Case study: Invoice processing with IBM Business Process Manager, Part 1: The business context and process model

Scott Glen

February 28, 2013

This two-part article provides a case study of a real-world IBM® Business Process Manager V8.0.1 invoice processing solution, using web services and Java™ integration to automate a process spanning multiple vendors and operational systems. Part 1 describes the business problem and the process and associated data structures used in the solution. Part 2 will continue the analysis, investigating the Java and web service integrations, as well as the image manipulation techniques involved in the solution.

View more content in this series

Business challenge

At a conceptual level, invoice processing is a fairly straightforward procedure, typically encompassing the following steps:

1. An invoice, which usually relates to the purchase of one or more items, is received by an Accounts Payable department.
2. The invoice is examined, validated, classified and typically associated with a specific individual or department within an organization.
3. If it relates to a purchase, then the invoice must be matched against the associated Purchase Order (PO).
4. Depending on the value of the invoice, it may be sent to the relevant individuals for approval.
5. Finally, the invoice is posted to the accounting system to record the completion of the transaction.

However, in today's complex business environments this process can span multiple technologies, individuals, departments and even organizations. It can be a time-consuming and labor-intensive procedure, exacerbated by the drive to outsource operations across multiple vendors and business partners. Automated solutions that support these processes have existed for many years, but can often require the organization to adapt their process to suit the capabilities provided by the technology.

This article presents a case study of a recent Business Solution Services (BSS) engagement, which used IBM Business Process Manager (BPM) Standard V8.0.1 technology to address many
of the challenges outlined above. The project demonstrates how a BPM solution can interoperate with multiple third parties providing independent scanning, indexing and coding capabilities. By adopting a modular approach, the solution shows that a single process can interact with a variety of suppliers in a consistent manner, and be used to provide a single point of contact for end-to-end process monitoring.

The system context

Figure 1 shows the system context, defining the three main external providers who contribute services to the end-to-end process.

Figure 1. System context

These third parties provide the following capabilities:

- **Scanning Provider** scans incoming paper invoices and generates digital versions.
- **Indexing Provider** typically uses Optical Character Recognition (OCR) technology to interrogate the digital images and extract the invoice content.
- **Workflow Provider** matches the invoice details against existing Purchase Orders, noting any discrepancies.

A single vendor may provide one or more of these capabilities, however one objective of the project was to ensure that there were no dependencies between providers, and that the process would be able absorb the impact of changing the underlying suppliers.

The business process

The business model consists of two separate views; the analysis model and the implementation model. The analysis model captures the operational intent from a business perspective, and provides a high-level view of the key tasks within the process. The implementation model refines this view to include technology considerations, and provides the executable model, which runs in IBM BPM.

Analysis model

The initial end-to-end analysis model was built through a series of business analysis workshops and contains the process milestones and associated high-level tasks shown in Figure 2.
Tasks in blue represent work that will be implemented by the IBM BPM solution, while the green tasks denote high-level activities that will be provided by the external third parties. Following this analysis, a process was modeled in Blueworks Live, which identified the key points of integration (API 1 to API 8) as shown in Figure 3.

The overall process is long running, lasting for several hours or even days, depending on the response times from the various providers involved in the solution. The activities conducted by these providers contain a mixture of human and automated tasks, however this complexity is encapsulated by the asynchronous interface which they expose to the BPM solution. The process adopts a wait/notification pattern, in which one-way web service calls are used to send messages to third party providers, instructing them to perform some activity. The process then waits for a notification message, indicating that the activity has completed.

This enables the interaction between BPM and the supporting providers to be loosely coupled, allowing the same BPM solution to be deployed to many different customers, each of whom uses different scanning, indexing or workflow providers. As long as the providers can meet the requirements of the interfaces, they will be able to support the process.
The process is initiated by the receipt of a batch of invoices from the Scanning Provider. Batches typically contain around fifty separate invoices and are supplied as a compressed zip file, which is transferred by FTP from the Scanning Provider to the BPM system. The FTP server deposits the batches in a specific directory which is monitored by the BPM solution.

Each API is essentially part of a two-way asynchronous communication. The first two integration points (API 1 and API 2) represent the interaction between the BPM solution and an Indexing Provider. The zipped invoice batch is passed by FTP to the Indexing Provider in API 1, where the contents of each invoice are extracted using OCR technology. The provider then generates a separate XML file for each invoice in the batch, and formats it using the TEAPPS industry standard before transferring it back to BPM in API 2, again using FTP. This means that a single batch containing fifty invoices would generate fifty individual XML response files.

**Figure 4. Indexing Provider interaction**

The BPM process waits for the TEAPPS files to arrive in a monitored directory, before extracting the contents and continuing processing.

The next pair of integrations is used to conduct matching and coding activities with a Workflow Provider. This interface is exposed by the provider as an asynchronous web service (API 5), which accepts a number of parameters that describe the invoice, as well as the actual scanned invoice image, encoded as a Base64 ASCII string. The result is returned to BPM through API 6, a web service interface which correlates the information and updates the relevant instance of the handling process.

Finally the invoice details are passed to the ERP system in API 7, which again exposes an asynchronous web service, with a confirmation message passed back to BPM in API 8 through another web service. Note that APIs 3 and 4 are internal within BPM (focusing on vendor verification) and have therefore been omitted from the diagram.

**Data structures**

Before we examine the Process Implementation model in detail, you should first gain an understanding of the main data structures involved in the solution, including those that support integration with third party providers.
BPM business objects

The main business object from a BPM perspective is the Invoice, which contains the core attributes necessary to describe the invoice within the process. Specifically it contains a list of InvoiceLine objects, each of which represents a distinct line item within the invoice, and contains a description of the item, its cost and tax. The InvoiceBatch object provides a summary representation of the batch file initially received from the Scanning Provider, and reuses the existing Invoice object (and hence implicitly the InvoiceLine object) to describe the contents of the batch.

Figure 5. Key business objects

Given the number of operational systems already involved in this solution, one objective was to not introduce a further System of Record (SoR), therefore the BPM solution does not have its own dedicated invoice database. Instead the process contains enough invoice details to enable it to make all operational decisions, invoke the required integrations and support the gathering of business metrics. Where access to all invoice information is required, the process either passes the underlying invoice image as part of a web service call, or displays the image to the user within a browser as part of a human task.

Batch manifest

The next structure of interest is the manifest file, which is supplied as part of an invoice batch from the Scanning Provider. The manifest is an XML file that provides an overview of the contents of the batch, including a unique id for each invoice. If multiple Scanning Providers are involved in the solution then this id must be unique across all providers, so a provider prefix should typically be appended. Listing 1 shows a section of the manifest file.
The Java Architecture for XML Binding (JAXB) allows Java developers to easily marshall and unmarshall Java classes to and from XML representations.

The document element is repeated for each invoice in the batch and contains an `<inputDocId>` element that is used by the BPM processes to uniquely identify an invoice throughout its lifetime. The BPM process uses Java integration to access a JAXB wrapper, which translates the XML representation into Java data objects, before returning the relevant information to BPM as TWObjects.

The use of JAXB greatly simplifies the mapping process because it allows nested classes to reflect the hierarchical nature of the XML structure. For example, at the root of the manifest file is a `<documentBatch>` element, which is mapped to the Java object in Listing 2.

**Listing 2. InvoiceBatch JAXB class**

```java
@XmlElement (name="documents")
Invoices invoices;
```

Note that the class contains an `XmlElement` tag identifying a `documents` element (highlighted in red), which relates to the `<documents>` tag shown in Listing 1 and which maps to the JAXB Java `Invoices` class shown in Listing 3.
Listing 3. JAXB Invoices class

```java
public class Invoices {
    @XmlElement (name="document")
    List<Invoice> invoices = new ArrayList<Invoice>();

    public List<Invoice> getInvoices() {
        return invoices;
    }
}
```

This class contains a list of `Invoice` objects, which are automatically populated with details from each `<document>` element of the manifest.

Listing 4. JAXB Invoice class

```java
public class Invoice {
    @XmlElement
    String filename;

    @XmlElement
    int numBytes;

    @XmlElement
    String inputDocId;
    ...
}
```

Using JAXB this way makes it a very straightforward task to translate between XML and Java, and it is a technique that is further employed to convert between Java and the TEAPSS XML industry standard defined in the next section.

**TEAPPS Indexed Invoices**

TEAPPS is an invoice messaging standard, created by Teito, an IBM Business Partner with operations throughout Northern Europe, and uses a comprehensive hierarchical XML structure to encapsulate the complexities of invoice handling. Figure 6 shows some of the TEAPPS classes.
Our project did not fully implement the TEAPPS standard, but created a core set of over fifty classes that mapped to the key entities necessary to describe the invoices used within the solution. These classes again used JAXB to translate between Java and XML representations and now exist as a reusable asset in IBM.

Each invoice processed by the Indexing Provider is transferred to the BPM solution as a single XML file, which contains a full description of the invoice, including details of the invoice lines with cost and taxation sections.

To uniquely identify an incoming invoice, the `<CONTENT_REF>` element of the TEAPPS structure was used to contain the correlation id that was originally defined in the `<inputDocId>` element of the batch manifest, as shown in Listing 1.

### Listing 5. TEAPPS XML section

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<INVOICE_CENTER>
  <TRANSPORT_FRAME>
    <TF_CODE>TF01</TF_CODE>
    <TIMESTAMP>20130119142648753</TIMESTAMP>
    <BATCH_ID>13777339</BATCH_ID>
    <CONTENT_RECEIVER>
      <RECEIVER_REF>BWEISE-0012928</RECEIVER_REF>
      <CONTENT_REF>BPMS-Invoice11</CONTENT_REF>
    </CONTENT_RECEIVER>
    <SENDER>BW003705925424</SENDER>
    <FB_REQUEST>1</FB_REQUEST>
  </TRANSPORT_FRAME>
  ...
</INVOICE_CENTER>
```

Once the TEAPPS JAXB classes have been created, the effort required to convert the TEAPPS invoice into Java format is only two lines of code, as shown in Listing 6.
Listing 6. Invoking JAXB mapping

```java
// setup object mapper using the outermost InvoiceCentre class
JAXBContext context = JAXBContext.newInstance(TEAPPS_InvoiceCenter.class);

// parse the XML and return an instance of the full InvoiceCentre class
TEAPPS_InvoiceCenter tic = (TEAPPS_InvoiceCenter)
context.createUnmarshaller().unmarshal(new File (invoiceFilename));
```

Implementation model

Whilst the analysis model was relatively straightforward, the practicalities of building a solution mean that the implementation model in Process Designer is slightly more complex. The following four core processes were developed:

- **Invoice Batch Monitor** is a lightweight "daemon"-style process, that runs continuously, periodically checking a well-known directory for incoming invoice batch files. In the final production solution, it is expected that this will be implemented by a Java application.
- **Indexed Document Monitor** is another monitoring process, examining a separate FTP directory for incoming TEAPPS response files from the Indexing Provider.
- **Invoice Batch Processor** is invoked to handle the receipt of an invoice batch.
- **Handle Invoice** coordinates the processing of a single invoice.

The monitor processes do not merit further investigation for our purposes, however the relationship between the two remaining processes is key to the operation of the solution.

**Invoice Batch Processor**

The Invoice Batch Processor is modeled around the concept of the batch being a single atomic unit of work; that is to say, the process will not complete until all the invoices within the batch have been processed.

**Figure 7. Invoice Batch Processor**

The process is initiated by an incoming message, which is sent by the Invoice Batch Monitor process to indicate that an invoice batch is ready for processing. It retrieves information from the batch through a Java integration, which also transfers the batch by FTP to the Indexing Provider. It then executes the Process Invoice task, which is implemented by the Handle Invoice linked process, and configured as a multi-instance loop as shown in Figure 8.
Figure 8. Multi-instance loop

The **Start Quantity** attribute defines how many instances of the task will execute; in this case, we use the number of invoices in the invoice batch file. The **Ordering** option has been set to enable all instances to run concurrently, while the **Flow Condition** is used to indicate that all instances must complete before the containing process can move on.

To define the data that is passed into each instance of the task, the **Mapping Input** uses a special system variable called `tw.system.step.counter` to iterate over the invoices list contained within the invoiceBatch object. Each time an instance of the task is created by the multi-instance loop, the counter is incremented. By using it as an index into the invoices list, you can therefore pass in a unique invoice to each instance of the task.

So for example, if we received a single batch containing ten invoices, then we would now have ten instances of the Handle Invoice process executing in parallel, each relating to a different invoice within the batch.

**Handle Invoice**

Where the Invoice Batch Processor treats a batch as an atomic unit of work, the Handle Invoice process is focused on handling a single invoice and performs many of the core processing activities identified in the high-level Blueworks Live model, as shown in Figure 9.

Figure 9. Handle Invoice
After invocation the process immediately enters a wait state. It stays in this state until the Indexed Document Monitor sends it a notification (using an undercover agent or UCA) indicating that an indexed TEAPP$ response file has arrived and is ready for processing. However, we are now operating on an invoice level, with multiple instances of this process in a wait state, therefore the UCA from the Indexed Document Monitor must correlate using a unique identifier to synchronize with the relevant instance of the process.

**Process Correlation**

Going back to Listing 5, you see that the TEAPP$ structure uses the `<CONTENT_REF>` element to uniquely identify an invoice. Once the JAXB integration has marshalled the TEAPP$ XML into Java objects, this value is passed back to the Indexed Document Monitor as a `correlationId` parameter as part of API 2, along with other elements from the TEAPP$ indexed document.

The monitor process then uses the UCA to send a notification to the relevant instance of the Handle Invoice process (Figure 9), where it will be received by the Wait For Indexed Notification intermediate message event. This event has been configured to match the incoming `correlationId` against the existing `inputDocId` enabling it to receive the appropriate invoice details.

**Figure 10. Wait For Indexed Notification process correlation**

<table>
<thead>
<tr>
<th>Correlation and Output Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select one variable to be used for correlation. Specify data mappings from the UCA output doc</td>
</tr>
<tr>
<td>ilationId (String)</td>
</tr>
<tr>
<td>indexedDocument (Index)</td>
</tr>
</tbody>
</table>

Variations of this technique have been applied throughout the processes, to correlate an incoming asynchronous web service response with the relevant process instance. You'll find more information about this in Part 2, which examines the Java and web service integrations more closely.

**Conclusion**

The first part of this series introduced the context of the business problem that the project addressed and describes the process and associated data structures that underpin the solution. **Part 2** will continue the analysis, investigating the Java and web service integrations, as well as the image manipulation techniques involved in the solution.