5
Using BPEL to Build Composite Services and Business Processes

In the previous two chapters, we've looked at how we can service enable functionality embedded within existing systems. The next challenge is how to assemble these services to build "composite" applications or business processes. This is the role of the Web Service **Business Process Execution Language** or **BPEL** as it's commonly referred to.

BPEL is a rich XML based language for describing the assembly of a set of existing web services into either a composite service or a business process. Once deployed, a BPEL process itself is actually invoked as a web service.

Thus anything that can call a web service, can also call a BPEL process, including of course other BPEL processes. This allows you to take a nested approach to writing BPEL processes, giving you a lot of flexibility.

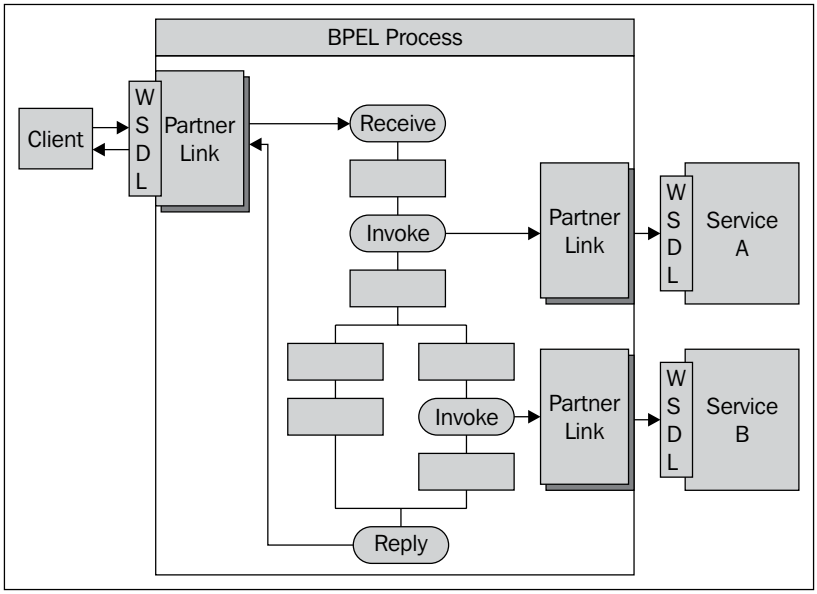
In this chapter we first introduce the basic structure of a BPEL process, its key constructs, and the difference between a synchronous and asynchronous service.

We then demonstrate through the building and refinement of two example BPEL processes, one synchronous the other asynchronous, how to use BPEL to invoke external web services (including other BPEL processes) to build composite services. During this procedure we also take the opportunity to introduce the reader to many of the key BPEL activities in more detail.

Basic structure of a BPEL process

The following diagram shows the core structure of a BPEL process, and how it interacts with components external to it; either web services that the BPEL process invokes (Service A and Service B in this case) or external clients that invoke the BPEL process as a web service.

From this we can see that the BPEL process divides into two distinct parts; the **Partner Links**, with associated **WSDL** files which describe the interactions between the BPEL process and the outside world; and the **core BPEL process** itself, which describes the process to be executed at run time.



Core BPEL process

The core BPEL process consists of a number of steps, or activities as they are called in BPEL.These consist of simple activities, including:

- `Assign`: It is used to manipulate variables.
- `Transform`: It is a specialised assign activity that uses XSLT to map data from a source format to a target format.
- `Wait`: It is used to pause the process for a period of time.
- `Empty`: It does nothing. It is used in branches of your process where syntactically an activity is required, but you don't want to perform an activity.

The structured activities which control the flow through the process. These include:

- `While`: It is used for implementing loops.
- `Switch`: It is a construct for implementing conditional branches.
- `Flow`: It is used for implementing branches which execute in parallel.
- `FlowN`: It is used for implementing a dynamic number of parallel branches.

There are messaging activities as well (for example `Receive`, `Invoke`, `Reply`, and `Pick`).

The activities within a BPEL process can be sub-divided into logical groups of activities, using the `Scope` activity. As well as providing a useful way to structure and organize your process, it also enables you to define attributes such as variables, fault handlers, and compensation handlers that just apply to the scope.

Variables

In addition each BPEL process also defines variables, which are used to hold the state of the process as well as messages that are sent and received by the process. They can be defined at the process level, in which case they are considered global and visible to all parts of the process. Or it can be declared within a `Scope` in which case they are only visible to activities contained within that `Scope` (and scopes nested within the scope to which the variable belongs).

Variables can be one of the following types:

- **Simple Type**: It can hold any simple data type defined by XML Schema (for example string, integer, boolean, and float).
- **WSDL Message Type**: It is used to hold the content of a WSDL Message sent to or received from partners.
- **Element**: It can hold either a complex or simple XML Schema element defined in either a WSDL file or a separate XML Schema.

Variables are manipulated using the `<assign>` activity, which can be used to copy data from one variable to another, as well as create new data using XPath Expressions or XSLT.

For variables which are WSDL Messages or Complex Elements we can work with it at the sub-component level by specifying the part of the variable we would like to work with using an XPath expression.

Partner Links

All interaction between a process and other parties (or partners) is via web services as defined by their corresponding WSDL files. Even though each service is fully described by its WSDL, it fails to define the relationship between the process and the partner, that is who the consumer of a service is and who the provider is. On first appearance, the relationship may seem implicit; however, this is not always the case so BPEL uses Partner Links to explicitly define this relationship.

Partner Links are defined using the `<partnerLinkType>` which is an extension to WSDL (defined by the BPEL standard). Whenever you reference a web service whose WSDL doesn't contain a `<partnerLinkType>`, JDeveloper will automatically ask you whether you want it to create one for you. Assuming you answer yes it will create this as a separate WSDL document, which then imports the original WSDL.

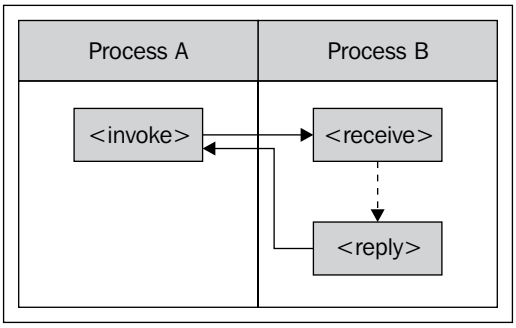
Messaging activities

BPEL defines three messaging activities `<receive>`, `<reply>`, and `<invoke>`. How you use these depends on whether the message interaction is either synchronous or asynchronous and whether the BPEL process is either a consumer or provider of the service.

Synchronous messaging

With synchronous messaging, the caller will block until it has received a reply (or times out); that is the BPEL process will wait for a reply before moving onto the next activity.

As we can see in the diagram below, Process A uses the `<invoke>` activity to call a synchronous web service (Process B in this case); once it has sent the initial request, it blocks and waits for a corresponding reply from Process B.



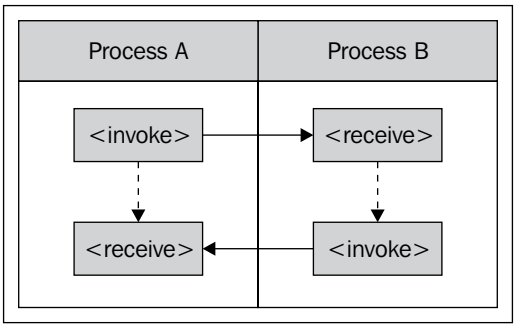
Process B, uses the <receive> activity to receive the request; once it has processed the request it uses the <reply> activity to send a response back to Process A.

Theoretically Process B could take as long as it wants before sending a reply, but typically Process A will only wait a short time (for example 30 seconds) before it times out the <invoke> operation under the assumption that something has gone wrong. Thus if Process B is going to take a substantial period of time before replying, then you should model the exchange as an Asynchronous Send-Receive (see next section).

Asynchronous messaging

With asynchronous messaging, the key difference is that once the caller has sent the request, the send operation will return immediately, and the BPEL process may then continue with additional activities until it is ready to receive the reply, at which point the process will block until it receives the reply (which may already be there).

If we look at the following diagram, you will notice that just like the synchronous request Process A uses the <invoke> activity to call an asynchronous web service. However, the difference is that it doesn't block waiting for a response, rather it continues processing until it is ready to process the response. It then receives this using the <receive> activity.



Conversely, Process B uses a <receive> activity to receive the initial request and an <invoke> activity to send back the corresponding response.

While at a logical level there is little difference between synchronous and asynchronous messaging (especially if there are no activities between the <invoke> and <receive> activity in Process A), at a technical level there is a key difference.

This is because with asynchronous messaging, we have two `<invoke>`, `<receive>` pairs, each corresponding to a separate web service operation; one for the request, the other for the reply.

From a decision perspective a key driver as to which to choose is the length of time it takes for Process B to service the request, since asynchronous messaging supports far longer processing times. In general once the time it takes for Process B to return a response goes above 30 seconds you should consider switching to asynchronous messaging.

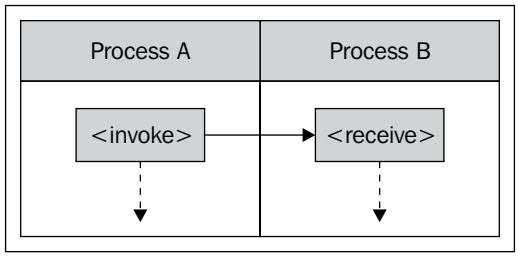
With potentially many instances of Process A and Process B running at the same time, BPEL needs to ensure that each reply is matched (or correlated) to the appropriate request. By default BPEL uses WS-Addressing to achieve this. We look at this in more detail in Chapter 14—*Message Interaction Patterns*.

One way messaging

A variation of asynchronous messaging is one way messaging (also known as **fire and forget**). This involves a single message being sent from the calling process, with no response being returned.

If we look at the following diagram, you will notice that just like the asynchronous request Process A uses the `<invoke>` activity to send a message to Process B.

Once Process A has sent the message, it continues processing until it completes, that is it never stops to wait for a response from Process B. Likewise Process B, upon receipt of the message, continues processing until it has completed and never sends any response back to Process A.



A simple composite service

Despite the fact that BPEL is intended primarily for writing long running processes, it also provides an excellent way to build a composite service, that is a service that is assembled from other services.

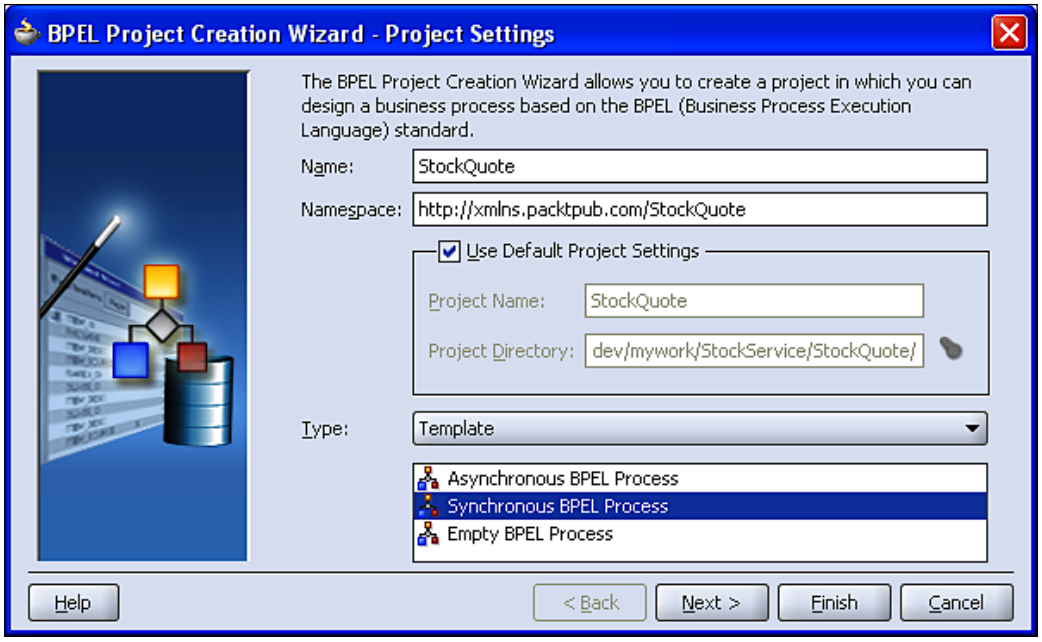
Let's takes a simple example: say I have a service that gives me the stock quote for a specified company, and that I also have a service that gives me the exchange rate between two currencies. I can use BPEL to combine these two services and provide a service that gives the stock quote for a company in the currency of my choice.

So let's create our stock quote service; we will create a simple synchronous BPEL process which takes two parameters, the stock ticker and the required currency. This will then call two external services.

Creating our Stock Quote service

Before we begin, we will create a **StockService** application, which we will use for all our samples in this chapter. To do this follow the same process we used to create our first application in Chapter 2.

Next add a BPEL project to our **StockService** application. Specify a name of **StockQuote** and select a synchronous BPEL process. However, at this stage **DO NOT** click **Finish**.

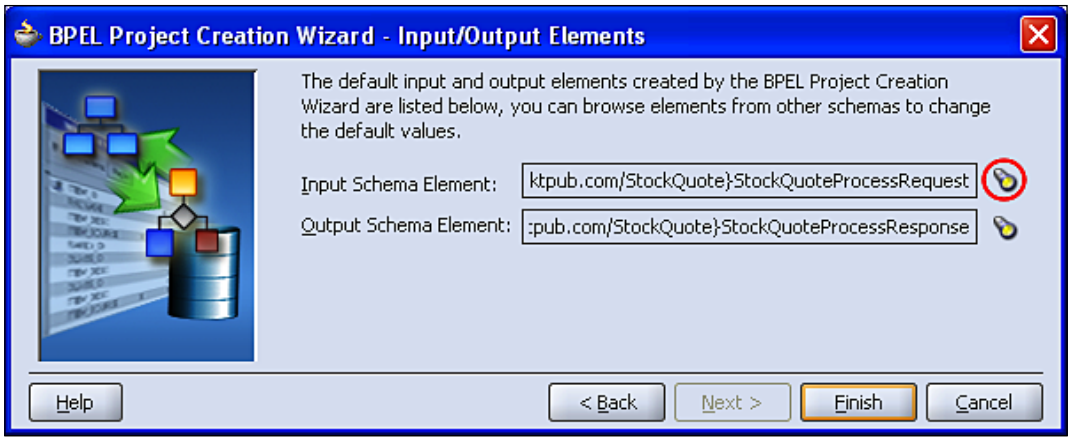


You may remember when we created our `Echo` service back in Chapter 2; JDeveloper automatically created a simple WSDL file for our service, with a single input and output field for our service. For our stock quote service we need to pass in multiple fields (that is Stock Ticker and Currency). So to define the input and output messages for our BPEL process we are going to make use of a predefined schema `StockService.xsd`, shown below (for brevity only the parts which are relevant to this example are shown; however, the complete schema is provided in the downloadable samples file for this book).

```
<?xml version="1.0" encoding="windows-1252"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns="http://xmlns.packtpub.com/StockService"
            targetNamespace="http://xmlns.packtpub.com/StockService"
            elementFormDefault="qualified" >

  <xsd:element name="getQuote"         type=" tGetQuote"/>
  <xsd:element name="getQuoteResponse" type=" tGetQuoteResponse"/>

  <xsd:complexType name="tGetQuote">
    <xsd:sequence>
      <xsd:element name="currency" type="xsd:string"/>
      <xsd:element name="stockSymbol" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="tGetQuoteResponse">
    <xsd:sequence>
      <xsd:element name="stockSymbol" type="xsd:string"/>
      <xsd:element name="currency" type="xsd:string"/>
      <xsd:element name="amount" type="xsd:decimal"/>
    </xsd:sequence>
  </xsd:complexType>

   …

</xsd:schema>
```

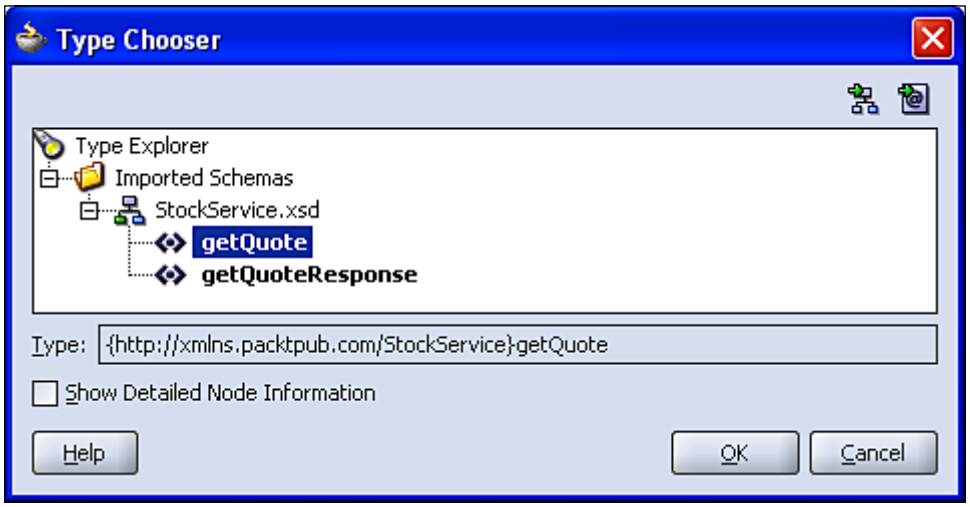
Import StockService schema

Within the **Create BPEL Process** dialogue, click on **Next**, this will show you the input and output elements for your process, as shown in the following screenshot:



To override the default input and output schema elements generated by JDeveloper, click on the **flashlight** (circled above). This will bring up a dialogue that allows you to browse all schemas imported by the BPEL process and select an element from them. In our case, as we have yet to import any schemas it will automatically launch the **Select Schema** window in file mode which will allow us to search our file system for an appropriate schema.

Find the `StockService.xsd` located in the samples folder for Chapter 5 and select this. It will now open the schema browser dialogue. Browse this and select the **getQuote** element as illustrated in the following screenshot:



Repeat this step for the **Output Schema Element**, but this time, select the **getQuoteResponse** element. Click **Finish** and this will create our **StockQuote** process.

You will see that by default JDeveloper has created a skeleton BPEL process, which contains an initial `<receive>` activity to receive the stock quote request, followed by a `<reply>` activity to send back the result (as we discussed in the earlier section—*Synchronous messaging*). In addition it will have created two variables: `inputVariable` which contains the initial stock quote request and `outputVariable` in which we will place the result to return to the requestor.

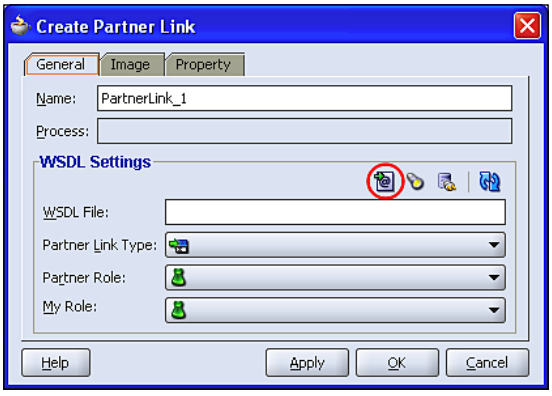
If you look in the **Projects** section of the **Application** navigator you will see it contains the file `StockQuote.wsdl`. This contains the WSDL description (including Partner Link extensions) for our process. If you examine this, you will see we have a single operation: `process`, which is used to call the BPEL process.

Calling the external web services

The next step is to call our external web services; for our stock quote service we are going to use Xignite's Quotes web service which delivers delayed equity price quotes from all U.S. stock exchanges (NYSE, NASDAQ, AMEX, NASDAQ OTC bulletin board, and Pink Sheets).

Before you can use this service you will need to register with Xignite; to do this or for more information on this and other services provided by Xignite go to `www.xignite.com`.

To call a web service in BPEL we first need to create a Partner Link (as discussed at the start of this chapter). So from the component pallet, select the **Services** drop down and drag a **PartnerLink** into the **Services** swim lane in your BPEL process. This will pop up the following screen:



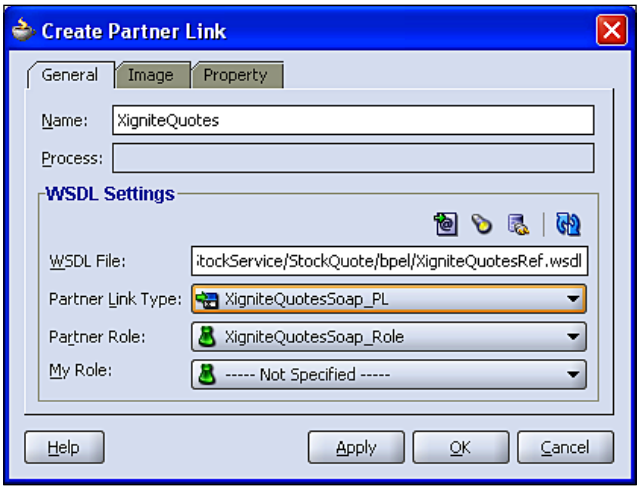
First enter a name for the Partner Link, for example `XigniteQuotes`. Next we need to specify the WSDL file for the Partner Link. JDeveloper provides the following ways to do this:

- **Browse WSDL File**: This allows us to browse the file system for WSDL files that define the service.
- **Service Explorer**: This allows us to browse services that are already defined within the composite application (for example other BPEL processes or ESB services).
- **Define Adapter Service**: This enables us to define adapter services (as covered in Chapter 3) directly within the context of a BPEL process.
- **Enter URL**: Directly enter the URL for the WSDL File into the corresponding field.

For our purposes, we have a local copy of the WSDL for Xignite's Quotes service, called `XigniteQuotes.wsdl`, which is included with the samples for Chapter 5. Click on the **Browse WSDL File ...** icon (highlighted in previous screenshot) then browse to and select this file (select **Yes** if prompted to create a local copy of the file).

JDeveloper will parse the WSDL and, assuming it is successful, will pop up a window saying there are no Partner Link types defined in the current WSDL, and will ask if you want to create partner links for the file: click **Yes**. JDeveloper will then create one **Partner Link Type** for each Port Type defined in the WSDL.

In cases where we have multiple Partner Link types, we will need to specify which one to use within our process. To do this, click on the drop down next to **Partner Link Type** and select the appropriate one. In our case we have selected **XigniteQuotesSoap_PL**, as shown in the following screenshot:



Finally we need to specify the **Partner Role** and **My Role**. When invoking a synchronous service, there will only be a **single** role defined in the WSDL, which represents the provider of the service. So specify this for the **Partner Role** and leave **My Role** as **----- Not Specified -----**.

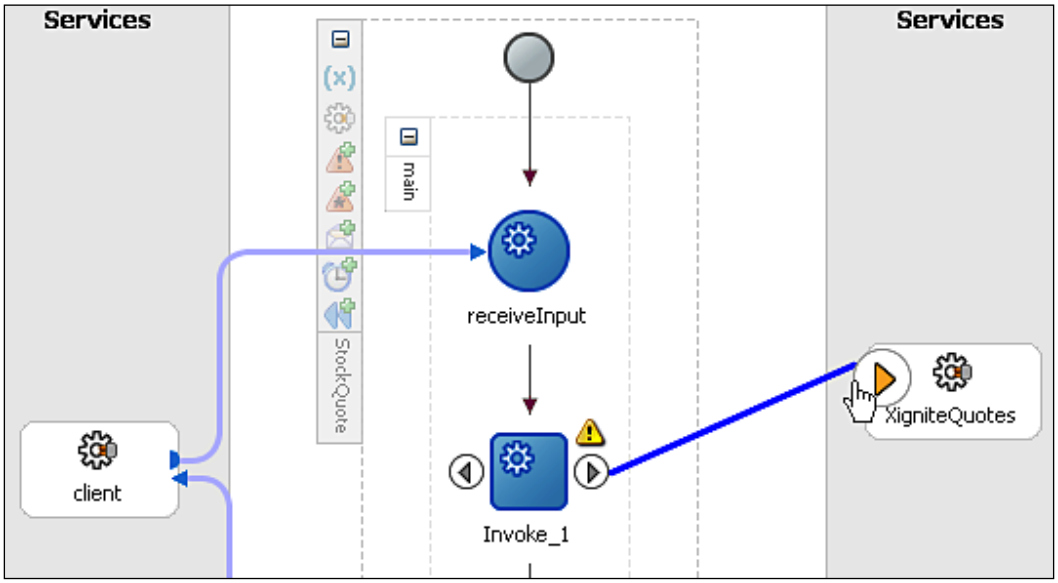
Best practice would dictate that rather than call the Stock Quote service directly from within BPEL, we would invoke it via the service bus.

Calling the web service

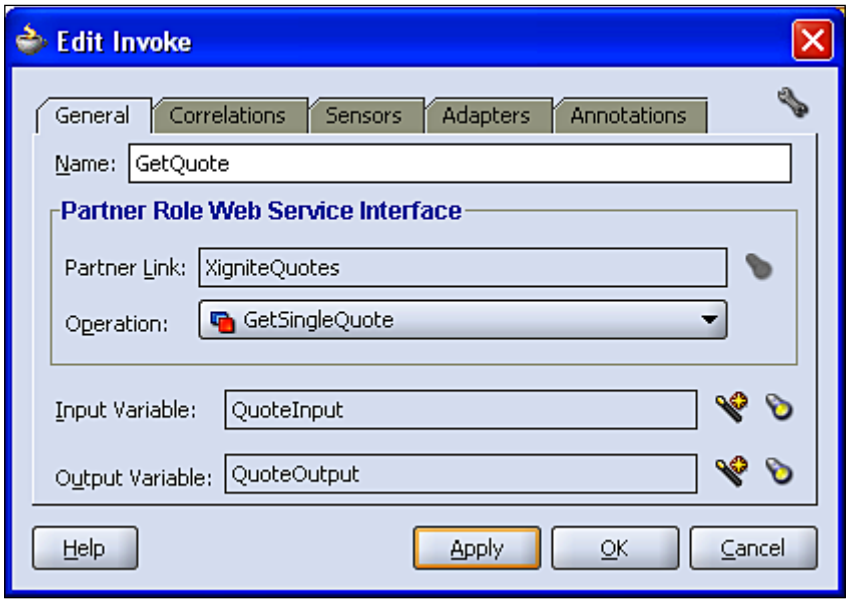
Once we have defined a Partner Link for the web service, the next step is to call it. As this is a synchronous service we will need to use an `<invoke>` activity to call it, as we described earlier in this chapter.

On the component palette, ensure the submenu of **Process Activities** is selected, and then from it drag an **Invoke** activity on to your BPEL process.

Next, place your mouse over the arrow next to the **Invoke** activity; click and hold your mouse button and then drag the arrow over your Partner Link, then release. This is shown in the following screenshot:



This will then pop up the **Edit Invoke** activity window as shown in the following screenshot:



We need to specify a number of values to configure the invoke activity, namely:

- **Name**: This is the name we want to assign to the invoke activity, and can be any value. So just assign a meaningful value such as `GetQuote`.
- **Partner Link**: This is the Partner Link whose service we want to invoke; it should already be set to use `XigniteQuotes`, as we have already linked this activity to that Partner Link. An alternative approach would be to click on the corresponding spotlight icon which would allow us to select from any Partner Link already defined to the process.
- **Operation**: Once we've specified a Partner Link, we need to specify which of its operations we wish to invoke. This presents us with a drop down, listing all the operations that are available; for our purpose select `GetSingleQuote`.
- **Input Variable**: Here we must specify the variable which contains the data to be passed to the web service that's being invoked. It is important that the variable is of type `Message`, and that it is of the same message type expected by the **Operation** (that is as defined in the WSDL file for the web service).

The simplest way to ensure this is to get JDeveloper to create the variable for you; to do this, click on the **magic wand** to the right of the input variable field. This will bring up the **Create Variable** window as shown below. You will notice that JDeveloper creates a default name for the variable (based on the name you gave the invoke operation and the operation that you are calling), you can override this with something more meaningful (for example `QuoteInput`).

- **Output Variable**: Finally, we must specify the variable into which the value returned by the web service will be placed. As with the input variable, this should be of type `Message`, and corresponds to the output message defined in the WSDL file for the selected operation. Again the simplest way to ensure this is to get JDeveloper to create the variable for you.

Once you've specified values for all these fields, as illustrated above, click **OK**.

Assigning values to variables

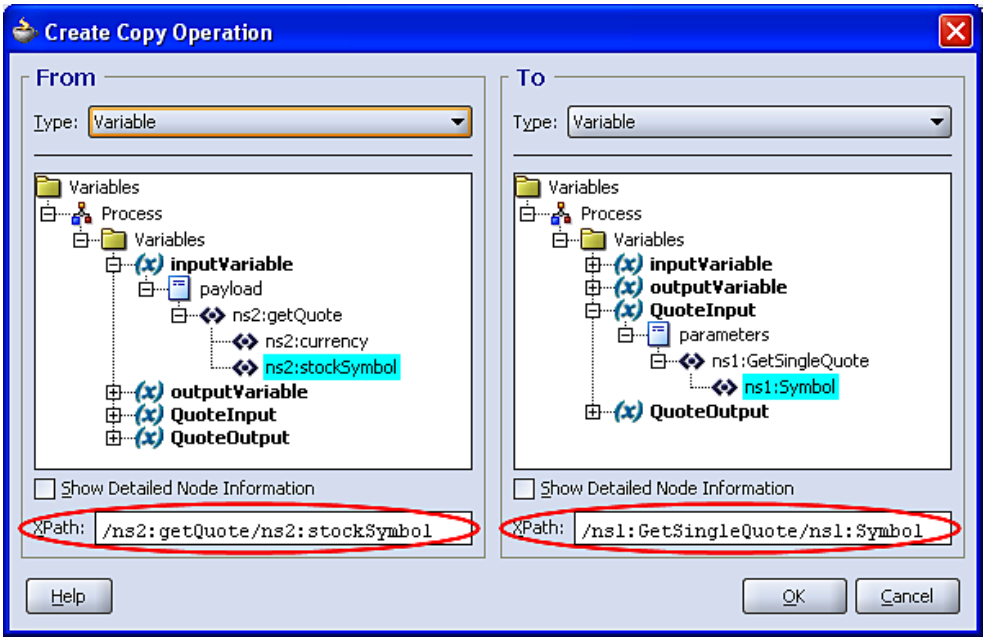
In our previous step, we created the variable `QuoteInput`, which we pass to our invocation of `GetSingleQuote`. However, we have yet to initialize the variable or assign any value to it.

To do this BPEL provides the `<assign>` activity, which is used to update the values of variables with new data. The assign activity typically consists of one or more copy operations. Each copy consists of a target variable, that is the variable that you wish to assign a value to and a source; this can either be another variable or an XPath expression.

For our purpose, we want to assign the stock symbol passed into our BPEL process to our `QuoteInput` variable.

To do this drag an assign activity from the component pallet on to your BPEL process at the point just before our invoke activity. Then double click on it to open up the **Assign** configuration window. Click on the **Create** menu and select **Copy Operation...**

This will present us with the **Create Copy Operation** window shown in the following screenshot:



On the left-hand side we specify the **From** variable (that is the source). Here we want to specify the stock symbol passed in as part of the input variable to the BPEL process, so expand the **inputVariable** tree and select **/ns2:getQuote/ns2:stockSymbol.**

For the target expand **QuoteInput** and select **/ns1:GetSingleQuote/ns1:Symbol**.

You will notice that for both the source and target, JDeveloper has created the equivalent XPath expression (circled in the previous screenshot).

The source and target can either be a simple type (for example `xsd:int`, `xsd:date`, or `xsd:string`), as in the above example. Or a complex type (for example `ns2:getQuote`), but make sure the source and target are either of the same type, or at least compatible.

Testing the process

At this stage, even though the process isn't complete, we can still save, deploy, and run our process. Do this in the same way as previously covered in Chapter 2. When you run the process from the BPEL console you will notice that it doesn't return anything (as we haven't specified this yet). But if you look at the audit trail you should successfully see the `GetSingleQuote` operation being invoked. Assuming this is the case, we know we have implemented that part of the process correctly.

Calling the exchange rate web service

The next step of the process is to determine the exchange rate between the requested currency and the US dollar (the currency used by the `GetSingleQuote` operation). For this we are going to use the currency convertor service provided by `webserviceX.NET`.

For more information on this and other services provided by webserviceX.NET go to `www.webservicex.net`.

This service provides a single operation `ConversionRate`, which gets the conversion rate from one currency to another. The WSDL file for this service can be found at the following URL:

```
http://www.webservicex.net/CurrencyConvertor.asmx?wsdl
```

For convenience we have included a local copy of the WSDL for `webserviceX.NET`'s Currency Convertor service, called `CurrencyConvertor.wsdl`, which is included with the samples for Chapter 5.

To invoke the `ConversionRate` operation, we will follow the same basic steps that we did in the previous section to invoke the `GetQuickQuote` operation. For the sake of brevity we won't repeat them here, but will allow the reader to do this.

For the purpose of following the examples below, name the input variable for the exchange rate web service `ExchangeRateInput` and the output variable `ExchangeRateOutput`.

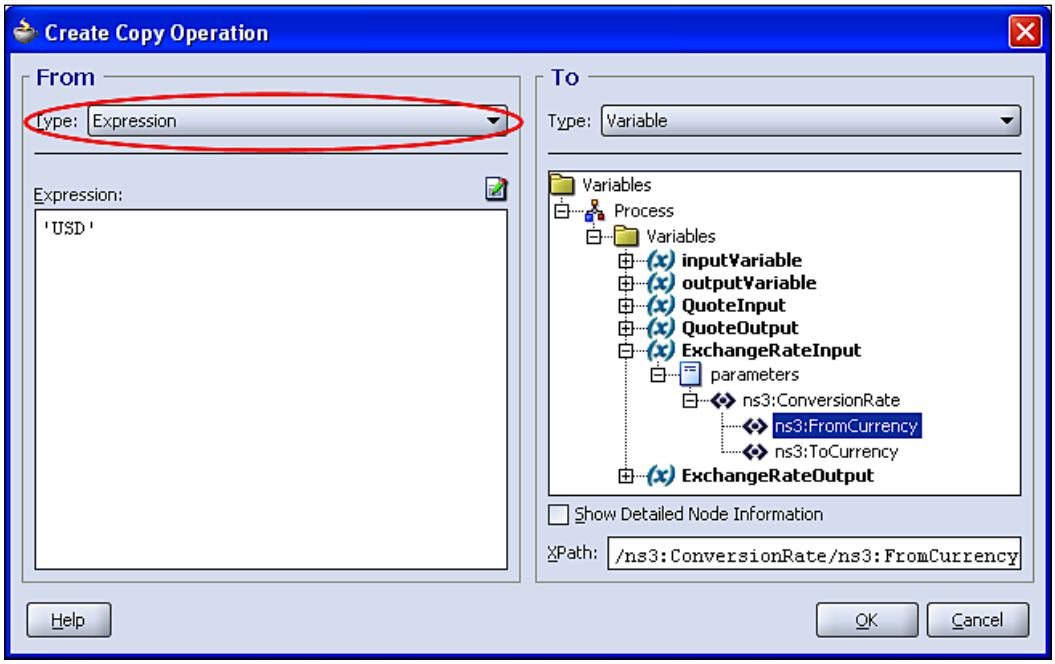
Assigning constant values to variables

The operation `ConversionRate` takes two input values:

- `FromCurrency`, which should be set to `USD`
- `ToCurrency`, which should be set to the currency field contained within the `inputVariable` for the BPEL process

To set the `FromCurrency`, create another copy operation. However for the **From** value select **Expression** as the **Type** (circled in the following screenshot).

This will replace the Variable browser with a free format text box. In here you can specify any value, within quotes, that you wish to assign to your target variable. For our purpose we have entered **'USD'**, as shown in the following screenshot:



To set the value of **ToCurrency**, create another copy operation and copy in the value of the `currency` field contained within the **inputVariable**.

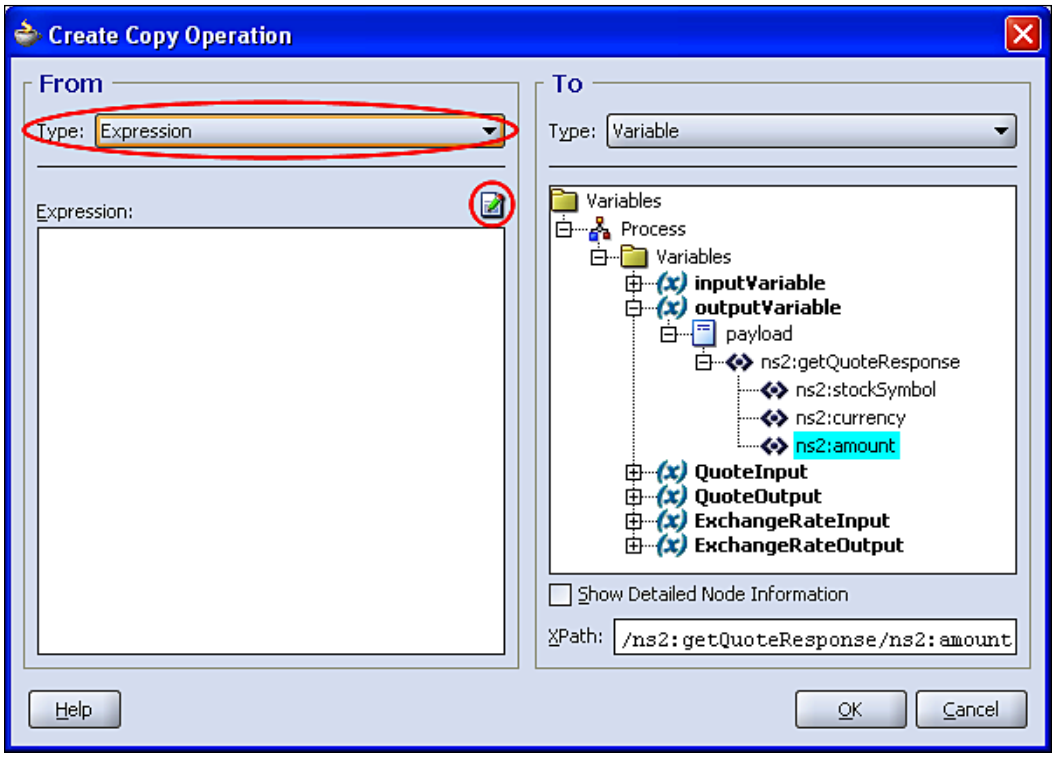
Again at this stage, save, deploy, and run the process to validate that we are calling the exchange rate service correctly.

Using the Expression builder

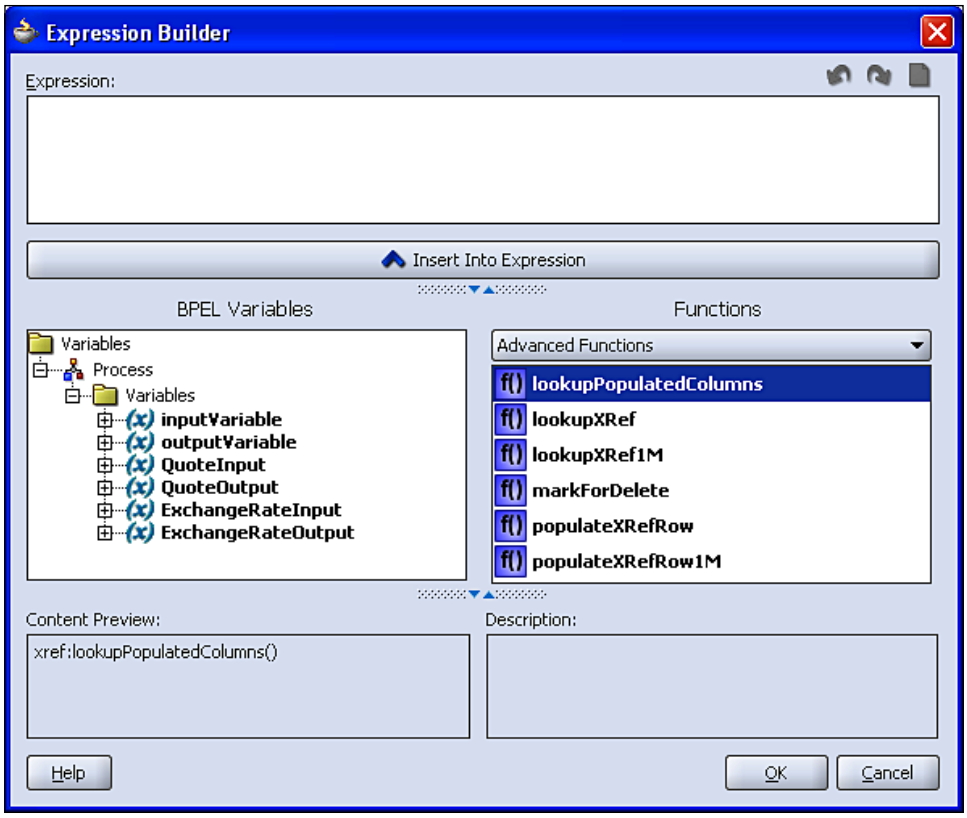
The final part of the process is now to combine the exchange rate returned by one service with the stock price returned by the other in order to determine the stock price in the requested currency and return that to the caller of the BPEL process.

To do this, we will again use an assign activity; so drag another assign activity onto the process, just after our second invoke activity. Now in our previous use of the assign activity, we have just used it to copy a value from one variable to another.

Here, it is slightly different, in that we want to combine multiple values into a single value, and to do that we will need to write the appropriate piece of XPath. Create a copy operation as before, but for the source type, select **Expression** from the drop down as shown in the following screenshot:



Now if you want, you can type in the XPath expression manually (into the **Expression** area), but it's far easier and less error prone to use the expression builder. To do this click on the **XPath expression builder** icon, circled in the previous figure. This will pop up the expression builder (shown in the following screenshot).



The expression builder provides a graphical tool for writing XPath expressions which are executed as part of the copy operation. It consists of the following areas:

- **Expression**: The top text box contains the XPath expression, which you are working on. You can either type data directly in here or use the Expression Builder to insert XPath fragments in here to build up the XPath required.
- **BPEL Variables**: This part of the expression builder lets you browse the variables defined within your BPEL process. Once you've located the variable that you wish to use click the **Insert Into Expression** button and this will insert the appropriate code fragment into the XPath Expression.

The code fragment is inserted at the point within the expression that the cursor is currently positioned.

- **Functions**: This shows you all the different types of XPath functions that are available to build up your XPath expression. To make it easier to locate the required function, they are grouped into categories such as String Functions, Mathematical Functions.

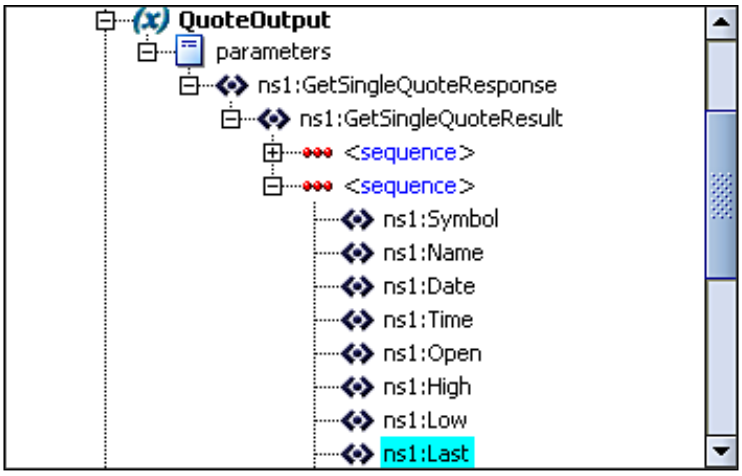
  The drop-down list lets you select the category that you are interested in (for example **Advanced Functions** as illustrated in the previous figure), and then the window below that lists all the functions available with that group.

  To use a particular function, select the required function and click **Insert into Expression**. This will insert the appropriate XPath fragment into the XPath Expression (again at the point that the cursor is currently positioned).
- **Content Preview**: This box displays a preview of the content that would be inserted into the XPath Expression if you clicked the **Insert into Expression** button. For example, if you had currently selected a particular BPEL variable, it would show you the XPath to access that variable.
- **Description**: If you've currently selected a function, this box provides a brief description of the function, as well as the expected usage and number of parameters.

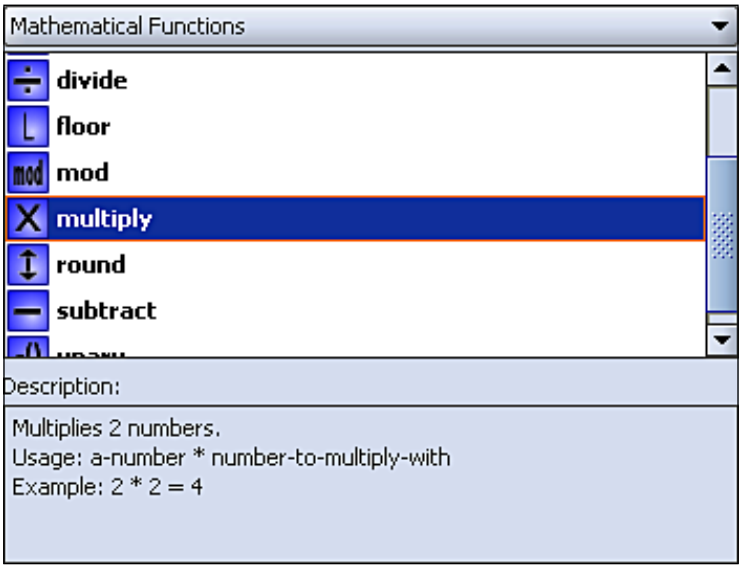
So let's use this to build our XPath expression. The expression we want to build is a relatively simple one, namely, the stock price returned by the stock quote service multiplied by the exchange rate returned by the exchange rate service.

To build our XPath expression, carry out the following steps:

1. First, within the **BPEL Variables** area, in the variable **QuoteOutput** locate the element **ns1:GetSingleQuoteResult/ns1:Last** as shown:



2. Then click **Insert into Expression** to insert this into the XPath Expression.
3. Next, within the **Functions** area, select the **Mathematical Functions** category and select the **multiply** function (notice the description in the **Description** box as shown in the following screenshot) and insert this into the XPath Expression.



4. Finally, back in the the **BPEL Variables** area, locate the element `ConversionRateResult` within the variable `ExchangeRateOutput` and insert that into the XPath Expression.

You should now have an XPath Expression similar to the one illustrated below; once you are happy with this click **OK**.

```
bpws:getVariableData('QuoteOutput','parameters','/nsl:GetSingleQuoteResponse
/nsl:GetSingleQuoteResult/nsl:Last') * bpws:getVariableData('ExchangeRateOutput'
,'parameters','/ns3:ConversionRateResponse/ns3:ConversionRateResult')
```

Finally make sure you specify the target part of the `Copy` operation, which should be the `amount` element within the `outputVariable`.

In order to complete the assign activity, you will need to create two more copy operations to copy the `Currency` and `StockSymbol` specified in the `inputVariable` into the equivalent values in the `outputVariable`.

Once done, your BPEL process should be complete, so deploy it to the BPEL engine and run the process.

Asynchronous service

Following our stock quote service; another service would be a stock order service, which would enable us to buy or sell a particular stock. For this service a client would need to specify the stock, whether they wanted to buy or sell, the quantity and the price.

It makes sense to make this an asynchronous service, since once the order had been placed it may take seconds, minutes, hours, or even days for the order to be matched.

Now, I'm not aware of any trade services that are free to try (probably for good reason!). However, there is no reason why we can't simulate one. To do this we will write a simple asynchronous process.

To do this add another BPEL project to our `StockService` application and give it the name of `StockOrder`, but specify that it is an asynchronous BPEL process.

As with the `StockQuote` process we also want to specify predefined elements for its input and output. The elements we are going to use are `placeOrder` for the input and `placeOrderResponse` for the output, the definitions for which are shown:

```
<xsd:element name="placeOrder"         type="tPlaceOrder"/>
<xsd:element name="placeOrderResponse" type="tPlaceOrderResponse"/>
<xsd:complexType name="tPlaceOrder">
  <xsd:sequence>
    <xsd:element name="currency"    type="xsd:string"/>
    <xsd:element name="stockSymbol" type="xsd:string"/>
    <xsd:element name="buySell"     type="xsd:string"/>
    <xsd:element name="quantity"    type="xsd:integer"/>
    <xsd:element name="bidPrice"    type="xsd:decimal"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="tPlaceOrderResponse">
  <xsd:sequence>
    <xsd:element name="currency"    type="xsd:string"/>
    <xsd:element name="stockSymbol" type="xsd:string"/>
    <xsd:element name="buySell"     type="xsd:string"/>
    <xsd:element name="quantity"    type="xsd:integer"/>
    <xsd:element name="actualPrice" type="xsd:decimal"/>
  </xsd:sequence>
</xsd:complexType>
```

These are also defined in the `StockService.xsd` that we imported for the `StockQuote` process; we will also need to import it into our `StockOrder` process, so that we can use it here (in Chapter 10 we will look at how we can share a schema across multiple processes).

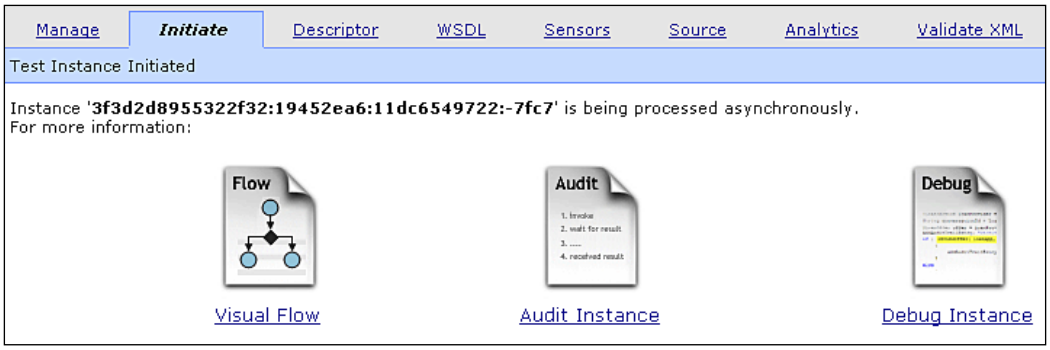
As we did when creating our `StockQuote` process, click on **Next** within the **Create BPEL Process** dialogue to display the input and output elements from the process. Then in turn for each field click on the flashlight to import the schema, bring up the type chooser and select the appropriate element definitions. Then click **Finish** to create the process.

You will see that by default JDeveloper has created a skeleton asynchronous BPEL process, which contains an initial `<receive>` activity to receive the stock order request, but this time followed by an `<invoke>` activity to send the result back (as opposed to a `<reply>` activity used by the synchronous process).

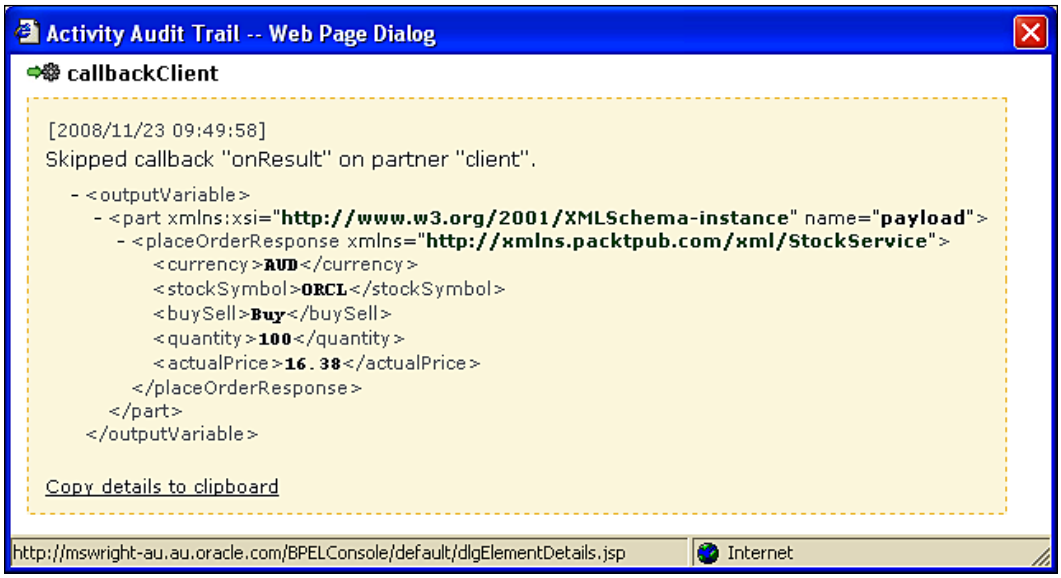
If you look at the WSDL for the process, you will see that it defines two operations; `initiate` to call the process, and `onResult` which will be called by the process to send back the result. Thus the client that calls the `initiate` operation will need to provide the `onResult` callback in order to receive the result (this is something we will look at in more detail in Chapter 14—*Message Interaction Patterns*).

Now for the purpose of our simulation we will assume that the `StockOrder` request is successful and the `actualPrice` achieved is always the bid price. So to do this, create an assign operation, that copies all the original input values to their corresponding output values. Deploy the process and run it from the console.

This time you will notice that no result is returned to the console, rather it displays a message to indicate that the process is being processed asynchronously, as shown in the following screenshot:



Click on the **Visual Flow** link to bring up the audit trail for the process and then click on the **callbackClient** activity at the end of the audit trail. This will pop up a window showing the details of the response sent by our process, as shown in the following screenshot:



Using the Wait activity

Now you've probably spotted the most obvious flaw with this simulation, in that the process returns a response almost immediately, which negates the whole point of making it asynchronous.

To make it more realistic we will use the `<wait>` activity to wait for a period of time. To do this drag the **Wait** activity from the component pallet onto your BPEL process just before the assign activity, and then double click on it to open the **Wait** activity window as shown on the next page.

The **Wait** activity allows you to specify that the process waits **For** a specified duration of time or **Until** a specified deadline. In either case you specify a fixed value or choose to specify an XPath Expression to evaluate the value at run time.

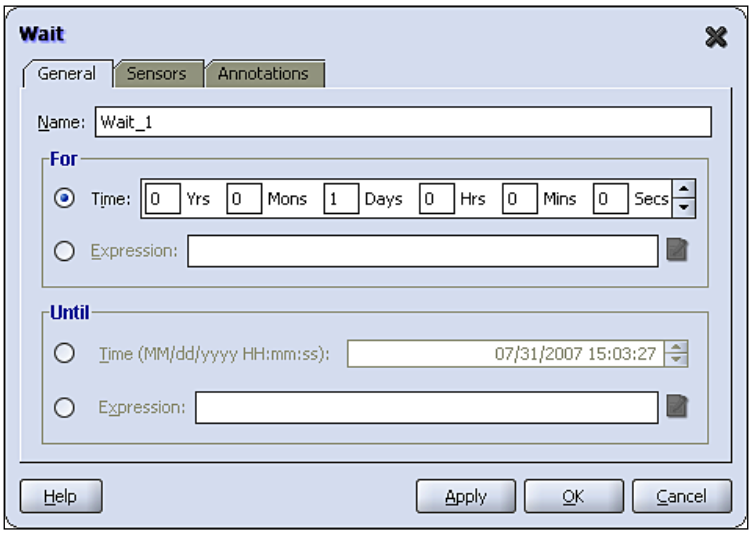
If you specify **Expression**, then if you click the icon to the right of it, this will launch the Expression builder that we introduced earlier in the chapter. The result of the expression must evaluate to a valid value of `xsd:duration` for periods and `xsd:dateTime` for deadlines.

The format of `xsd:duration` is `PnYnMnDTnHnMnS`; for example `P1M` would be a duration of 1 month and `P10DT1H25M` would be 10 days, 1 hour, and 25 minutes.

For deadlines the expression should evaluate to a valid value of `xsd:date`.

The structure of `xsd:dateTime` is `YYYY-MM-DDThh:mm:ss+hh:mm`, where the `+hh:mm` is optional and is the time period offset from UTC (or GMT if you prefer), obviously the offset can be negative or positive.

For example `2008-08-19T17:37:47-05:00` is the time 17:37:47 on August 19th 2008, 5 hours behind UTC (that is Eastern Standard Time in the US).



For our purposes we just need to wait for a relatively short period of time, so set it to wait for one minute.

By default, Wait will wait for a period of **one day**. So after you've changed the **Mins** field to `1`, ensure that **Days** field is set to `0`. I've often had people complaining that the BPEL process has been waiting more than a minute and something has gone wrong!

Now save, deploy and run the process. If you now look at the audit trail of the process you will see it has paused on the Wait activity (which will be highlighted in orange).

Improving the stock trade service

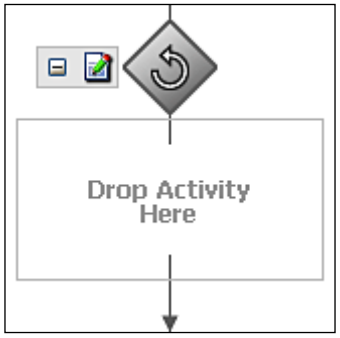
We have a very trivial trade service, which always results in a successful trade after 1 minute. Let's see if we can make it a bit more "realistic".

We will modify the process to call the **StockQuote** service and compare the actual price against the requested price. If the quote we get back matches or is better than the price specified, then we will return a successful trade (at the quoted price). Otherwise we will wait a minute and loop back round and try again.

Creating the while loop

The bulk of this process will now be contained within a while loop, so from the **Process Activities** list of the **Component Pallet** drag a **while** activity into the process.

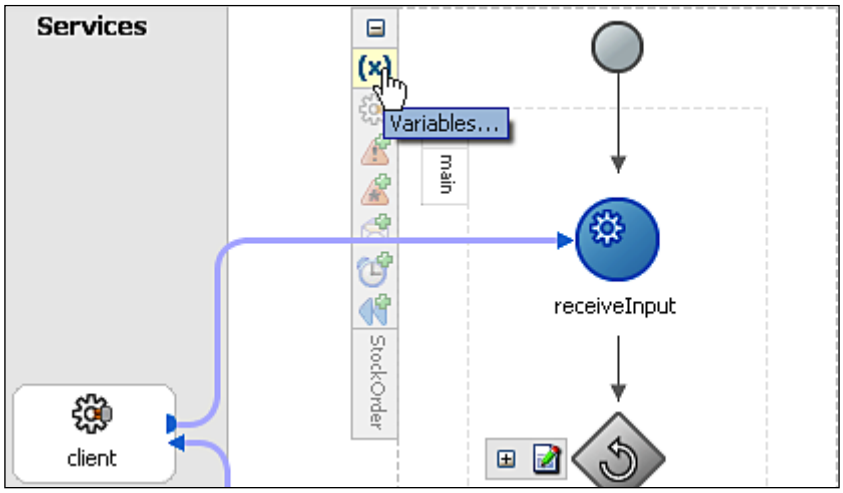
Click on the plus symbol to expand the **while** activity; it will now display an area where you can drop a sequence of one or more activities that will be executed every time the process iterates through the loop.



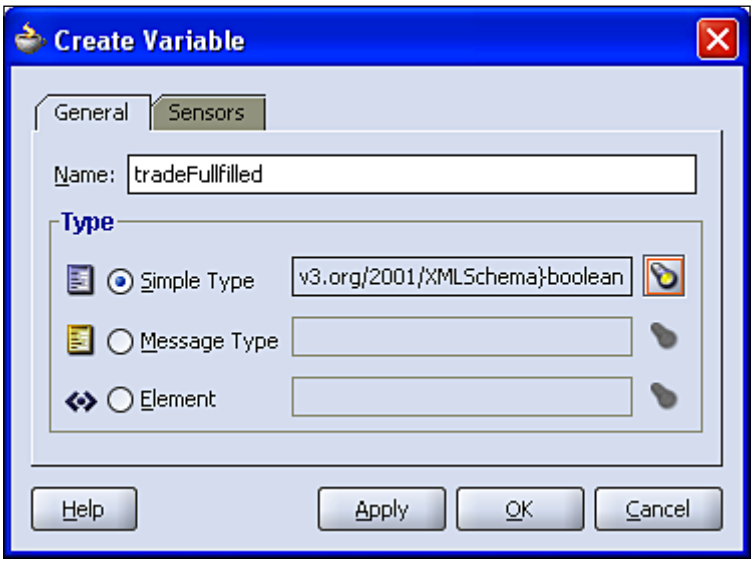
We want to iterate through the loop until the trade has been fulfilled, so let's create a variable of type `xsd:Boolean` called `tradeFullfilled` and use an assign statement before the while loop to set its value to `false`.

The first step is to create a variable of type `xsd:Boolean`. Up to now we've used JDeveloper to automatically create the variables we've required, typically as part of the process of defining an `invoke` activity. However, that's not an option here.

If you look at the diagram of your BPEL process you will see that it is surrounded by a light grey dashed box, and on the top left hand side are a number of icons. If you click on the top one of these icons **(x)** (as shown in the following screenshot), this will open a window which lists all the variables defined to the process.



At this stage it will list just the default `inputVariable` and `outputVariable` which were automatically created with the process. Click on the **Create** button; this will bring up the **Create Variable** window, as shown in the following screenshot:

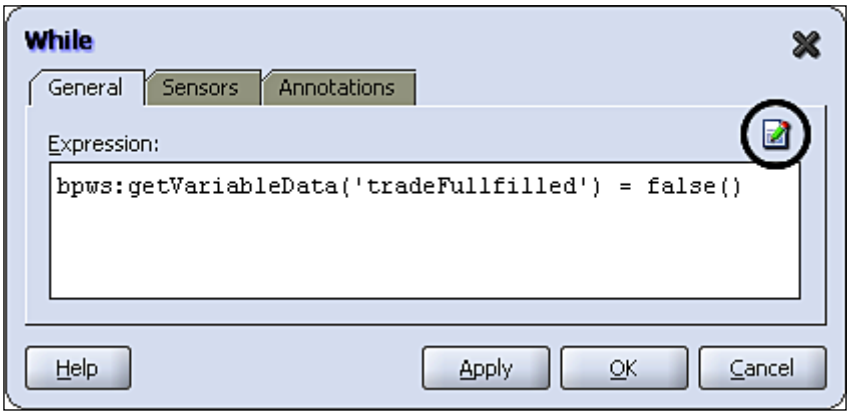


Here we simply specify the **Name** of the variable (for example `tradeFullfilled`) and its type. In our case we want an `xsd:Boolean`, so select **Simple Type** and click the flashlight to the right of it.

This will bring up the **Type Chooser**, which will list all the simple built in data types defined by XML Schema. Select **Boolean** and then click **OK**.

We need to initialize the variable to false, so drag an assign statement on to your process just before the while loop. Use the function `false()`, under the category **Logical Function** to achieve this.

Next we need to set the condition on the while loop, so that it will execute only while `tradeFulfilled` equals `false`. Double click on the **while** loop; this will open the **While** activity window, as shown in the following screenshot:



We must now specify an XPath expression which will evaluate to either `true` or `false`. If you click on the expression builder icon (circled in the previous screenshot), this will launch the Expression builder. Use this to build the following expression:

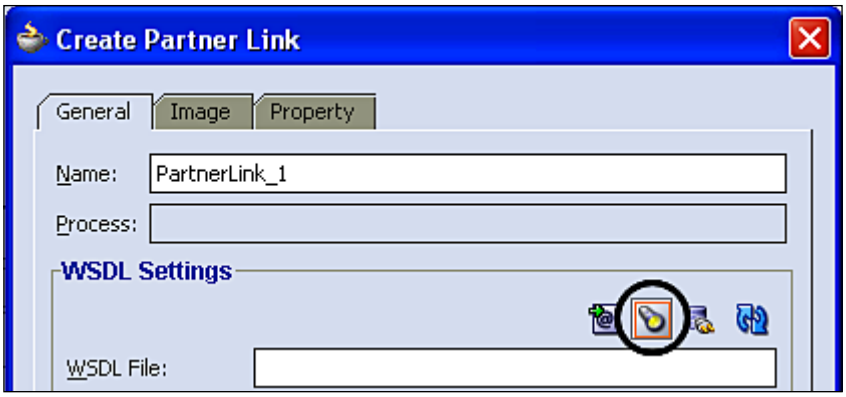
```
bpws:getVariableData('tradeFullfilled') = false()
```

Once we are happy with this click **OK**.

Checking the price

The first activity we need to perform within the `while` loop is to get a quote for the stock that we are trading. For this we will use the stock quote process we created earlier.

The approach is very similar to the one used when calling an external web service (as we did when implementing the StockQuote process). Create a Partner Link as before, but this time click on the **Service Explorer** icon, circled in the following diagram:



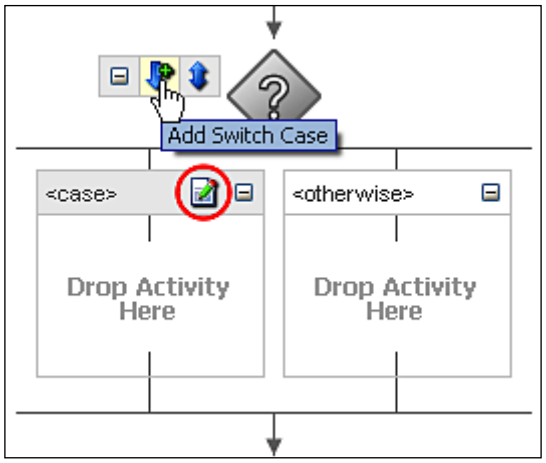
This will launh the **Service Explorer** window, which allows us to browse all the services defined to the SOA Suite. Expand the **BPEL Services** node and locate the **StockQuote** process as shown below in the following screenshot. Select this and click **OK**.

From here implement the required steps to invoke the **process** operation of the **StockQuote** process, making sure that they are included within the `while` loop.

Using the Switch activity

Remember our requirement is that we return success if the price matches or is better than the one specified in the order. Obviously whether the price is better depends on whether we are selling or buying. If we are selling we need the price to be equal to or greater than the asking price; whereas if we are buying we need the price to be equal to or less than the asking price.

So for this we will introduce the `<switch>` activity. Drag a `<switch>` activity from the **Process Activities** list of the **Component Pallet** on to your process, and then click on the plus symbol to expand the `<switch>` activity. By default it will have two branches as illustrated in the following screenshot:



The first branch contains a `<case>` condition, with a corresponding area where you can drop a sequence of one or more activities that will be executed if the condition evaluates to `true`.

The second branch contains an `<otherwise>` sub-activity, with a corresponding area for activities. The activities in this branch will only be executed if all case conditions evaluate to `false`.

We want to cater for two separate tests (one for buying, the other for selling), so click on the **Add Switch Case** arrow (highlighted in the previous diagram) to add another `<case>` branch.

Next we need to define the test condition for each `<case>`. To do this, click on the corresponding **ExpressionBuilder** icon to launch the expression builder (circled in the previous screenshot). For the first one use the expression builder to create the following:

```
bpws:getVariableData (  'inputVariable','payload',
                        '/ns1:PlaceOrder/ns1:BuySell') = 'Buy' and
bpws:getVariableData (  'inputVariable', 'payload',
                        '/ns1:PlaceOrder/ns1:BidPrice') >=
bpws:getVariableData (  'StockQuoteOutput', 'payload',
                        '/ns3:GetQuoteResult/ns3:Amount')
```

For the second branch, use the expression builder to define the following:

```
bpws:getVariableData (  'inputVariable','payload',
                        '/ns1:PlaceOrder/ns1:BuySell') = 'Sell' and
bpws:getVariableData (  'inputVariable', 'payload',
                        '/ns1:PlaceOrder/ns1:BidPrice') <=
bpws:getVariableData (  'StockQuoteOutput', 'payload',
                        '/ns3:GetQuoteResult/ns3:Amount')
```

Once we have defined the condition for each case, we just need to create a single `<assign>` activity in each branch. This needs to set all the values in the `outputVariable` to the corresponding values in the `inputVariable`, except for the `ActualPrice` element, which we should set to the value returned by the StockQuote process. Finally we also need to set `tradeFullfilled` to `true`, so that we exit the `while` loop.

The simplest way to do this is drag the original `<assign>` we created in the first version of this process and drag it onto the first branch and then modify it as appropriate. Then create a similar assign activity in the second.

You've probably noticed that you could actually combine the two tests into a single test; however we took this approach to illustrate how you can add multiple branches to a switch.

If we don't have a match, then we want to wait a minute and then circle back round the `while` loop and try again. As we've already defined a `<wait>` activity, simply drag this from its current position within the process, into the Activity area for the `<otherwise>` activity.

That completes the process, so try deploying it and running it from the console.

The other obvious thing is that this process could potentially run forever if we don't get a stock quote in our favor. One way to solve this would be to put the while activity in a scope and then set a timeout period on the scope so that it would only run for so long.

Summary

In this chapter we've gone beyond individual services and looked at how we can use BPEL to quickly assemble these services into composite services. Using this same approach we can also implement end-to-end business processes or complete composite applications (something we will do in the second section of this book).

You may have also noticed that although BPEL provides a rich set of constructs for describing the assembly of a set of existing services, it doesn't try to reinvent the wheel where functionality is already provided by existing SOA standards. Rather it has been designed to fit naturally with and leverage the existing XML and web services specifications, such as XML Schema, XPath, XSLT, and of course WSDL and SOAP.

This chapter should have given you a solid introduction into the basic structure of a BPEL process, its key constructs, and the difference between a synchronous and asynchronous service. Building the examples will help to re-enforce this as well as give you an excellent grasp of how to use JDeveloper to build BPEL processes.

Even though this chapter will have given you a good introduction to BPEL, we haven't yet looked at much of its advanced functionality, such as its ability to handle long-running processes, its fault and exception management, and how it uses compensation to undo events in the case of failures. These are areas we will cover in more detail within later chapters of this book.