiled classes are in JAR file *SpjSbcArticle.jar*. To use the Java extension functions from the XSLT

[9] All the JARs in this directory will be loaded by the BC during startup, so the SAP XSLT processor will have access to the Java extension functions in the JAR file during XSLT transformation. You have to restart the BC server to have it load the new JAR.

stylesheet, the JAR file needs to be copied to the *code\jars* directory of the *SAP* package (e.g., *C:\sapbc46\Server\packages\SAP\code\jars*). Then the BC server must be restarted to load the JAR file.

To run the example, you can follow the same steps for the BC flow service example (*spjexamples.idoc.outbound:processBommat03PacketWithFlow*) described in the first article. Note that you will need to modify the routing rule to route the IDocs to the new service for XSLT.

Data Mapping Using a Java Service

Java is the native language of the SAP Business Connector. The BC provides a Java API that you can use in your Java services to perform many tasks (e.g., pipeline manipulation, service invocation, etc.). In addition, all the standard and optional JDK APIs, as well as third-party Java APIs, can be used. The BC also provides an IDoc-Java API[10] that enables the processing of SAP IDocs. You can build your own Java service to perform the data transformation to extract data from the source and compose the resulting format as desired.

Performing data transformation in Java has the following advantages:

- It is potentially faster.
- It can handle very complex logic and heavy computation with relative ease.
- It can utilize the BC Java API and the IDoc-Java API, as well as other third-party Java APIs.

However, it also has the following disadvantages:

- Coding in Java requires Java programming expertise.
- The IDoc-Java API supports only a single IDoc; it does not support IDoc packets.

If you need to process multiple IDocs in a packet, there are workarounds available to separate the individual IDocs from the IDoc packet, so they can be processed by the Java service using the BC's IDoc-Java API. However, these workarounds typically add overhead and are not efficient in terms of performance. That means that Java services based on the IDoc-Java API are mainly suitable only for processing single IDocs, not IDoc packets. If you need to process IDoc packets and want to use Java code for some complex computation, usually it will be better, in terms of performance, to use a BC flow service and only invoke custom Java services when necessary (the BC flow service technique was described in detail in the first article, "XML Messaging with the SAP Business Connector Part 1: Direct IDoc-XML Data Exchange and Outbound IDoc-to-XML Data Mapping with Flow Services"). Alternatively, you could mainly use XSLT transformation, discussed in the previous section, and invoke Java extension functions only when necessary.

Of course in BC projects, a Java service approach is still worth considering if only single IDocs are involved and performance is a high priority.

A Java service that processes a single IDoc typically has the following code blocks:

1. Instantiate the *IDOC* Java object based on the data of a single IDoc in the pipeline.
2. Find the relevant IDoc segments and retrieve the IDoc field values of interest.

Listing 13: Instantiate a New IDoc Instance

```
Values idoc_pipeline = in;
IDOC idoc = new IDOC(idoc_pipeline);
```

[10] This is the IDoc-Java API for Java services running on the BC. Do not confuse it with the new SAP Java IDoc Class Library, based on the Java Connector (JCo), which is still in beta at the time of writing.

Listing 14: Access IDoc Segments and Fields

```
IDOCSegment segE1STZUM = null;
segE1STZUM = idoc.findSegment("E1STZUM");
category = segE1STZUM.getSDATA("STLTY");
```

Listing 15: Perform Value Transformation

```
strDATUV = segE1STKOM.getSDATA("DATUV");
validFromDate = strDATUV.substring(2, 4) + "-" + strDATUV.substring(4, 6) +
  "-" + strDATUV.substring(6, 8);
```

3. Compose the resulting data format using the data retrieved from the IDoc and generate the XML document.

4. Transmit the resulting XML document to the target system.

Note that the BC's IDoc-Java API requires the Java service to be created using the old *Values* signature.[11] You can change the Java service signature preference in the BC Developer. Select *Edit* → *Preferences* → *Services*, and then select the desired Java service signature in the *Default Java service signature* section.

I will now discuss the important segments of a sample Java service that processes a single *BOMMAT03* IDoc instance, maps it to an external XML document, and transmits the results to the target system. The complete example service is contained in the *SPJ* download package.

First we make a copy of the pipeline input, and then we use the copy as input to the constructor of the *IDOC* class in order to instantiate a new *IDOC* instance using the data contained in the pipeline (see **Listing 13**).

> ✓ ***Note!***
>
> *Since we will be using the BC IDoc-Java API here, we need to specify com.wm.pkg.sap.idoc.* as an imported package in the Imports section of the Java service's Shared tab, together with all the other Java packages/classes that need to be imported.*

In **Listing 14**, we declare an *IDOCSegment* instance variable named *segE1STZUM*. Then we use the *findSegment(String segmentName)* method of the *IDOC* class to find the first occurrence of the segment *E1STZUM*. After finding the segment, we access the field values by using the *getSDATA(String fieldName)* method of the *IDOCSegment* class.

In **Listing 15** we perform a value transformation for the date field. The *DATUV* field in the *E1STKOM* segment contains the date in the *yyyymmdd* format. We want to change the format to *yy-mm-dd*, so we are

[11] The SAP BC supports two Java service signatures: *Values* and *IData*. The SAP BC 4.6 uses *IData* by default.

Listing 16: Process All Segments with a "while" Loop

```
IDOCSegment segE1STPOM = null;
Vector vecComponent = new Vector();
int index = 0;
while ((segE1STPOM = idoc.findSegment("E1STPOM", index)) != null) {
  index = segE1STPOM.getIndex() + 1;
  …
}
```

Listing 17: Compose the Record Object

```
componentItemNumber = segE1STPOM.getSDATA("POSNR");
…
Values valComponent = new Values();
valComponent.put("@itemnumber", componentItemNumber);
…
vecComponent.add(valComponent);
```

using the *substring()* method of the *String* class to extract the two-digit year, month, and day values, and then concatenate them with "-" as the separator.

In the IDoc, the *E1STPOM* segment may occur multiple times. To process all the *E1STPOM* segments, we use a *while* loop in **Listing 16**. First we set the integer variable *index* to *0*, so *idoc.findSegment("E1STPOM", index)* will find and return the first instance of the *E1STPOM* segment. After processing the first instance, we set *index* to the value that points to the first segment after the current *E1STPOM* segment, and repeat *findSegment()* to find the next occurrence. This process is repeated until all the *E1STPOM* segments are found and processed (i.e., when *findSegment()* returns *null*).

Within each loop iteration, we use the *getSDATA()* method of the segment object to retrieve the field values we are interested in and compose a record object called *valComponent* (see **Listing 17**). We then add the record object to the vector *vecComponent* instantiated earlier (refer back to Listing 16) before we enter the *while* loop.

After the *while()* loop, the vector *vecComponent* will collect all the *valComponent* record objects created, each mapped from one occurrence of the *E1STPOM* segment. We use *Vector* here because we do not know how many *E1STPOM* segments there will be at runtime.

In **Listing 18**, we convert the vector of *Values* objects to an array of *Values* objects (i.e., a record list).

Now, with all the data available, we can assemble a *Values* object in the desired record format (see **Listing 19**).

We then use the *pub.web:recordToDocument* built-in service to convert the record to an XML string (**Listing 20**). By using the *Values* object and

Listing 18: Convert the Vector to a Record List

```
int size = vecComponent.size();
Values[] component = new Values[size];
for (int i=0; i < size; i++) {
  component[i] = (Values) vecComponent.elementAt(i);
}
```

Listing 19: Format the Values Object

```
Values valBillOfMaterial = new Values();
valBillOfMaterial.put("@MaterialNumber", baseMaterialNumber);
valBillOfMaterial.put("Category", category);
valBillOfMaterial.put("Components", components);
…
Values valXML = new Values();
valXML.put("BillOfMaterial", valBillOfMaterial);
```

Listing 20: Convert the Record to an XML String

```
in.put("boundNode", valXML);
Service.doInvoke("pub.web", "recordToDocument", in);
String xmlString = in.getString("xmldata");
```

Listing 21: Initiate the HTTP POST

```
in.put("url", "http://localhost:4242");
in.put("method", "POST");
Values data = new Values();
data.put("string", xmlString);
in.put("data", data);
Values headers = new Values();
headers.put("Content-type", "text/xml");
in.put("headers", headers);
Service.doInvoke("pub.client", "http", in);
```

the *recordToDocument* service, we can create XML data easily without using the rather complex Document Object Model (DOM) API.

In **Listing 21**, we initiate the HTTP POST using the *pub.client:http* built-in service. Please note that we set the *Content-type* HTTP header to *text/xml*.

> ✓ ***Performing Value Lookup in a Java Service***
>
> *Since we are programming in Java, performing value lookup when using a Java service is quite easy. You can write Java code that performs the text-file-based lookup, or JDBC database-table-based lookup, as static methods defined in the Shared Source section of the Java service's Shared tab. These static methods can then be invoked from the code of the Java service.*

An Example Java Service

The BC service *spjexamples.idoc.outbound: processBommat03SingleWithJava*, available with the download, is an example Java service that maps a single *BOMMAT03* IDoc to an external XML document and transfers the XML document to the external HTTP server using HTTP POST.

You can use transaction *BD30* to send a single *BOMMAT* IDoc from the SAP system and configure the BC routing rule to route the IDoc to the Java service.

Alternatively, a captured pipeline containing a single *BOMMAT03* IDoc instance is provided as file *Bommat03SinglePipeline.xml*. You can use the *spjexamples.idoc.outbound: testProcessBommat03SingleWithJava* flow service to restore the pipeline from the file and test the Java service.

Inbound IDoc Communication Using Non-SAP XML

With everything you have learned so far in this two-part discussion, you are now familiar with the concept of outbound IDoc communication using non-SAP XML, and the various considerations involved in this process, including the different data-mapping options available to you. Armed with this information, let's now examine *inbound* IDoc communication using non-SAP XML, where the document travels from the business partner's system, to the Business Connector, and then to your SAP system.

There are three main tasks involved in inbound IDoc communication when using non-SAP XML:

- Send an external XML document to the BC service.
- Map the data from the XML document to an IDoc.
- Post the IDoc to the SAP system.

Let's take a look at each of these tasks in turn.

Send an External XML Document to the BC Service

The Business Connector provides the following automated mechanisms for receiving arbitrary XML documents, parsing them, and passing them as input to a specified service:

- A client can submit an XML document to a service in a string variable of any name.
- A client can submit an XML document to a service in a special string variable named *$xmldata*. The BC will automatically parse the XML string into a node (a parsed XML document) and pass the node to the service.
- A client can post an XML document to a service via HTTP. The BC will automatically parse the XML document into a node (a parsed XML document) and pass the node to the service. This is

Figure 1 Built-In Services for Incoming XML Documents

Built-In Service	Description
pub.web:stringToDocument	Converts a text string representing an XML document into a node object (a parsed XML document).
pub.web:documentToRecord	Converts a node object (a parsed XML document) to a Values object (a record variable). This service transforms each element and attribute in the source document to an element in a Values object.

probably the most practical option. The HTTP header *Content-type* needs to be set to *text/xml*. The XML document is sent in the HTTP request body.

- A client can FTP an XML document to a service.
- A client can email an XML document to a service.

The BC provides several built-in services that can be used to deal with the incoming XML document (see **Figure 1**).

The BC Developer can provide some help in creating flow services that receive an XML document. The starter flow *Receive an XML Document* can be used by providing a sample instance of the XML document, DTD, or XML schema of the XML document. The generated flow and record definition can then be used as a starting point for further developing the flow.

Within the flow, you can also perform data validation of the received XML data. There are mainly two types of data validations as far as the incoming XML data is concerned: XML validation and record validation.

XML Validation

XML validation verifies the structure and content of the XML document against a BC schema. A BC schema is a freestanding element in the BC that acts as a blueprint or model against which you validate an XML document. It can be generated from a DTD, an XML schema definition, or an XML document that references an existing DTD.

To perform XML validation, the *pub.schema:validate* service needs to be invoked with the following parameters:

- *object* — The node object (the parsed XML document) to be validated
- *conformsTo* — The fully qualified name of the BC schema that you want to validate against

Record Validation

Record validation verifies the structure and content of an individual record variable in the pipeline against a BC record definition.

This can be useful if the incoming XML document is already converted to a record variable by *pub.web:documentToRecord.*

To perform record validation, the *pub.schema:validate* service needs to be invoked with the following parameters:

- *object* — The record variable to be validated
- *conformsTo* — The fully qualified name of the record definition that you want to validate against

> ✓ ***Note!***
>
> *Before we discuss the options for mapping an external XML document to an IDoc, let's first look at the IDoc format required for posting to the SAP system. The easiest way to post IDocs to the SAP system from a BC service is to use the built-in pub.sap.transport.ALE:OutboundProcess service. This service expects the IDoc data in the flat structure format — IDOC_CONTROL and IDOC_DATA for version 2 IDocs, or IDOC_CONTROL_REC_40 and IDOC_DATA_REC_40 for version 3 IDocs. The service supports posting of both single IDocs and IDoc packets.*

Map the Data from the XML Document to an IDoc

Since the *pub.sap.transport.ALE:OutboundProcess* service expects the IDoc data in the flat structure format, that is the target IDoc format you must map the external XML document to. This mapping is the reverse of mapping from the IDoc to the external XML format (detailed in the discussion on outbound IDoc communication with non-SAP XML). The techniques used for the data mapping are the same, so in the following sections, I will point out the few things that need special attention for inbound XML-to-IDoc mapping.

Again, you can use a BC flow service, XSLT transformation, or a Java service for inbound mapping, just as with outbound mapping. With inbound mapping, each technique also has the same pros and cons that it had with outbound mapping.

Data Mapping Using a BC Flow Service

The following are the main steps necessary for using a flow service:

1. Receive the external XML data as a node (a parsed XML document).

2. Invoke *pub.web:documentToRecord* to convert the node to a record and bind the record definition for the external XML document to it (to make it expandable for mapping in subsequent steps).

3. Map the external data to a record that corresponds to the hierarchical structure of the IDoc, using MAP operations with transformers, LOOP operations, etc., as necessary.

4. Invoke *pub.sap.idoc:transformHierarchyToFlat* to transform the IDoc to a flat structure.

5. The flat-structured IDoc is then ready to be posted to the SAP system.

An IDoc packet can be handled easily in the flow mapping. If there are multiple IDocs that need to be posted to the SAP system in an IDoc packet, the hierarchically structured target variable of the mapping will contain multiple *IDOC* elements. The LOOP operation can be used to collect the multiple *IDOC* elements, one *IDOC* element at a time, within one loop iteration.

Data Mapping Using XSLT Transformation

The main steps necessary for using XSLT transformation to perform the inbound mapping are:

1. Receive the external XML document as a string.

2. Transform the external XML document to an IDoc-XML document by invoking the service *pub.sap.xslt.transformation:XSLTransformation* and providing the XSLT stylesheet file name. The XSLT stylesheet must be developed and tested outside the BC, possibly by using some third-party XSLT editor tool.

3. Invoke the service *pub.sap.idoc:decode* to convert the IDoc-XML document to a flat-structured IDoc.

Listing 22: Get the Control Record and Set the Control Fields

```
IDOCControl ctrl = idoc.getControlRecord();
ctrl.setField("DIRECT", "2");
…
```

Listing 23: Create and Populate the Data Segments

```
IDOCSegment segE1ALER1 = new IDOCSegment("E1ALER1");
segE1ALER1.setSDATA("MESTYP", "MATMAS");
…
segE1ALEQ1.setSDATA("LOW", materialNumber);
…
```

4. The flat-structured IDoc is then ready to be posted to the SAP system.

An IDoc packet can be handled easily using XSLT transformation. If there are multiple IDocs that need to be posted to the SAP system in an IDoc packet, the XSLT stylesheet can be developed accordingly to produce multiple *IDOC* elements in the resulting IDoc-XML document. After *pub.sap.idoc:decode*, these multiple IDocs will be packaged together and can then be posted to the SAP system.

Data Mapping Using a Java Service

The BC's IDoc-Java API can also be used to compose an IDoc based on data provided in the external XML document. Again, this API supports only single IDocs. IDoc packets are not directly supported.

We can develop the Java service to take a node (a parsed XML document) as input. In the Java service, we can invoke the *pub.web:documentToRecord* service to convert the node to a record. We can then easily access the data in the record by using the methods of the *Values* class. This is easier than parsing the XML document using the standard DOM or SAX APIs for XML.

With the data from the external XML document available, we can start constructing the IDoc.

The following code instantiates a new version 3 *IDOC* object called *idoc*:

```
IDOC idoc = new IDOC(true);
```

We can then get the control record of the IDoc and set the necessary control fields, as shown in **Listing 22**.

In **Listing 23**, we create the data segments. The *setSDATA(String fieldName, String value)* method of the *IDOCSegment* class can be used to set the data field values. The value may be a hard-coded value, may come from the external XML document, or may be some intermediate computation result.

Listing 24: Assemble the IDoc from Data Segments

```
idoc.append(segE1ALER1);
idoc.insert("E1ALER1", segE1ALEQ1, true);
```

Now we can assemble the IDoc by putting all its data segments together in the correct hierarchical order, as shown in **Listing 24**.

The *getValues()* method of the *IDOC* class returns the IDoc in the flat structure, which is now ready for posting to the SAP system:

```
Values pipelineIDOC =
  idoc.getValues();
```

> ✓ ***Value Lookup for Inbound Data Mapping***
>
> *Value lookup for inbound data mapping can be done using the same techniques detailed in the outbound data-mapping discussion.*

Post the IDoc to the SAP System

As noted on page 96, the built-in service *pub.sap.transport.ALE:OutboundProcess* is the easiest way to post the flat-structured IDoc to the SAP system.

The input parameter *transportParams/server* contains the name of the SAP server entry (defined using the BC Administrator via *Adapters* → *SAP* → *SAP Servers*) that should be used as the target for the IDoc posting. Since different SAP systems are typically used for development, testing, and production, it is not a good idea to hard-code the target SAP system in the BC service. Instead, you can put this information in a configuration file and dynamically look up the SAP server entry to be used at runtime.

Suppose you have a configuration file containing the following data:

```
sapdest = SAPSystemToUse
```

You can then create a simple Java service for configuration lookup. It will be quite similar to the Java service discussed for file-based value lookup in the first article of this two-part discussion. This configuration lookup Java service can then be used in your main service, before the invocation of the *pub.sap.transport.ALE:OutboundProcess* service, to determine the target SAP server to be used for the IDoc posting.

Guaranteed Delivery

The *pub.sap.transport.ALE:OutboundProcess* service has an optional input parameter called *$tid*. If provided, it will be used as the TID for the tRFC posting of the IDoc to the SAP system. If not provided, the *pub.sap.transport.ALE:OutboundProcess* service will create a new TID and use it for the tRFC posting.

When the SAP system receives a tRFC IDoc posting, it will first check the TID value received against its TID table. If the TID is new, the IDoc posting will be processed. If the TID is marked as

previously processed, which indicates that it is a duplicate, the IDoc posting will be discarded.

Depending on the business arrangement between your company and your business partner, there are two main scenarios for inbound guaranteed delivery:

- **Your business partner's system is responsible:** In this case, your business partner's system generates a TID (a globally unique 24-character string) using their own routine, then attaches the TID to the XML transmission to your BC (e.g., as an HTTP header field of the HTTP POST). Your BC service then uses this TID as the *$tid* parameter value of the *pub.sap.transport.ALE:OutboundProcess* invocation. If there is any error during the processing (from the business partner's system, to the BC, and then to SAP), the business partner's system will be notified of the failure and should retransmit the XML document with the exact same TID. Thus this TID protects the entire end-to-end communication, achieving guaranteed delivery and avoiding duplication.

- **Your BC service is responsible:** In this case, your business partner's system will consider the transmission successful once the XML document is completely transmitted to the BC. It is then your BC service's responsibility to make sure that upon receiving the external XML document, the IDoc will be posted to the SAP system once and only once. To achieve this:

  - The BC service that receives the incoming XML document needs to confirm the success of the transmission with the business partner's system, once the XML document is fully received and stored in a persistent storage medium (e.g., a message queue, a database table, a file, etc.).

  - If there is any problem during the data mapping, the BC service that performs the mapping should log the error to the persistent storage medium and notify the administrator (via email, for example) for troubleshooting.

  - The BC service must be designed to automatically repeat the *pub.sap.transport.ALE:OutboundProcess* posting in case of transient network or SAP server problems. The reposting must use the exact same TID as the original posting attempt.

In this scenario, depending on the requirements, third-party queuing software may be useful for persistently storing the incoming XML document, the result of the data mapping, and the status of the IDoc posting.

Examples of Inbound XML-to-IDoc Mapping

The file *MaterialDataRequestSample1.xml* is provided with the download as a sample external XML document. Its DTD is also provided as *MaterialDataRequest.dtd*.

To post the IDoc to the SAP system, the target SAP system needs to be defined in the BC Administrator on the *SAP Servers* page (*Adapters* → *SAP* → *SAP Servers*). The SAP server entry can be defined with an arbitrary name. After that, you need to indicate which SAP server should be used by the example services by changing the value of the *sapdest* entry in the *config.txt* file, which resides in the *resources* directory of the *SPJ* package (e.g., *C:\sapbc46\Server\packages\SPJ\resources*).

An Example BC Flow Service

The BC service *spjexamples.xml.receiving:Xml2IdocWithFlow* is an example of inbound XML-to-IDoc mapping with a flow. It takes a parsed XML document (node) as input, maps the external XML document to the IDoc type *ALEREQ01*, and posts it to the SAP system specified. You can use the BC Developer menu path

Helpful Hints

The following is a quick reference of the helpful hints presented throughout the discussions in this article.

Data Mapping Using XSLT Transformation

- ☑ The 4.6 SAP Business Connector delivers a built-in XSLT engine and the built-in service *pub.sap.xslt.transformation:XSLTransformation.*
- ☑ Capture sample source and target XML documents for XSLT stylesheet development and testing.
- ☑ Use a third-party XSLT editor to improve the efficiency and productivity of stylesheet development.
- ☑ Use XSLT Java extension functions for complex logic, heavy computation, or accessing other Java APIs. It can also be used to perform file-based or database-table-based value lookups.
- ☑ Test the XSLT stylesheet directly with the BC's built-in XSLT processor by running it directly from the command line.
- ☑ The services *pub.sap.idoc:encode* and *pub.sap.idoc:decode* can be used to encode/decode an IDoc-XML document.

Data Mapping Using a Java Service

- ☑ Java code can easily handle complex logic or heavy computation, as well as access other Java APIs.

Test → Send XML file to send the sample XML document to this BC service. In the SAP system, transaction *WE05* can be used to display the posted IDoc.[12]

[12] You may see that the IDoc has an error status code in your SAP system. This means the IDoc is successfully posted to your SAP system, but your SAP system is not configured to correctly process the incoming IDoc. Ask your local ALE guru for help in getting the IDoc processed by the application.

An Example XSLT Transformation

The BC service *spjexamples.xml.receiving:Xml2IdocWithXslt* is an example of inbound XML-to-IDoc mapping with XSLT transformation. It uses the stylesheet file *MaterialDataRequest_Alereq01.xslt*, which resides in the *resources* directory of the *SPJ* package, (e.g.,

- ☑ Java-service-only approaches are appropriate only for single IDocs.
- ☑ For IDoc packets, a flow/Java combination or XSLT/Java combination can be used if necessary.

Inbound IDoc Communication Using Non-SAP XML

- ☑ The easiest way to post IDocs to the SAP system from a BC service is to use the built-in service *pub.sap.transport.ALE:OutboundProcess*.
- ☑ The service *pub.sap.transport.ALE:OutboundProcess* expects the IDoc data in the flat structure format. This is the target format of the BC service data mapping.

Data Mapping Using a BC Flow Service (Inbound)

- ☑ For incoming XML documents, use the *pub.web:documentToRecord* service to convert the parsed XML document to a record, and bind the record definition to it to make it "mappable."
- ☑ For inbound IDocs, *transformHierarchyToFlat* can be used to convert the IDoc from a hierarchical structure to a flat structure.

Guaranteed Delivery (Inbound)

- ☑ The guaranteed delivery of inbound communication can be the responsibility of the business partner's system or the responsibility of your BC service.
- ☑ Inbound guaranteed delivery by BC services is more complex. Third-party queuing software may be useful in these cases.

C:\sapbc46\Server\packages\SPJ\resources). This service takes the XML document as input in its string input parameter *in_xml_string*, maps the external XML document to the IDoc type *ALEREQ01*, and posts it to the SAP system specified. To test it, you can use the BC Developer menu path *Test* → *Run*, and copy and paste the contents of the external XML document to the *in_xml_string* argument.

An Example Java Service

The BC service *spjexamples.xml.receiving:Xml2IdocWithJava* is an example of inbound XML-to-IDoc mapping with a Java service. It takes a parsed XML document (node) as input, maps the external XML document to the IDoc type *ALEREQ01*, and posts it to the SAP system

specified. To test it, you can use the BC Developer menu path *Test → Send XML file* to send the sample XML document to the service *spjexamples.xml.receiving:testXml2IdocWithJava*, which will in turn execute the service *spjexamples.xml.receiving:Xml2IdocWithJava* .

Conclusion

In the first article of this two-part discussion ("XML Messaging with the SAP Business Connector Part 1: Direct IDoc-XML Data Exchange and Outbound IDoc-to-XML Data Mapping with Flow Services"), you learned how to perform direct IDoc-XML messaging with business partners, how to use a BC flow service for outbound non-SAP XML messaging with IDocs, and when to use which. Here in the second part, I discussed the other two options for outbound non-SAP XML messaging — XSLT transformation and Java services — including pointers on the best situations to use each. I also outlined the special considerations involved in inbound non-SAP XML messaging with IDocs.

I hope the information provided in these articles has provided you with the "big picture," as well as enough detail on XML messaging with the SAP Business Connector to get you started, and has enabled you to take advantage of the SAP Business Connector in your own IDoc-based XML messaging projects. Have fun!

Robert Chu joined SAP at the end of 1996. He currently works for the Integration and Certification Center at SAP Labs in Palo Alto, California. Prior to this, he worked in technical consulting and training at SAP America and SAP Asia. Robert's current focus is the SAP integration technologies. He has been regularly teaching classes in this area at SAP training centers and is the main author of the BIT531 training course, as well as a few other internal workshops. Robert has spoken at the past three SAP TechEd events. In addition to his SAP expertise, he is also an SCEA (Sun Certified Enterprise Architect) for J2EE, an MCSD (Microsoft Certified Solution Developer), and an MCSE (Microsoft Certified System Engineer). Robert can be reached at robert.chu@sap.com.

