

# Discover the business logic of BPEL4WS

Skill Level: Intermediate

[Frank Cohen \(fcohen@pushtotest.com\)](mailto:fcohen@pushtotest.com)

Founder  
PushToTest

[David Nielsen \(david@persistent.com\)](mailto:david@persistent.com)

Technical Consultant  
Persistent Web Technologies

20 Jun 2003

This tutorial illustrates the function and benefits of the BPEL4WS specification. The tutorial then explores the application of BPEL4WS in a real-world business process.

## Section 1. Introduction

### Getting started

Wide-spread enterprise adoption of SOAP and WSDL is underway. With that, system architects and software developers are concentrating their attention on ways to efficiently link groups of Web Services into business workflows. In an ideal world, a meta language handles all the mundane tasks of making calls to SOAP-based services and provides the language constructs (looping, conditionals, variables, and interfaces to the local system resources) to write workflow-based software applications. For example, a bank setting a loan application goes through several steps of approval. Each approval comes from a SOAP-based Web service request. Software developers use the workflow language to identify the order of the steps for approval and how the workflow branches depending on the results of each Web service response.

BPEL4WS (known as BPEL and pronounced bee-pel) is a combination of efforts by 3 major enterprise application server vendors. IBM and Microsoft combined their

previous individual efforts -- IBM's Web Services Flow Language (WSFL) and Microsoft's XLANG specification -- to create the BPEL4WS specification. In addition, BEA Systems, itself a co-author of Web Services Choreography Interface (WSCI, pronounced whiskey), a competitive business process specification, joined the effort. The effort drew support from Siebel and SAP. Recently, Sun Microsystems joined the OASIS WSBPEL technical committee to further the effort.

Automating the interaction of business processes between applications, departments and enterprises recently took a giant leap forward when IBM, Microsoft and BEA announced that they intend to submit version 1.1 to the Organization for the Advancement of Structured Information Standards (OASIS) technical committee under royalty-free terms. The news is reminiscent of the traction XML Web Services received when IBM and Microsoft teamed to propose SOAP to the W3C in September of 2000.

## Advantages of BPEL4WS

There are two reasons why developers should look at BPEL4WS right now:

1. For developers that have already adopted SOAP-based Web services, BPEL4WS delivers immediately useful code that you do not have to write. When your application requires several Web service calls with business logic to make choices then your application is a candidate to use BPEL4WS. One call to a BPEL4WS flow encapsulates the many Web service calls your application would otherwise make to process a workflow. BPEL4WS avoids your needing to write workflow code in Java in a servlet or Enterprise Java Bean (EJB) component.
2. For developers that have avoided SOAP-based Web services, BPEL4WS makes the Web service platform more attractive. BPEL4WS makes Web services more than a lightweight way to implement external APIs for your applications. With BPEL4WS Web services get closer to a rich platform for processing data according to business process requirements. Consequently, your Java code becomes less complex and more reliable as the BPEL4WS engine handles calls to SOAP-based Web Services and workflows based on their results.

To help you understand BPEL4WS, this tutorial illustrates the function and benefits of the BPEL4WS specification. The tutorial then explores the application of BPEL4WS in a real-world business process.

## Section 2. Loan application example

### The QuickApps Company

Here in California the real estate market is going very strong even during the downturn in the economy. Our local newspaper recently reported that demand for available homes for sale is high and prices appear to be holding. This is driving loan processors to handle historically high levels of loan applications. In this tutorial we will look into how the BPEL4WS technology facilitates loan processing at QuickApps, a fictitious title company.

QuickApps has optimized the flow of loan approvals so that with two loan officer approvals and a good credit rating a loan application is rapidly approved. The QuickApps IT team implemented this flow in an application named LoanBroker.

LoanBroker will show you how BPEL4WS enables the following:

- Identify a workflow transaction, known as a *business flow*
- Interface with existing resources using standard Web service protocols and email protocols
- Use both synchronous and message-style (asynchronous) methods
- Use local resources, including data storage, to make queries
- Handle error conditions and exceptional states.

### The QuickApps process

LoanBroker follows this process to approve each loan:

- 1) A new loan application is assigned a transaction number and is dated.
- 2) The loan application is checked for valid data using XML Schema data type definitions in a WSDL document.
- 3) The service checks the credit rating of the applicant and rejects applications where a negative credit rating exists.
- 4) In parallel the flow sends the application to two loan companies to receive their approval and proposed interest rate. The system waits for up to 24 hours for approval by both officers before rejecting the application.

5) With approval the system chooses the lowest available interest rate and extends an offer to the applicant.

To accomplish this workflow this tutorial makes use of these resources:

- A synchronous credit rating service takes a social security number as input and returns a credit rating.
- Asynchronous services that request the loan officer to approve the application as input and returns the loan officer's approval signature and proposed interest rate in XML form.

All of the function of LoanBroker flow will be implemented using a BPEL4WS engine and scripted entirely in XML -- no Java code is necessary to build the QuickApps workflow system.

See [Resources](#) for all of the services used in this tutorial.

## Compiling and deploying

We begin by compiling and deploying the QuickApps BPEL business flow. The examples in this tutorial have been compiled using the Collaxa BPEL Orchestration Server. The same process code should work on any BPEL4WS-compliant 1.1 engine.

BPEL4WS defines a process to encompass an entire transaction. The LoanBroker process is defined in three files:

1. LoanFlow.bpel is a BPEL4WS source file.
2. LoanFlow.wsdl is a Web Service Description Language (WSDL) file describing the interface to the service itself and the XML types it uses.
3. LoanFlow.xml is a deployment descriptor that specifies where to find the WSDL files for services that are called by the BPEL process and other run-time configuration parameters.

## How to run the compiler

Several simple steps are needed to configure and run the BPEL4WS compiler. To make this easy, this tutorial provides an Apache Ant build script. Ant is a popular utility that uses an XML script to build and deploy applications. Follow these steps to run the Ant build script:

**Step 1.** Make sure the Collaxa Orchestration Server is running. On Windows 2000, start the Collaxa engine from the Start menu (**Start => Programs => Collaxa => Web Service Orchestration Server 2.0 => Start Collaxa Server**).

## How to run the compiler (continued)

**Step 2.** Collaxa provides *cxant*, a shell script to set environment variables and call the Ant build system. From the LoanFlow directory run *cxant* in a command prompt. This compiles and deploys the process and its dependent services, as shown here:

```
prompt>cd c:\QuickApps\LoanFlow
prompt>cxant
...
AsyncLoanService:
[schemac]
schemac> parsing schema file
'C:\collaxa\cxdk\samples\05.AsyncLoanService\
AutoLoanTypes.xsd'
...
[schemac]
schemac> generating java files ...
[schemac]
schemac> compiling java files ...
...
BUILD
SUCCESSFUL
```

The BPEL4WS engine parsed the XML description of the LoanBroker process and built all of the Java code, SOAP code, WSDL definitions, and deployment descriptions needed to run the LoanBroker service. In addition to building the LoanBroker process, the Ant build script deploys the process to the running BPEL4WS engine. Next, the tutorial will spend some time showing you the QuickApps process running on the Collaxa server. Then it will show how the process XML is constructed.

## Starting a process

**Step 3.** Collaxa provides a browser-based user interface to see scenarios in operation. Open a browser to <http://localhost:9700/BPELConsole>, as shown in Figure 1.

## Figure 1. Open BPELConsole

**Step 4.** Click on the **LoanFlow** process to create a new instance of the LoanBroker approval business flow.

## Initiate the flow

### Figure 2. Complete the form

**Step 5.** Complete the form, as shown in Figure 2. The actual values are not checked for accuracy, so feel free to enter whatever you wish. Next, click **Initiate Flow**.

## Check the flow

### Figure 3. QuickApps processing the application

**Step 6.** In Figure 3, the Collaxa console shows that the QuickApps process instantiation is processing the application. Click the **Visual Flow** icon to see the progress of the loan application.

## See the progress

### Figure 4. Visual representation of BPEL business flow

**Step 7.** Scroll down to the bottom of the visual representation of the BPEL business flow to see the StarLoanService (see Figure 4). The console shows that the workflow is waiting for the StarLoanService loan officer to approve the loan. The display also shows that the UnitedLoanService loan officer already approved the loan.

## Review the loan application

**Step 8.** The next step is to switch roles and assume the identity of the loan officer at StarLoan. Point your browser to <http://localhost:9700/StarLoanUI>, as shown in Figure 5.

### Figure 5. Open StarLoanUI

**Step 9.** Click the **Review Loan Application** link.

## Complete the form

## Figure 6. Complete the form

**Step 10.** Enter an interest rate in the APR field (see Figure 6). By default the UnitedLoanService loan officer extended a rate of 10% to the applicant. The business logic built into the QuickApps scenario chooses between the UnitedLoanService and StarLoan service based on the lower interest rate offered. If you enter a value less than 8%, then the applicant's loan will be fulfilled by the StarLoan service. Click the **Approve** button.

## Learn the outcome

**Step 11.** Return to the BPEL console by pointing your browser to <http://localhost:9700/BPELConsole>, as shown in Figure 7.

## Figure 7. Back to BPELConsole

**Step 12.** Notice in the lower right portion of the console that the QuickApps scenario flow completed. Click on **1: Instance #1 of LoanFlow** to learn the outcome of the loan application.

## View the loan that won

## Figure 8. Visual representation of BPEL business flow

**Step 13.** Scroll down to the bottom of the visual representation of the history of this BPEL flow (see Figure 8). Notice that both loan officers were contacted and provided their approval to the loan. The flow determined which loan officer offered the better interest rate. Then the flow notified the loan applicant. To view which loan officer the flow chose, click the **Callback (receiveResult)** object.

## The best choice

## Figure 9. Flow choice for best interest

**Step 14.** The new browser window shows the flow's choice for best interest rate (see Figure 9). The flow ends by calling back to the service that instantiated it. This technique lets business flows become services that can be used to compose higher level flows. For example, after a flow calls the LoanBroker business flow, it can also initiate a flow to notify the applicant and mail the necessary forms to complete the overall transaction.

Next we will look into the BPEL4WS code we used to implement the LoanBroker process.

---

## Section 3. Digging into the code

### Behind the scenes

BPEL4WS processes use XML documents to define the location of services, the methods to call the services, and the flow of requests from one service to another. Using XML encoding makes BPEL4WS very extensible and flexible. The example given in the previous section makes the LoanBroker loan processing flow look easy. In fact, a lot is happening behind the scenes.

### LoanBroker defined

The LoanBroker process is defined in three files:

1. LoanFlow.bpel is a BPEL4WS source file.
2. LoanFlow.wsdl is a Web Service Description Language (WSDL) file describing the interface to the service itself and the XML types it uses.
3. LoanFlow.xml is a deployment descriptor that specifies where to find the WSDL files for services that are called by the BPEL Scenario and other run-time configuration parameters.

### Identifying the service descriptions

We will begin by looking at the contents of LoanFlow.xml. This document is a deployment descriptor that tells the BPEL4WS engine where to find the descriptions of the Web services that the flow will request. Here is the document in its entirety, followed by a detailed explanation:

```
<?xml version="1.0" encoding="UTF-8"?>
<bpel-process src="LoanFlow.bpel LoanFlow.wsdl">
  <!-- properties for partner "UnitedLoanService" -->
  <properties id="UnitedLoanService">
    <property name="wsdl-location">
```

```
        http://examples.pushtotest.com/collaxa/default/UnitedLoan?wsdl
    </property>
</properties>
<!-- properties for partner "StarLoanService" -->
<properties id="StarLoanService">
    <property name="wsdl-location">
        http://examples.pushtotest.com/collaxa/default/StarLoan?wsdl
    </property>
</properties>
<!-- properties for partner "creditRatingService" -->
<properties id="creditRatingService">
    <property name="wsdl-location">
        http://examples.pushtotest.com/collaxa/default/CreditRatingService?wsdl
    </property>
</properties>
</bpel-process>
```

## The flow in more detail

Now we will look at LoanFlow.xml in detail. The LoanFlow.xml document identifies three SOAP-based Web Services: UnitedLoanService, StarLoanService, and creditRatingService. The document begins by identifying the BPEL document and the WSDL description of the flow.

```
<bpel-scenario src="LoanFlow.bpel LoanFlow.wsdl">
```

Prior to XML, this file might have been a simple properties file containing name/value pairs. With XML you can identify additional attributes for each element without having to go back to the Java, C++, or other code that will use these property settings.

```
<!-- properties for partner "creditRatingService" -->
<properties id="creditRatingService">
    <property name="wsdl-location">
        http://examples.pushtotest.com/collaxa/default/CreditRatingService?wsdl
    </property>
```

```
</properties>
```

The QuickApps flow will use three SOAP-based Web services. The first of these is the `creditRatingService`. The `properties` element tells the BPEL4WS engine where to find the WSDL definition for the `creditRatingService`.

BPEL4WS leverages many of the strengths of SOAP-based Web services. Consider that the WSDL for the `creditRatingService` tells the BPEL4WS engine the encoding style of the requests and how to make the requests. The WSDL shows that the service expects a document-literal encoded SOAP request to pass the loan application data to a loan officer. The WSDL tells the BPEL4WS engine to expect a document-literal encoded SOAP response containing the details of the loan officer's approval.

## The `creditRatingService` WSDL

The `creditRatingService` WSDL also identifies the request method to make the request for approval using a message-style asynchronous call to the `creditRatingService`. Asynchronous calls use a request/call-back semaphore for handling transactions, including methods to initiate the request, poll the service for completion, and get the results upon completion. See [Resources](#) to view the entire WSDL for the `creditRatingService`.

```
<!-- properties for partner "UnitedLoanService" -->
<properties id="UnitedLoanService">
  <property name="wsdl-location">
    http://examples.pushtotest.com/collaxa/default/UnitedLoan?wsdl
  </property>
</properties>
<!-- properties for partner "StarLoanService" -->
<properties id="StarLoanService">
  <property name="wsdl-location">
    http://examples.pushtotest.com/collaxa/default/StarLoan?wsdl
  </property>
</properties>
```

The `UnitedLoanService` and `StarLoanService` services are SOAP-based message-style asynchronous services that provide loan officer approval functions. The entries for each service in `LoanFlow.xml` show the BPEL4WS engine where to find the WSDL descriptions for each service.

## Creating the flow

Now that you see how the LoanBroker flow identifies the external functions it will use, let's look into how the BPEL4WS language enables you to program the QuickApps flow. See [Resources](#) to find the LoanFlow.bpel code.

All BPEL4WS documents contain these elements:

`<process>` - Identify the namespace and WSDL information for the flow itself

`<partners>` - Declare the parties involved. Named partners are defined, each characterized by a WSDL serviceLinkType.

`<containers>` - Defines the activities that form the composition.

`<sequences>` - A set of steps that defines a transaction.

## The process element

These elements usually appear in the BPEL4WS document in this order, although the specification says they may appear in any order. Lets look at how the QuickAppsLoanFlow.bpel document declares these elements.

### `<process>`

First there is the `<process>` element.

```
<process name="LoanFlow"
  targetNamespace="http://samples.cxdn.com"
  suppressJoinFailure="yes"
  xmlns:tns="http://samples.cxdn.com"
  xmlns:tls="http://samples.cxdn.com"
  xmlns:crs="http://demo.cxdn.com"
  xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
>
```

The `<process>` element identifies the name of the flow and includes the namespaces that will allow it to refer to the required WSDL information, where the message definitions are defined (<http://samples.cxdn.com>), the target namespace of the loan approver (<http://samples.cxdn.com>), and the target namespace of the process's own WSDL (<http://demo.cxdn.com>). The process is now able to use the

LoanFlow approver service as a component.

## The partnerLink Element

```
<partnerLinks>
  <partnerLink name="client"
    partnerLinkType="tns:LoanFlow"
    partnerRole="LoanFlowRequester"
    myRole="LoanFlowService" />
  <partnerLink name="creditRatingService"
    partnerLinkType="crs:CreditRatingService"
    partnerRole="CreditRatingServiceService" />
  <partnerLink name="UnitedLoanService"
    partnerLinkType="tls:LoanService"
    myRole="LoanServiceRequester"
    partnerRole="LoanServiceService" />
  <partnerLink name="StarLoanService"
    partnerLinkType="tls:LoanService"
    myRole="LoanServiceRequester"
    partnerRole="LoanServiceService" />
</partnerLinks>
```

The `partnerlink` element declares the parties involved. Named partners are defined, each characterized by a WSDL `partnerLinkType`. For this example, the partners are the credit reporting service and loan officer financial institutions. The `myRole` and `partnerRole` attributes on a partner specifies how the partner and the process will interact given the `partnerLinkType`. The `myRole` attribute refers to the role in the `serviceLinkType` that the process will play, whereas the `partnerRole` specifies the role that the partner will play. You will see this play out in the partner definitions below.

## Variable elements for flow points

```
<variables>
  <!-- input of this process -->
  <variable name="input"
```

```
        messageType="tns:initiateLoanFlowSoapRequest" />
<variable name="crInput"
        messageType="crs:processCreditRatingServiceSoapRequest" />
<variable name="crOutput"
        messageType="crs:processCreditRatingServiceSoapResponse" />
<variable name="crError"
        messageType="crs:processCreditRatingServiceSoapFault" />
<!-- input of united loan and star loan-->
<variable name="loanApplication"
        messageType="tls:initiateLoanServiceSoapRequest" />
<!-- output of united loan-->
<variable name="loanOffer1"
        messageType="tls:onLoanServiceResultSoapRequest" />
<!-- output of star loan-->
<variable name="loanOffer2"
        messageType="tls:onLoanServiceResultSoapRequest" />
<!-- output of this process -->
<variable name="selectedLoanOffer"
        messageType="tns:onLoanFlowResultSoapRequest" />
</variables>
```

The LoanBroker flow is defined to provide an asynchronous message-based mechanism for approving loan applications. The variable elements provide an easy mechanism to refer to each Web service request. The script defines variable elements for each step to complete the overall transaction of getting the loan approved.

## Identifying the sequence

A process may contain only one activity, which in this case is the `<sequence>` element. The `<sequence>` element identifies how the BPEL4WS engine will receive activity that can take the customer's message and put it in the appropriate container. The definition of a receive activity must include the partner that will send it the message, and the port type and operation of the process to which the partner is targeting this message. When the process gets a message, it routes the request to an active receive activity that has a matching partner-portType-operation triplet and passes the message.

The BPEL4WS specification prevents receive activities with the same partner-portType-operation triplet that are active at the same time. In that case the activity places the message in the specified container and terminates.

```
<sequence>
  <receive name="receiveInput" partnerLink="client"
    portType="tns:LoanFlow"
    operation="initiate" variable="input"
    createInstance="yes"/>
```

The first part of the `<sequence>` receives a `loanApplication` message from requestor. The flow receives a loan application business document. This kicks off the flow.

## Copying The SSN

```
<assign>
  <copy>
    <from variable="input" part="parameters"
      query="//SSN"/>
    <to variable="crInput" part="parameters"
      query="/processCreditRatingService/ssn"/>
  </copy>
</assign>
```

Next the flow copies the Social Security Number (SSN) from the input `LoanApplication` container to the `creditRating` service input container.

## Invoking the services

```
<scope containerAccessSerializable="no">
```

The `<scope>` element identifies a group of actions that will invoke the synchronous `creditRatingService`, handle exceptions thrown and evaluate the applicants credit before moving on to seek approval from the loan officers.

## Handling exceptions

```
<faultHandlers>
  <catch faultName="crs:NegativeCredit"
    faultContainer="crError">
    <assign>
      <copy>
        <from expression="-1000"/>
        <to container="loanApplication"
          part="xmlLoanApp"
          query="/xmlLoanApp/CreditRating"/>
      </copy>
    </assign>
  </catch>
</faultHandlers>
```

The `<faultHandlers>` element watches for faults (exceptions) being thrown from `creditRatingService`. This defines a scope for handling faults from it and sets the credit rating in the loan application business document if you get a credit rating back. In the case of a `NegativeCredit` exception, set it to -1000.

## Invoking the CreditRating Service

```
<sequence>
  <invoke name="invokeCR"
    partnerLink="creditRatingService"
    portType="crs:CreditRatingService"
    operation="process"
    inputVariable="crInput"
    outputVariable="crOutput" />
```

Invoke the `CreditRating Service` -- the URL of this service's WSDL is specified in the deployment descriptor.

## Receive and handle the rating

```
<assign>
  <copy>
    <from variable="crOutput" part="parameters"
      query="//result"/>
    <to variable="input" part="parameters"
      query="/initiateLoanFlow/xmlLoanApp/CreditRating"/>
  </copy>
</assign>
```

Add the credit rating you received to the loan application business document.

## Wait For both loan officers

```
</sequence>
</scope>
```

The previous code in the `<sequence>` contacted the `creditRatingService` with the customer information for the applicant. The credit rating is copied to the application and sent on to two loan officers for approval. The flow needs to wait until both loan officers have replied.

## Initialize the input

```
<assign>
  <copy>
    <from variable="input" part="parameters"
      query="//xmlLoanApp"/>
    <to variable="loanApplication" part="parameters"
      query="/initiateLoanService/xmlLoanApp"/>
  </copy>
</assign>
```

The flow then initializes the input of the `UnitLoan` and `StarLoan` Web service requests to approve the loan.

## Invoke the UnitedLoanService

```
<flow>
  <sequence>
    <!-- initiate the remote service -->
    <invoke name="invokeUnitedLoan"
      partnerLink="UnitedLoanService"
      portType="tls:LoanService"
      operation="initiate"
      inputVariable="loanApplication"/>
    <!-- receive the result of the remote service -->
    <receive name="receive_invokeUnitedLoan"
      partnerLink="UnitedLoanService"
      portType="tls:LoanServiceCallback"
      operation="onResult"
      variable="loanOffer1"/>
  </sequence>
```

Now we will invoke the first loan provider at UnitedLoanService. The request is sent using a message-driven asynchronous Web service call. The `<invoke>` element tells the BPEL4WS engine to make the request immediately. The `<receive>` element waits until the response is received from UnitedLoanService.

## Invoke the StarLoanService too

```
<sequence>
  <!-- initiate the remote service -->
  <invoke name="invokeStarLoan"
    partnerLink="StarLoanService"
    portType="tls:LoanService"
    operation="initiate"
    inputVariable="loanApplication"/>
  <!-- receive the result of the remote service -->
  <receive name="receive_invokeStarLoan"
```

```
    partnerLink="StarLoanService"
    portType="tls:LoanServiceCallback"
    operation="onResult"
    variable="loanOffer2"/>
  </sequence>
</flow>
```

Without waiting for the first loan provider (UnitedLoanService), the second `<sequence>` element sends a message-driven asynchronous Web service call to StarLoanService. The `<invoke>` element tells the BPEL4WS engine to make the request immediately. The `<receive>` element waits until the response is received from StarLoanService.

## Choose the best loan

```
<switch>
```

Since the `<switch>` element is embedded in the parent `<sequence>` it knows to wait until both of the loan offer responses are received before evaluating the next stage. With both loan offer responses the `<switch>` element will evaluate which loan officer proposed a lower interest rate (Annual Percentage Rate, APR.)

```
<case condition="bpws:getVariableData('loanOffer1','parameters','result/APR')
> bpws:getVariableData('loanOffer2','parameters','result/APR') ">
<!-- Then take loanOffer2 -->
  <assign>
    <copy>
      <from variable="loanOffer2" part="parameters"
        query="result"/>
      <to variable="selectedLoanOffer"
        part="parameters"
        query="/onLoanFlowResult/result"/>
    </copy>
  </assign>
</case>
```

UnitedLoanService has the lower rate, so copy the loan officer approval into a result object.

## Save the approval

```
</case>
<otherwise>
  <assign>
    <copy>
      <from variable="loanOffer1" part="parameters"
        query="result" />
      <to variable="selectedLoanOffer"
        part="parameters"
        query="/onLoanFlowResult/result" />
    </copy>
  </assign>
</otherwise>
```

StarLoanService has the lower rate, so copy the loan officer approval into a result object.

## Send word to the customer

```
</switch>
<invoke name="replyOutput"
  partnerLink="client"
  portType="tns:LoanFlowCallback"
  operation="onResult"
  inputVariable="selectedLoanOffer" />
```

The final step sends the loan application, including the loan officer approval and credit rating, back to your client who kicked off the whole process.

## Exposing LoanBroker as a service itself

The BPEL4WS process model is based on a peer-to-peer interaction model that is described in WSDL. That is to say both the process and partner services are modeled as SOAP-based services that are described using WSDL. When you

compiled the LoanBroker service using a BPEL4WS engine, the compiled application included a WSDL document describing LoanBroker as a service.

This means LoanBroker not only runs as a flow by itself, LoanBroker can also be called by other flows. The QuickApps company can even expose the LoanBroker flow as a service to its partners with little extra effort.

Next we will describe the important elements of the WSDL document that describes LoanBroker. See [Resources](#) for the complete WSDL document.

## Defining the types used

```
<types>
<schema attributeFormDefault="qualified"
  elementFormDefault="qualified"
  targetNamespace="http://www.autoloan.com/ns/autoloan"
  xmlns="http://www.w3.org/2001/XMLSchema">
```

BPEL4WS WSDL documents use XMLSchema to define the complex data types to be used in the flow.

## Loan officer approval data type

```
<complexType name="LoanOffer">
  <sequence>
    <element name="ProviderName" type="string"/>
    <element name="Selected" type="boolean"/>
    <element name="Approved" type="boolean"/>
    <element name="APR" type="double"/>
  </sequence>
</complexType>
```

BPEL4WS uses document-literal encoding to marshal and unmarshal the data moved between Web services. In this case the response from a loan officer contains a complex data type representing the loan officer's approval, including a reference to the loan application and the interest rate offered.

## Bind the data types

The WSDL document includes a data type definition for each of the data objects exchanged in each Web service request and response.

```
<schema attributeFormDefault="qualified"
  elementFormDefault="qualified"
  targetNamespace="http://samples.cxdn.com"
  xmlns="http://www.w3.org/2001/XMLSchema">
  <element name="xmlLoanApp" type="s1:LoanApplication"/>
  <element name="result" type="s1:LoanOffer"/>
</schema>
```

The WSDL document defines then binds the complex data type to each SOAP-based Web service call. In this case the loan officer Web Service will make a request by sending the LoanApplication data and receive a LoanOffer object.

Next the WSDL document describes the interaction with the Web services and the code objects that will handle making flow decisions based on the responses.

## Binding the data types to the Web service

```
<schema
  targetNamespace="http://www.openuri.org/2002/04/soap/conversation/"
  xmlns="http://www.w3.org/2001/XMLSchema">
  <element name="StartHeader" type="conv:StartHeader"/>
  <element name="ContinueHeader"
    type="conv:ContinueHeader"/>
  <element name="CallbackHeader"
    type="conv:CallbackHeader"/>
  <element name="StartAcknowledgeHeader"
    type="conv:StartAcknowledgeHeader"/>
  <complexType name="StartHeader">
    <sequence>
      <element name="conversationID" type="string"/>
      <element name="callbackLocation" type="string"/>
    </sequence>
  </complexType>
</schema>
```

```
        </sequence>
    </complexType>
    <complexType name="ContinueHeader">
        <sequence>
            <element name="conversationID" type="string"/>
        </sequence>
    </complexType>
    <complexType name="CallbackHeader">
        <sequence>
            <element name="conversationID" type="string"/>
        </sequence>
    </complexType>
    <complexType name="StartAcknowledgeHeader">
        <sequence>
            <element name="referenceID" type="string"/>
        </sequence>
    </complexType>
</schema>
</types>
<message name="StartHeader">
    <part name="StartHeader" element="conv:StartHeader"/>
</message>
<message name="ContinueHeader">
    <part name="ContinueHeader"
        element="conv:ContinueHeader"/>
</message>
<message name="CallbackHeader">
    <part name="CallbackHeader"
        element="conv:CallbackHeader"/>
</message>
<message name="StartAcknowledgeHeader">
    <part name="StartAcknowledgeHeader"
        element="conv:StartAcknowledgeHeader"/>
</message>
```

```

<message name="initiateSoapRequest">
  <part name="xmlLoanApp"
    element="tns:xmlLoanApp" />
</message>
<message name="initiateSoapResponse" />
<message name="receiveResultSoapRequest">
  <part name="result"
    element="tns:result" />
</message>
<message name="receiveResultSoapResponse" />
<message name="pollResultSoapRequest" />
<message name="pollResultSoapResponse">
  <part name="result"
    element="tns:result" />
</message>
<slnk:serviceLinkType name="LoanFlowLT">
  <slnk:role name="LoanFlowService">
    <slnk:portType name="tns:LoanFlow" />
  </slnk:role>
  <slnk:role name="LoanFlowRequester">
    <slnk:portType name="tns:LoanFlowCallback" />
  </slnk:role>
</slnk:serviceLinkType>
<portType name="LoanFlow">
  <operation name="initiate">
    <input message="tns:initiateSoapRequest" />
    <output message="tns:initiateSoapResponse" />
  </operation>
  <operation name="pollResult">
    <input message="tns:pollResultSoapRequest" />
    <output message="tns:pollResultSoapResponse" />
  </operation>
</portType>

```

Each entry defines the Web service call to make and binds the result to an object.

## Define the call-backs

```
<portType name="LoanFlowCallback">
  <operation name="receiveResult">
    <input message="tns:receiveResultSoapRequest"/>
    <output message="tns:receiveResultSoapResponse"/>
  </operation>
</portType>
```

Finally, the WSDL document also describes call-back interfaces for asynchronous message-style Web service calls.

---

## Section 4. Conclusion

### Conclusion

In this tutorial we illustrated the function and benefits of the BPEL4WS specification. We considered the overall view of what BPEL4WS is about and then explained partners, faults, compensation, and lifecycle.

## Downloads

Description	Name	Size	Download method
Source code for LoanFlow bpel, wsdl, and xml	ws-bpeltutcode.zip	4 KB	<a href="#">HTTP</a>

[Information about download methods](#)

# Resources

## Learn

- Read the [Business Process Execution Language for Web services](#) specification.
- [Standards roadmap](#) -- understand the impact and importance of standards and specifications for the development of SOA and Web services.
- [SOA and Web services](#) -- hosts hundreds of informative articles and introductory, intermediate, and advanced tutorials on how to develop Web services applications.
- [Architecture: Build for the future](#) -- visit the Architecture area on developerWorks and get the resources you need to advance your skills in the architecture arena.

## Get products and technologies

- Download the [Collaxa BPEL Orchestration Server 2.0](#) from Collaxa.

## Discuss

- [developerWorks blogs](#) -- get involved in the developerWorks community.

# About the authors

## Frank Cohen

Frank Cohen is the "go to" guy for enterprises needing to test and solve problems in complex interoperating information systems, especially Web services. Frank is founder of [PushToTest](#), a test automation solutions business. Frank maintains TestMaker, a free open-source utility for building intelligent test agents to check Web services for scalability, performance and functionality. PushToTest Global Services customizes TestMaker to an enterprise's specific needs, conducts scalability and performance tests, and trains enterprise developers, QA analysts and IT managers on how to use the test environment for themselves. Details are at <http://www.pushtotest.com>. Contact Frank at [fcohen@pushtotest.com](mailto:fcohen@pushtotest.com).

---

## David Nielsen

David Nielsen is a technical consultant at Persistent Web Technologies

in San Francisco. He is also the Chair of the Software Development Forums Web Services Special Interest Group in Silicon Valley. He received a bachelors degree in Business Administration from California Polytechnic State University at San Luis Obispo. You can contact the author at [david@persistent.com](mailto:david@persistent.com).