Collaxa

last modified: September 6, 2002, ref: WSOS 2.0 beta 1

Collaxa WSOS 2.0: An Introduction

Abstract

This white paper describes how the Collaxa Web Service Orchestration Server and BPEL Scenario™ can integrate web services into collaborative business processes and long-running business transactions.

Audience

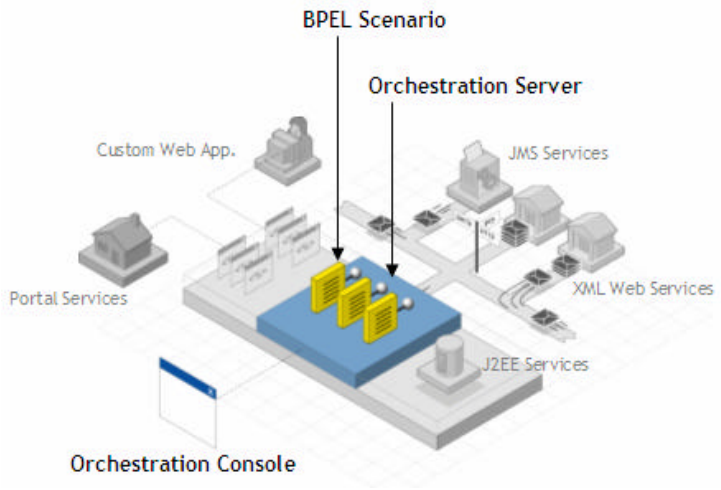
Java developers and architects, consultants and IT managers

Table of Contents

The 2-Minute Overview

Collaxa Web Service Orchestration Server

Collaxa offers a complete and standard-based solution to orchestrating web services into long-running transactions and collaborative business processes.



The **BPEL Scenario**™ is an innovative and flexible orchestration abstraction that enables developers to capture the flow, interaction logic and business rules that tie a set of services into an end-to-end business process.

The **Orchestration Server** encapsulates the facilities needed to execute BPEL Scenarios and guarantee the integrity of the long-running business transaction or collaborative business process.

The **Orchestration Console** provides administration, debugging capabilities and activity monitoring to help enterprises manage distributed business processes.

Standards and interoperability

The Collaxa WSOS is based, soup-to-nuts, on open standards (XML, SOAP, WSDL, BPEL4WS, BTP, WS-Coordination and WS-Transaction) and interoperates with Microsoft .Net, IBM WebSphere and BEA Workshop services.

Key Features

- Asynchronous Conversation (callback, correlation, state management)
- Flow Coordination (parallel and dynamic branching, sophisticated join patterns)
- Business Transaction Management (WS-Coordination, WS-Transaction)
- Hierarchical Exception Management
- XML Web Services (SOAP over HTTP, JMS, SMTP).
- User Interactions (manual tasks, portal integration, email notification, LDAP integration)
- Bi-directional Java-to-XML Mapping (based on XML Schema)
- Asynchronous Flow Debugger
- Audit Trails
- Business Activity Monitoring
- Management Console and API
- Side-by-side Versioning
- Clustering

What is Web Service Orchestration?

WEB SERVICES, STANDARDS FOR INTEROPERABILITY

As we use it here, the term "Web Services" refers to a set of open standards (XML, SOAP, WSDL, etc.) that increase interoperability and simplify integration across heterogeneous systems throughout the extended enterprise (See Figure 1).

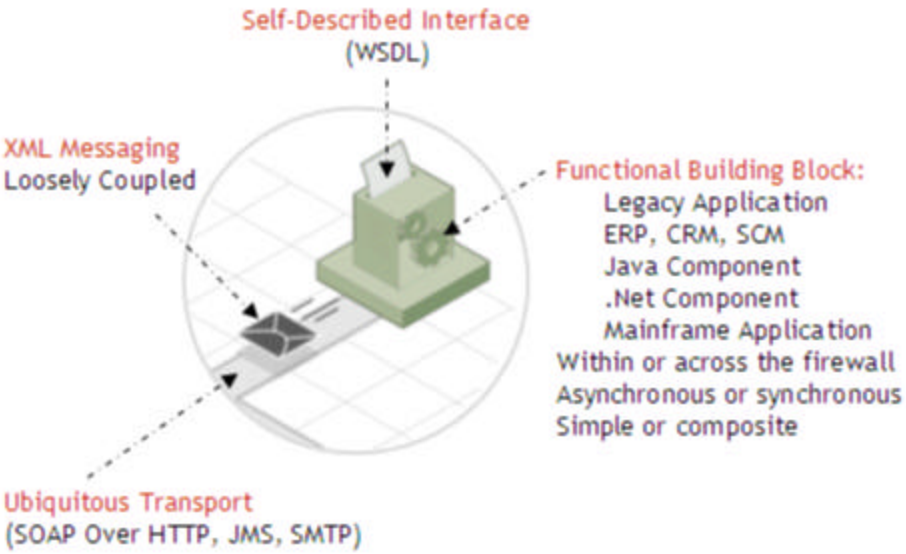


Figure 1 - Web Services: Industry Standards for Interoperability

The same way standards like HTTP and HTML revolutionized the way people access content and applications, Web Services have the potential to transform networked, heterogeneous systems into a true distributed computing platform and allow systems and people to cooperate simply and reliably.

MAKING WEB SERVICES WORK TOGETHER

Making Web Services work is a 2-step process: first you publish them and then you orchestrate them. Publishing means taking a piece of functionality that already exists within a system like a mainframe, a Java application, an ERP system or a .Net component and making it available over the network so that it can be easily integrated into other applications. Orchestration means composing multiple services into a long-lived, multi-step business transaction.

Let's take the example of a Loan Procurement application:

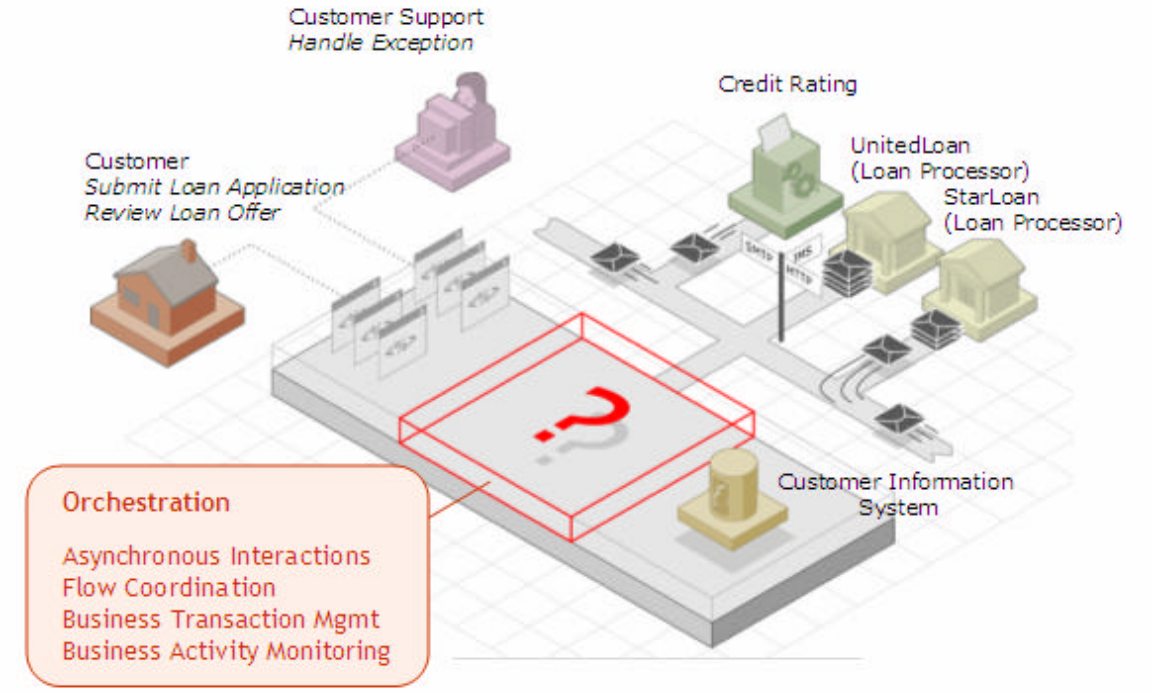


Figure 2 - Composite Loan Procurement Application

Once all the building blocks (Credit Rating, Customer Information Systems, Loan Processors, etc.) are published as JMS or XML Web Services, developers work with business analysts to integrate them into a long-lived, multi-step loan procurement process. The requirements that need to be implemented as part of the fabric that ties together those building blocks can be broken down into 4 categories: Asynchronous Interactions, Flow Coordination, Business Transaction Management and Activity Monitoring.

Requirement #1: Stateful and Asynchronous Interactions

→ Each building block has its own processing performance: The portal might be able to handle 100 concurrent accesses whereas the credit rating system which is based on older technology can only handle 20.

→ Each building block has its own lifecycle and maintenance requirements: UnitedLoan is offline for maintenance 3 hours per month.

→ Some services require manual intervention for knowledge contribution, approval or exception management: StarLoan might require a loan advisor to manually approve some class of loan requests.

Given that context, connecting those building blocks synchronously would result in a very brittle application. Asynchronous, message-driven interactions can overcome those limitations: services talk to each other using messages that get queued and can be processed asynchronously, thereby increasing reliability and throughput[1].

[1] Compare the number of emails you can handle per day with the number of phone conversations you can have.

However, asynchrony puts the burden on developers to handle callbacks, message correlation, conversational state, expiration, etc… It also fragments the application's business logic into small, difficult-to-read, difficult-to-reuse and difficult-to-manage code fragments. These are not trivial requirements given that most current applications are built using procedural programming techniques.

Requirement#2: Flow Coordination

Given that it might take a day for the loan processor to process a loan application and return an offer, the application has to interact with both processors in parallel. How do you branch your application to make sure that the loan applications are submitted to both loan providers in parallel? How do you handle sophisticated join patterns such as: canceling all pending requests/branches when the user approves one of the offers.

Requirement #3: Business Transaction Management

Some business rules require "atomic" semantics across loosely-coupled services. In addition, exceptions must be handled gracefully which can be one of the most challenging aspects of orchestration logic:

→ Building block services can throw faults: "The submitted SSN is invalid".

→ Building block services can timeout: A credit rating was supposed to be issued within 30 minutes but has not been received!.

→ A travel reservation use case provides a good example of transaction semantics: "Book the flight on United if and only if we can also book a room and a car".

Managing business transactions and exceptions in a loosely-coupled and asynchronous environment increases significantly the size, complexity and variability of the orchestration fabric.

Requirement #4: Business Activity Monitoring

Implementing the fabric that coordinates service interactions is only half the problem. Given the heterogeneous and distributed nature of the application, administration and monitoring are as important, if not more important, when it comes time to deploy the composite application in production. This raises such questions as:

→ How do administrators manage/cancel transactions?

→ What is the status and history of a specific transaction?

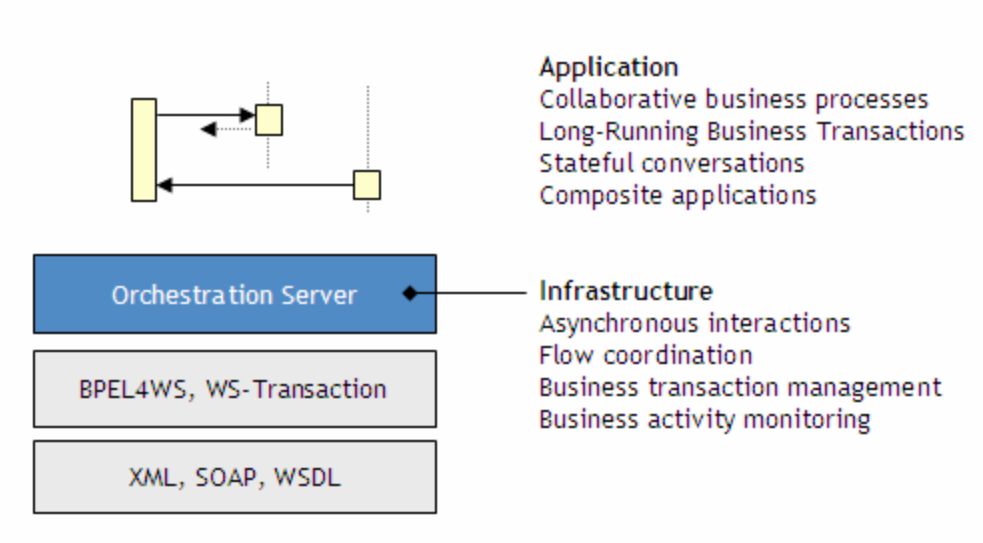
→ How do you add a new loan processing partner?

→ How do you implement side-by-side versioning?

→ How do you create a dashboard that allows business users to monitor the number of transactions initiated each week, the average offer and response time from each loan processor, etc…

DEFINING ORCHESTRATION

The orchestration requirements (asynchronous interactions, flow coordination, business transaction management and activity monitoring) are common to all applications that need to coordinate multiple synchronous and asynchronous web services into a multi-step business transaction. Implementing them in custom code as part of each service-oriented application is complex and difficult to maintain.



A new set of Web Services standards (BPEL4WS, WS-Transaction and WS-Coordination) and a new category of software infrastructure called the Web Service Orchestration Server are emerging to reduce the cost and complexity associated with delivering and managing these types of distributed, process-centric, service-oriented applications.

The Collaxa Web Service Orchestration Server

The Collaxa Web Service Orchestration Server helps enterprises reduce the cost and complexity of orchestrating web services into long-running business transactions and collaborative business processes. It is based on interoperability standards (such as XML, SOAP, WSDL, BPEL4WS, WS-Coordination, WS-Transaction and JMS) and works with your existing IT infrastructure including portals, application servers and messaging infrastructure.

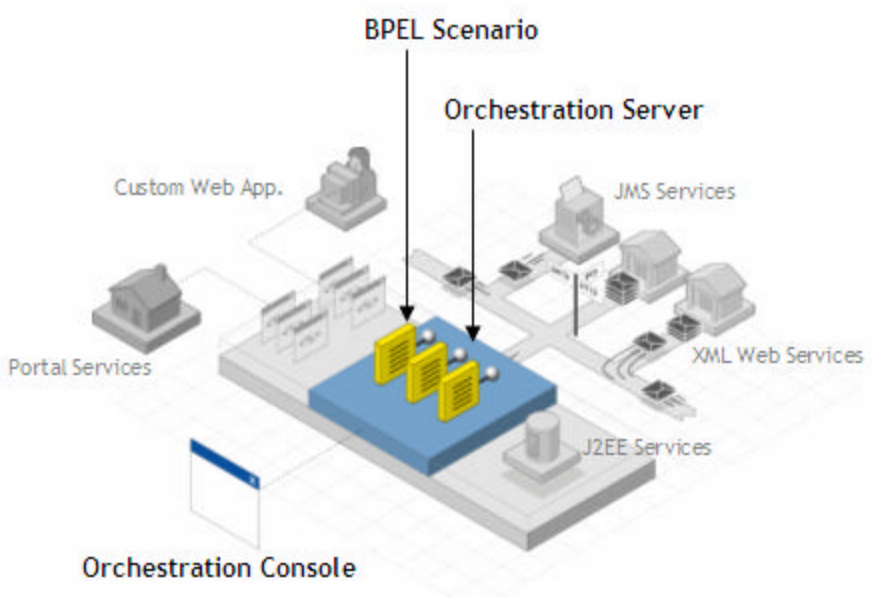


Figure 3 - Collaxa Web Service Orchestration Server

The Collaxa Web Service Orchestration Server is a 3-part solution:

- The **BPEL Scenario** is an innovative and flexible orchestration abstraction that captures the flow, interaction logic and business rules that tie a set of services into an end-to-end business process.
- The **Orchestration Server** encapsulates the facilities needed to deploy BPEL Scenarios and execute long-running business transactions or collaborative business processes.
- The **Orchestration Console** provides administration, testing and debugging capabilities and activity monitoring to help enterprises manage distributed business processes.

BPEL SCENARIO

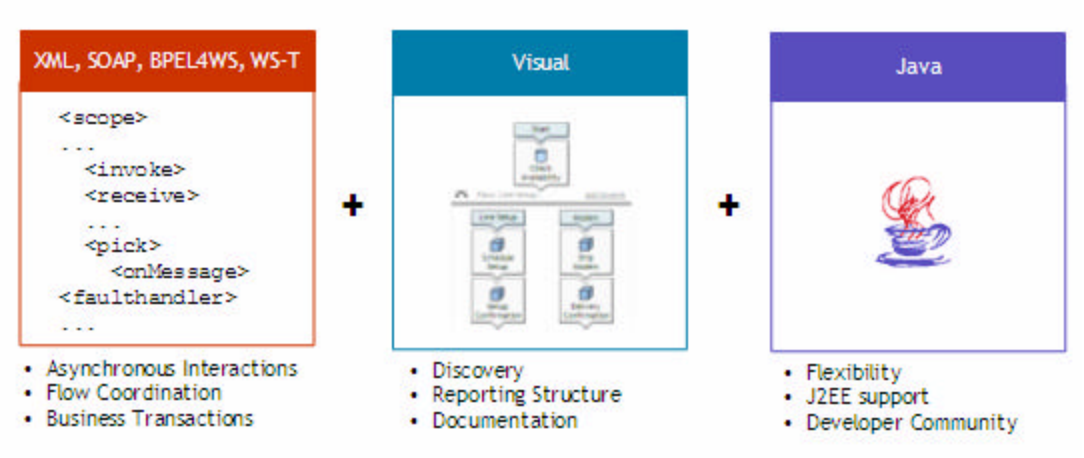


Figure 4 - BPEL Scenario, a Unifying Abstraction

The BPEL Scenario has been designed to combine the semantics and standards of BPEL4WS, the discovery and reporting capabilities of a visual model and the flexibility and power of Java into a unifying, developer-friendly orchestration abstraction.

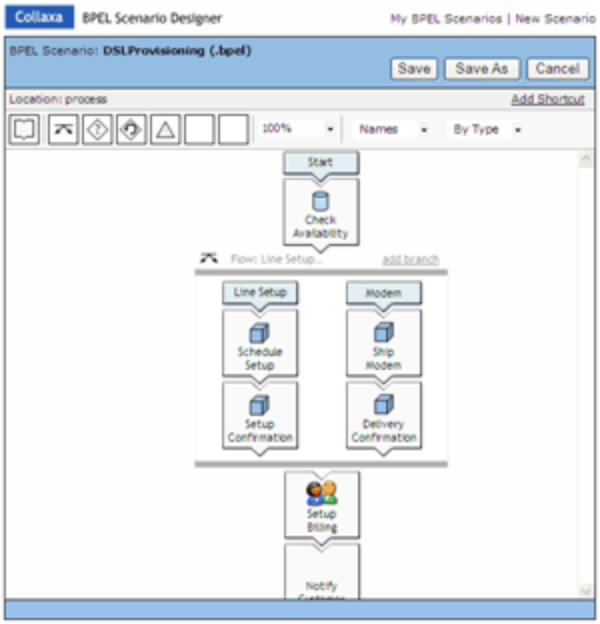


Figure 5 - Collaxa BPEL Scenario Designer, Business Analyst's View

The BPEL Scenario Designer is a visual tool that allows business analysts to capture and document the requirements of a BPEL Scenario. Built-in wizards guide users through the definition of the different parts of a scenario: partners, containers, flows, compensation rules and exception handlers. The visual model is also leveraged at run-time for audit trailing and reporting.

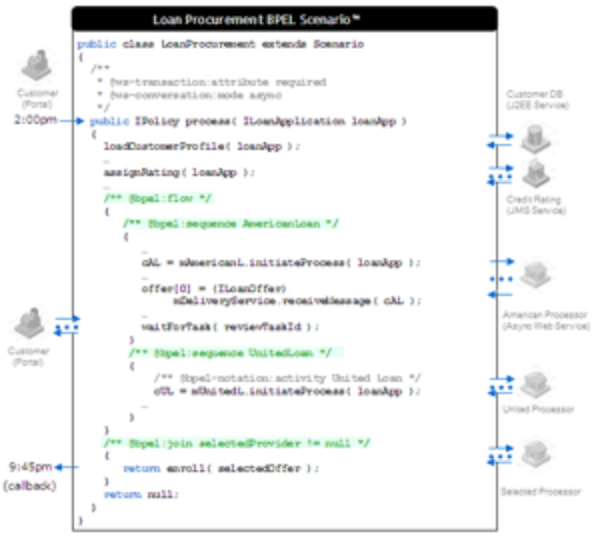


Figure 6 - BPEL Scenario, Developer's View

If necessary, Java Programmers use the developer's view of the BPEL Scenario, which is a Java-based, JSP-like abstraction, to complete the implementation.

The BPEL Scenario is then deployed to the Web Service Orchestration Server where it can be invoked through its XML Web Service, JMS or Java interfaces.

WEB SERVICE ORCHESTRATION SERVER

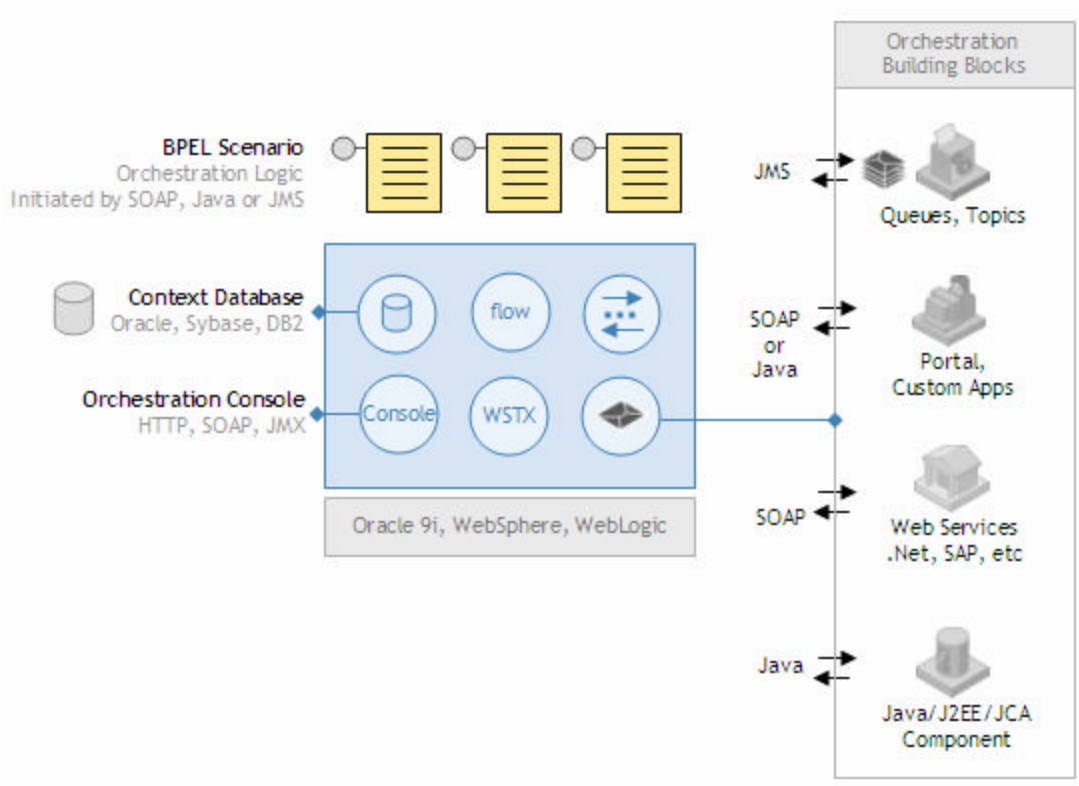


Figure 7 - Internals of the Orchestration Server

The Web Service Orchestration Server provides the run-time facilities need to deploy, execute and manage BPEL Scenarios:

- The **scenario manager** detects when a running scenario is waiting for an asynchronous callback, bookmarks its state and passivates it in a database. When the callback is received, the scenario is reactivated and its execution resumed. This looks to the developer as if the scenario was running in a persistent and fault-tolerant thread.
- The **flow coordinator** manages static and dynamic branching and sophisticated join-patterns. Branches can be arbitrarily nested.
- The **conversation manager** handles asynchronous interactions including message correlation and state management.
- The **delivery service** manages reliable message exchanges between the orchestration server and web service participants. It supports both HTTP and JMS. The delivery service manages all callbacks (aka asynchronous notifications).
- The **transaction manager** coordinates the transactional semantics of each BPEL Scenario. A BPEL Scenario can be both a transaction coordinator and transaction participant.
- The **console** provides visibility into the execution of scenario instances and the orchestration server.

  The complexity of these modules and facilities is hidden behind the simplicity of the BPEL Scenario.

WEB SERVICE ORCHESTRATION CONSOLE

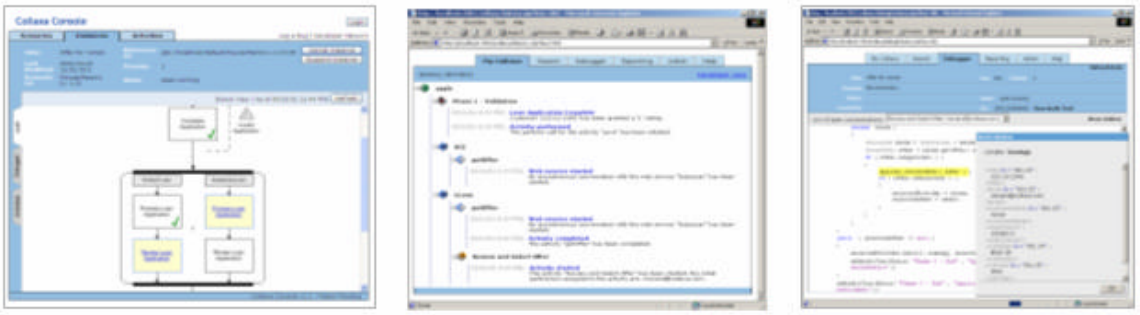


Figure 8 - Orchestration Console, Audit Trails and Debugging

Once deployed, BPEL Scenarios automatically benefit from the testing, administration and monitoring tools provided by the orchestration server. Each tool provides a different view into the execution of a scenario:

- The **debugger** allows the developer to select a running scenario, view its call stack and inspect its variables and XML documents.
- The **audit trail** provides the history of a scenario.
- The **conversation log** is a swimline view of the messages sent and received by the scenario.
- The **transaction log** shows the list of transactions and participants associated with each scenario and their state.

Example: A Loan Procurement BPEL Scenario

In this example, we will review how the Collaxa Web Service Orchestration Server and the BPEL Scenario can be used to integrate various web services, JMS queues and user interactions into a long-running Loan Procurement process.

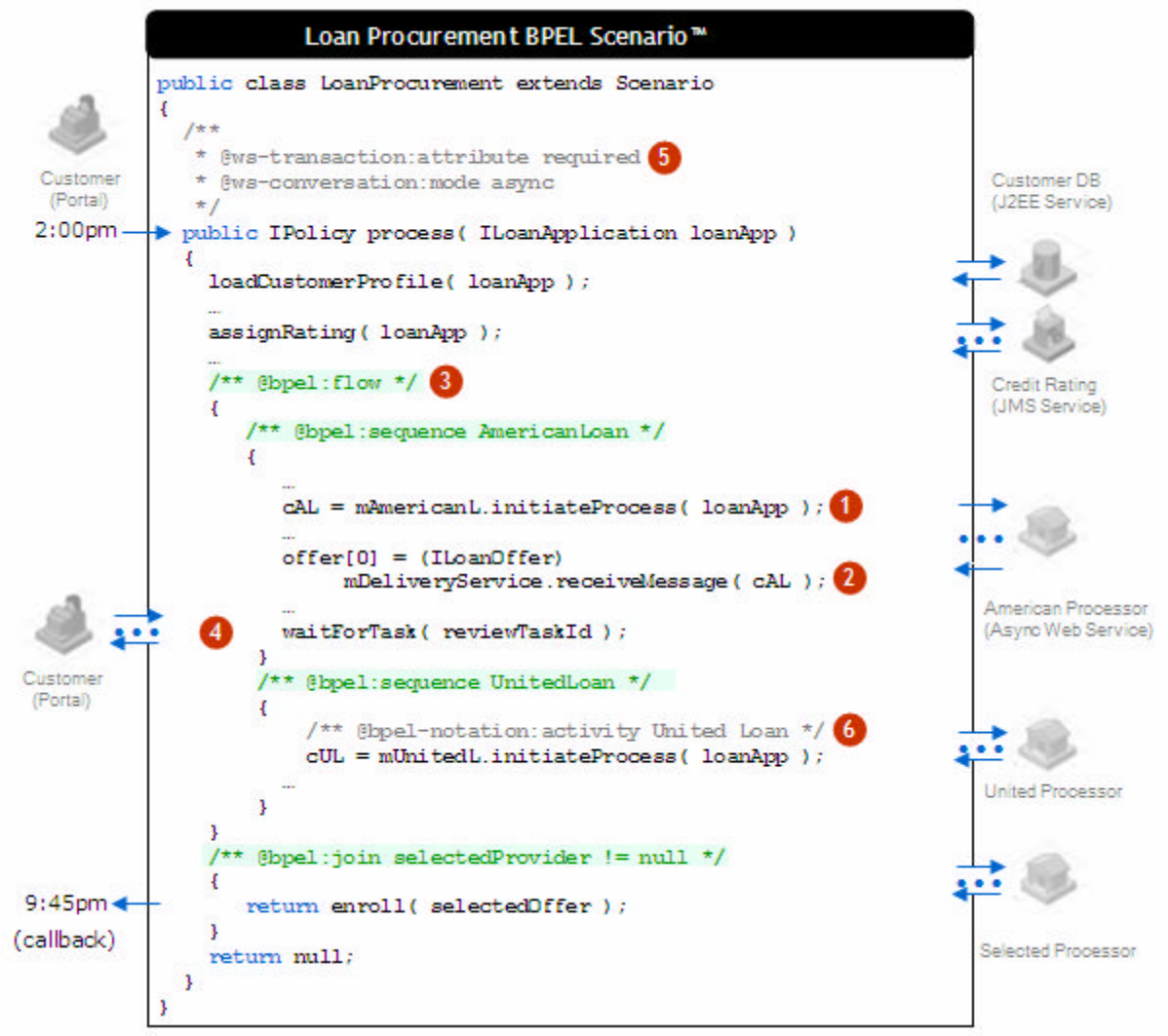


Figure 9 – LoanProcurement BPEL Scenario™ (Developer View)

Like JSPs, BPEL Scenarios are based on Java and increase the level of congruence between a problem domain and the solution domain: a JSP is a programming abstraction of a dynamic web page; a BPEL Scenario is a programming abstraction of a long-running business transaction or a collaborative business process. In both cases, “what you think is what you code”.

Let’s use the LoanProcurement described in Figure 9 to highlight some of the features of the BPEL Scenario:

1. **Web Service Invocation (XML Schema, SOAP, WSDL)**
   The ScenarioBean abstraction hides the complexity of invoking web services:
   ```
   mAmericanL.initiateProcess( loanApp );
   ```
   Both XML documents described by XML Schema as well as Web Services described by WSDL are wrapped and exposed as Java objects, hiding Java to XML mapping, low level DOM manipulation and SOAP marshalling.

2. **Asynchronous Interactions**
   Waiting for callback notification is managed and hidden inside the orchestration server. More specifically when invoking the delivery service
   ```
   offer = (ILoanOffer) deliverService.receiveMessage( c );
   ```
   the server detects that the scenario is waiting for a callback notification, pauses the execution of the scenario and waits until the offer is received by the delivery service. This operation could last 10 seconds or 10 days.

3. **Flow Coordination**
   In this example, we use the `/** @bpel:flow */` and `/** @bpel:join */` BPEL orchestration tags to specify that the AmericanLoan and UnitedLoan services are to be invoked in parallel. This is especially important as both loan processors process the loan application asynchronously - parallel branching shortens the time the customer has to wait before seeing both offers.

4. **User Interaction/Portal Integration**
   The task concept and built-in worklist service represent interactions with end-users through any UI; such as a custom web front-end, a portal or email interface. When waiting for the customer to review each offer:
   ```
   waitForTask( reviewTask );
   ```
   the orchestration server detects that the scenario is waiting for an asynchronous operation to complete. It pauses the execution of the scenario and passivates it until the task is completed. Tasks create breakpoints that interrupt the flow of the scenario and allow users to intervene, providing information, making decisions or managing exceptions.

5. **Business Transactions (ws-coordination, ws-transaction)**
   The BPEL Scenario lets developers assign transactional semantics to each operation. The resulting business transactions are coordinated and managed by the orchestration server using ws-coordination and ws-transaction.

6. **Visual Annotation, Audit Trail and Reporting**
   JavaDoc annotation comments `/**bpel-notation:…*/` capture meta information regarding the automated business process. This meta information is used to generate visual representations of scenarios, audit trails and business reports.

7. **Publishing Asynchronous and Complex Web Services**
   When deployed, a BPEL Scenario is automatically published as a Web Service and a WSDL file is generated.

INITIATING A BPEL SCENARIO

Before jumping into the details of what the developer view of a BPEL Scenario looks like, let's review how BPEL Scenarios are exposed to the outside world and how they are initiated.

BPEL Scenarios are deployed to an orchestration server where they can be initiated in one of four ways:

1. The *business delegate*[2] *interface* of the BPEL Scenario is used by Java components such as JSPs and Servlets.
2. The *Web Service interface* of the BPEL Scenario is used by SOAP clients. When deployed, a BPEL Scenario is automatically published as a Web Service and a WSDL file is generated. Therefore, BPEL Scenarios can be easily invoked by other BPEL Scenarios but also by a Visual Basic application, an Excel spreadsheet or another J2EE application.
3. The *JMS interface* of a BPEL Scenario can be used to initiate scenario processing based on receipt of a message to a JMS queue in a manner similar to the way Message Driven Beans work. For more information, see the JMS chapter in the WSOS Developer Guide.
4. The *HTML Orchestration Console*[3] is used by developers to interactively unit test and debug BPEL Scenarios.

In our loan procurement example, when a customer submits a loan application through the portal, a JSP is invoked which uses the business delegate interface of the LoanProcurement BPEL Scenario to initiate a new instance.

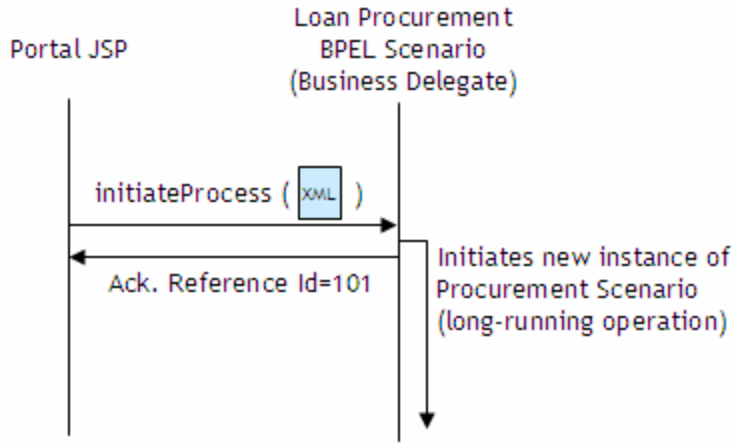


Figure 10 – Initiating a BPEL Scenario instance from a Portal JSP

Because the loan procurement scenario performs asynchronously, the business delegate returns a handle to the initiated transaction to the calling JSP: "we are working on your request, here is the reference id". This handle can be used to obtain information on the transaction in progress, for example to check its completion status or cancel it. For more details, regarding the source code of the JSP that initiates the scenario, please refer to the Collaxa WSOS developer guide.

ASYNCHRONOUS WEB SERVICE INTERACTIONS

XML-to-Java and Java-to-XML Mapping

XML documents play a very important role in orchestration. They are used both to store the context of the long-lived multi-step business transaction as well as to exchange information among the various orchestrated services.

[2] Please refer to the "Core J2EE Patterns: Best Practices and Design Strategies" ISBN-0130648841 for more information on the Business Delegate design pattern

[3] Please refer to the WSOS documentation for more information on the orchestration console.

In the loan procurement scenario, for example, the submitted loan application and collected loan offers are XML documents defined using XML Schema. Marshalling, manipulating and persisting XML documents is not a trivial task, but is handled automatically by the orchestration server.

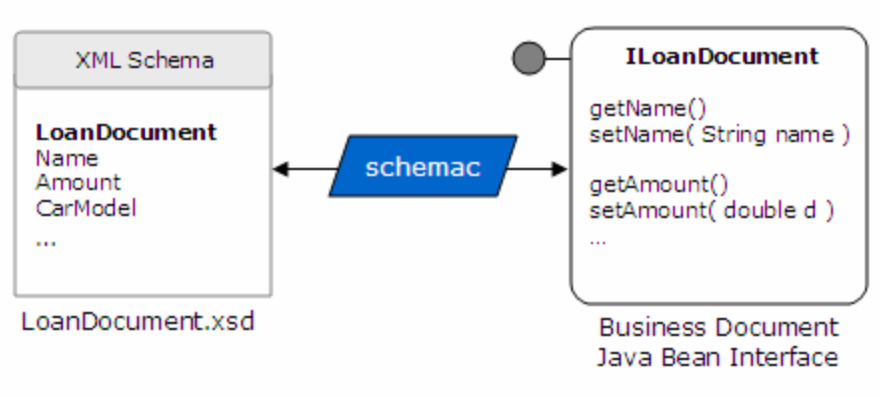


Figure 11 - schemac: XML to Java Mapping Utility

The Collaxa Web Service Orchestration Server provides a facility that wraps XML documents into Business Documents, called *schemac*. A Business Document is a Java object whose interface reflects the structure defined by an XML Schema. The Orchestration Server provides container-managed marshalling and persistence for business documents, shielding developers from low-level DOM manipulation and complex persistence tasks. The Collaxa Business Document facility supports simple types, complex types, nested complex types, inherited types and arrays.

Interacting with Asynchronous Web Services

BPEL Scenarios dramatically simplify the invocation and integration of both synchronous and asynchronous web services.

In the loan procurement application, the operations performed on the credit rating and the 2 loan processors are asynchronous. For example, it could take several days between the time a loan processor receives a loan application and the time it generates and returns an offer.

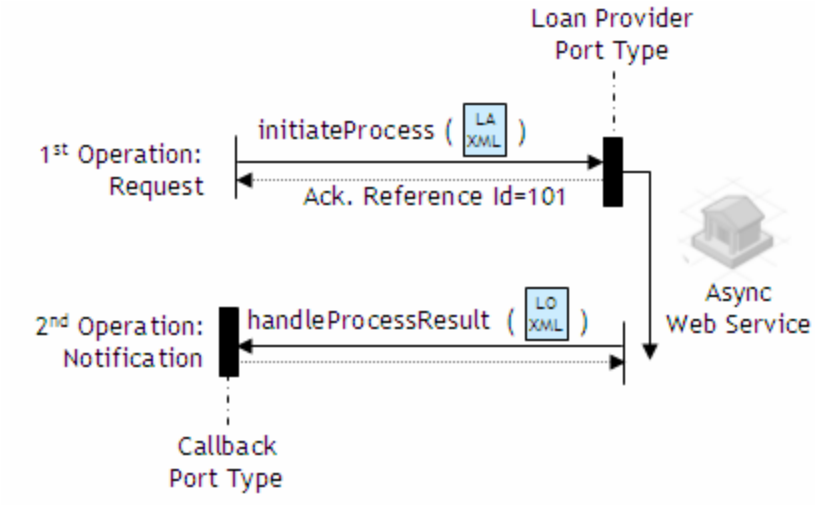


Figure 12 – Interacting with an Asynchronous Web Service

Handling asynchrony, and its associated callback mechanism, using traditional coding techniques is complex: developers need to define a database schema to persist the state and context of each transaction and conversation, set up listeners, map each response back to the transaction and determine the next set of actions to be performed based on the response and context. This can be very complex to manage, implement and monitor as the number of end points and conversations increase.

BPEL Scenarios and the delivery service module of the Collaxa Orchestration Server have been designed to manage and hide the complexity of asynchronous interactions. `deliveryService.receiveMessage(…)` allows developers to specify within the scenario code that they are waiting for an asynchronous callback/notification[4] from a remote web service . The orchestration server detects the dependency on an asynchronous callback, pauses the execution scenario, bookmarks its state and passivates it in a database until the callback/notification is actually received. When the notification is received, the scenario is reactivated and execution resumes from the bookmarked state.

The first step in invoking a web service as part of a BPEL scenario is to use the wsdlc command line tool to compile the WSDL description of that service into a Web Service connector. The connector is a Java class representation of the remote service and hides the complexity related to SOAP marshalling.

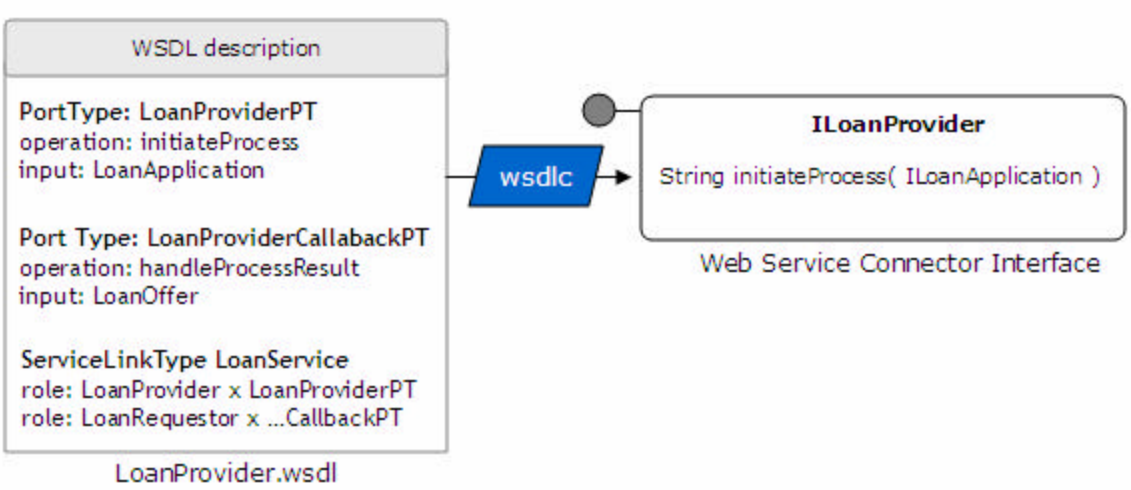


Figure 13 - Generating a Web Service Connector[5]

Note 1: No Java class or interface is generated for the callback interface. The callback interface is automatically implemented by the delivery service and is transparent to the developer. The messages received through the callback interface are accessible through `deliveryService.receiveMessage(…)` call.

Note 2: When invoking the one-way `initiateProcess` operation, the orchestration server initiates a new conversation, passes the conversation id to the invoked service as part of the SOAP header[6] and returns it to the scenario. The conversation id is used to correlate callback messages.

The generated ILoanProvider connector can then be referenced and invoked as part of the BPEL Scenario implementation.

[4] It might take arbitrarily long to get the actual notification - anywhere from seconds to days.
[5] Please refer to the BPEL4WS spec for information related to ServiceLinks
[6] See the ws-conversation spec for information about the structure of the SOAP header.

Shown below is a source code fragment illustrating how a loan procurement scenario could invoke asynchronous loan provider services with a BPEL Scenario:

```
public class LoanProcurement extends Scenario
{
  /** @ws-conversation:mode async */
  public IPolicy process( ILoanApplication loanApp )
  {
     // ###################################################
     // Step #1: Initiate Asynchronous Loan Processor
     // ###################################################

     // 1.1. Create a reference to the unitedLoan Web Service provider
     // note: web service locations are defined in the Scenario
     //       deployment descriptor so that they can be easily updated.
     //       Please refer to Developer Guide for more information.
     ILoanProvider unitedLoan =
          (ILoanProvider) lookup( "web-service:/unitedLoan" );

     // 1.2. Ask UnitedLoan to initiate processing the loan application.
     // note: convId is the conversation id of this asynchronous
     //       operation. The conversation id is used for waiting for
     //       the notification or polling for the result.
     String convId = unitedLoan.initiateProcess( loanApp );

     // ###################################################
     // Step #2: Receive Notification from Loan Processor
     // ###################################################

     // 2.1 Create a reference to the delivery service that will receive
     // the callback notification from the asynchronous loan processor.
     IDeliveryService deliveryService =
        (IDeliveryService) lookup( "delivery-service" );

     // 2.2. Wait until the delivery services receives the asynchronous
     // callback from the loan provider. It might take anywhere between
     // a few minutes to a few hours for the loan provider to process
     // the application and perform the callback.
     // note: The execution of the scenario will pause until the
     //       callback notification is received.
     ILoanOffer offer =
             (ILoanOffer) deliveryService.receiveMessage( convId );

     // ###################################################
     // Step #3: Add XML representation of loan offer to the
     //          audit trail of  this business transaction.
     // ###################################################
     addAuditTrailEntry(  "Offer Received" , "UnitedLoan's offer" ,
                          offer.toString()
                          );
     // Rest of the logic need to enroll customer and generate policy
     …
     return policy;
  }
}
```

What happens behind the scenes?

Developers have several common questions when they are first exposed to the BPEL Scenario abstraction: How is the call `(ILoanOffer) deliveryService.receiveMessage( convId )` implemented? Is there a physical thread waiting for the offer to get generated? What happens if it takes 5 days before the response comes back? What happens if the server crashes while waiting of the callback notification?

These are all important questions but the second, regarding the existence of a physical thread, really gets to the heart of the matter. And the answer is: no, there are no physical threads blocking and waiting for asynchronous callbacks. When the BPEL Scenario invokes the delivery service and asks for the receipt of a message - `deliveryService.receiveMessage( convId )`, the orchestration server detects that the operation is a callback notification and if the message has not already been delivered, the state of the Scenario is bookmarked and it is passivated in a database. When the delivery service receives the callback notification from the loan processor, the Scenario is reactivated and its execution is resumed. This behavior means that an arbitrary amount of time can pass before a response to an asynchronous request is received and the system is fully fault-tolerant.

FLOW COORDINATION

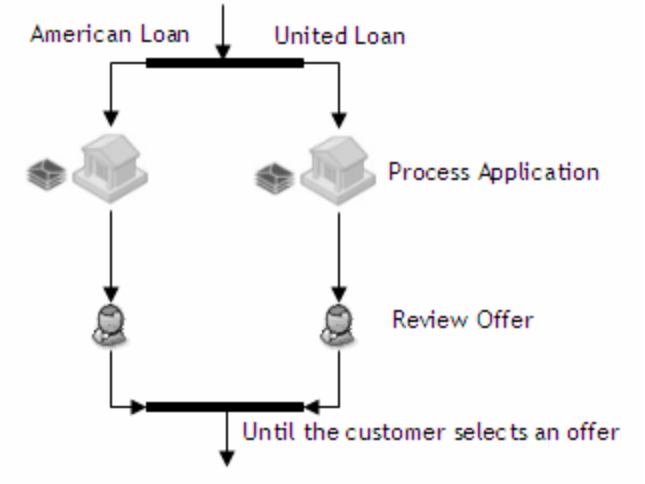


Figure 14 - Parallelism and join patterns

In the loan procurement scenario, we need to invoke 2 loan processors which are implemented as asynchronous web services. Each might take up to 5 days to process the application and generate an offer. In order to streamline the overall process, we need to be able to invoke those services in parallel.

The `/** @bpel:flow */`…`/** @bpel:sequence*/`…`/** @bpel:join */` orchestration tags allow the developer to define parallel execution branches and sophisticated join patterns.

Shown below is a source code fragment from the loan procurement scenario which illustrates the use of the @bpel:flow orchestration tag.

Source Code Fragment

```
public class LoanProcurement extends Scenario
{
  /** @ws-conversation:mode async */
  public ILoanPolicy process( ILoanApplication loanApp )
  {
     // …
     boolean offerSelected = false;
     ILoanOffer offers = new ILoanOffer[ 2 ];

     // Ask UnitedLoan and American Loan to process the loan
     // application in parallel.
     /** @bpel:flow */
     {
          /** @bpel:sequence United */
          {
               // Initialize unitedLoan
               …
               // Ask UnitedLoan to process application.
               String convId = unitedLoan.initiateProcess( loanApp );

               // Wait until it delivery service receives the callback
               // notification.
               offers[0] =
                   (ILoanOffer) deliveryService.receiveMessage( convId );

               // Ask customer to review offer
               offerSelected = reviewOffer( offers [ 0 ] );
          }

          /** @bpel:sequence American */
          {
               // Initialize American Loan
               …
               // Ask American Loan to process application.
               String convId = americanLoan.initiateProcess( loanApp );

               // Could take up to 5 days to return (async).
               offers[1] =
                   (ILoanOffer) deliveryService.receiveMessage( convId );

               // Ask customer to review offer
               offerSelected = reviewOffer( offers [ 1 ] );
          }
     }
     /** @bpel:join offerSelected == true */
     {
          // logic that is executed when the first offer is
          // selected.
     }
     //…
  }
}
```

At run-time, when the `/** @bpel:flow */` tag is reached, the orchestration server executes both branches in parallel. As soon as the `/** @bpel:join */` condition is

reached, remaining active branches are cancelled and the orchestration server resumes execution at the beginning of the `/** @bpel:join */` block.

Note: There are cases when developers do not know the number of branches until run-time. For example, we might want to create a scenario where loan application are submitted to all loan processors defined in a database. In that case, at design/implementation time, developers do not know the number of loan processors that need to be invoked. The `/** @bpel:flowN */` orchestration tag is defined to allow developers to handle that use case through the dynamic branching capabilities of the orchestration server.

USER INTERACTIONS/PORTAL INTEGRATION

Interactions with end users are an important part of the activities orchestrated by BPEL Scenarios, for example: manual approvals, knowledge contribution and exception management.

Shown below is the source code from the loan procurement scenario for having the customer [manually] review a loan offer which has been received.

Source Code Fragment

```
public class LoanProcurement extends Scenario
{
  /** @ws-conversation:mode async */
  private boolean reviewOffer( ILoanOffer offer )
  {
     // ########################################
     // Step #1: Initialize Task
     // ########################################
     ITask reviewTask = TaskFactory.create();
     reviewTask.addPerformer( mCustomerEmail );
     reviewTask.setProperty( “loanOffer”, offer );

     // ########################################
     // Step #2: Wait for task to complete
     // ########################################
     waitForTask( reviewTask );

     // The next statement is only executed when the task is completed
     // by the customer. The offer business document contains the flag
     // that defines if the customer has selected the offer.
     return offer.isSelected();
}
```

In the loan procurement example, the scenario needs to engage the customer to have him or her approve and select the generated offers. BPEL Scenarios have built-in support for user interactions through the notion of tasks and a built-in worklist service. Tasks create breakpoints in the scenario for users to participate. Business documents and other Java objects can be associated with the task as properties to facilitate data interchange between the BPEL Scenario and the external application that manages the interaction with the user. That application can be a portal, a custom JSP or a form embedded in an email – in fact, any user interface is supported.

Interactions with end users are asynchronous by nature: while waiting for a task to complete, the orchestration server will pause the execution of the scenario and passivate that instance of the scenario (in the database). Upon completion of the task, its properties will be updated and the execution of the scenario will resume. This functionality makes people and manual tasks just another asynchronous service to the BPEL Scenario developer.

Worklist Service
Java Server Pages or Portal applications can use the Task API of the orchestration server to query the list of tasks, load properties, update them and complete the task. Please refer to the Developer Guide for more information.

Escalations
Expiration dates and escalation patterns can also be implemented using BPEL Scenario events. Please refer to the Developer Guide for more information.

Roles
LDAP directory support is available for the BPEL Scenario to provide flexible and sophisticated user authentication and role resolution support. Please refer to the CXDN Knowledge Base for examples.

EXCEPTIONS AND TIMEOUTS

Exceptions and timeouts are an integral part of orchestration logic and need to be managed appropriately. In BPEL Scenarios, exceptions are handled using the Java `try…catch` mechanism while timeouts are handled with expiration events (see the Developer Guide for more information regarding orchestration events).

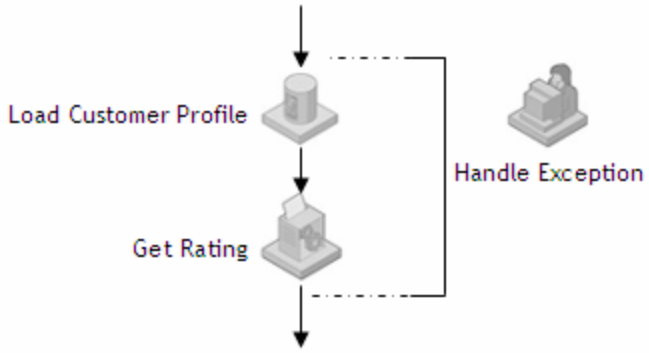


Figure 15 - Handling Exceptions

As part of the loan procurement application, we need to implement the application's orchestration logic so that if an exception is thrown while the customer profile is being looked up or the credit rating is being generated, a customer support rep is notified and he or she is assigned a task to manually complete the application. This behavior is implemented with the Java `try…catch` exception management and BPEL Scenario's support for user interactions (aka tasks), as shown in the source code below.

Source Code Fragment

```
public class LoanProcurement extends Scenario
{
  /** @ws-conversation:mode async */
  public IPolicy process( ILoanApplication loanApp )
  {
     // …
     try
     {
          // Load Customer Profile – Can throw exceptions
          loadCustomerProfile( loanApp );

          // Assign Rating – Can throw exceptions
          assignRating( loanApp );
     }
     catch( InvalidCustomerException ice )
     {
          // Business logic responsible for handling exceptions.
          // That logic can change the data and re-invoke the partner
          // or create a task and ask a user to manually resolve the
          // exception.
          …
     }

     //…
  }
}
```

BUSINESS TRANSACTIONS

In some use cases, exception handling is not enough. Let's imagine that you are implementing a TripPlanner Scenario in which you are booking a flight reservation and a hotel reservation. You need to make sure that both the operations are either confirmed or cancelled together. In addition, cancellation cannot be implemented with a rollback mechanism because the services may be provided by an external system or company. Transaction management standards such as WS-Coordination, WS-Transaction and BTP are emerging to address these requirements.

The BPEL Scenarios let developers assign transactional semantics to each operation:
`/** @ws-transaction:requires-new */`

The orchestration server will initiate a new business transaction when the scenario is initiated and the bookTrip method invoked. The transaction context will be automatically propagated to all web services invoked as part of the scenario, using WS-Coordination and WS-Transaction. When the scenario completes, all transaction are completed. If an exception is thrown and not handled by the scenario, the orchestration server will automatically cancel/compensate each participant.

Shown below is a code example implementing a TripPlanner scenario.

```
public class TripPlanner extends Scenario
{
  /**
   *  @ws-conversation:mode async
   *  @ws-transaction:attribute requires-new
   */
  public ITripReceipt bookTrip( ITripInfo info )
  {
    // Declare and initialize some variables: mAvis, mUnited, mDelivery
    …
    /** @bpel:flow */
    {
       /** @bpel:sequence BookFlight*/
       {
          // Ask United to book the flight based on the
          // specified information
          String unitedConv = mUnited.bookFlight( info );

          // Wait until united callsback with the ticket
          // confirmation.
          eTicket =
              (ITicket) mDelivery.receiveMessage( unitedConv );

       }
       /** @bpel:sequence BookCar*/
       {
          // Ask Avis to reserve a Car
          String avisConv = mUnited.bookCar( info );

          // Wait until avis callsback with the car reservation
          // information.
          carReza =
              (ICarReservation) mDelivery.receiveMessage( avisConv );
       }
    }
    // Generate and return receipt
    ITripReceipt receipt = TripReceiptFactory.create();
    receipt.setETicket( eTicket );
    receipt.setCarReservation( carReza );
    return receipt;
  }
}
```

@ws-transaction:attribute	
required	Leverage the caller's transaction context if specified or create a new transaction context if caller is not part of a transaction.
requires-new	Create independent transaction context.
mandatory	Throw an exception if caller is not providing a transaction context.
none	Do not create a transaction context when interacting with web services.

For more information and examples, please refer to the Business Transaction chapter of the Collaxa Developer Guide.

Next Steps: Kick The Tires

The Collaxa Developer Network (CXDN) offers everything you need to start composing, deploying and testing your first BPEL Scenario, including:

- A complete Developer Guide and Tutorial
- Full executable source code for the examples shown in this paper
- Free evaluation version of the Collaxa WSOS
- Sample BPEL Scenarios
- Free online support
- Training and webinar events

→ http://www.collaxa.com/developer.welcome.html

For other information regarding the Collaxa Web Service Orchestration Server or to get evaluation support for a specific project, please email us at sdave@collaxa.com.

Thank you for evaluating Collaxa.