Implementing Service-Oriented Architectures (SOA) with the Java EE 5 SDK

By Gopalan Suresh Raj, Binod PG, Keith Babo, and Rick Palkovic

Sun Microsystems, Inc.
www.sun.com

May 2006, Revision 02

Contents

Service-Oriented Architectures and the Java EE 5 SDK

Service-oriented architectures (SOA) promise to implement composite applications that offer location transparency and segregation of business logic. Location transparency allows consumers and providers of services to exchange messages without reference to one another's concrete location. Segregation of business logic isolates the core processes of the application from other service providers and consumers.

Together, these features let you replace or upgrade individual components in the composite application without affecting other components or the process as a whole. Moreover, you can independently specify alternative paths through which the various parts of the composite application exchange messages.

This article presents concepts and language constructs necessary to developing a SOA composite application in Java EE 5 and describes an example application based on a loan application use case. Examples from the sample application are used throughout the article.

In the loan application use case, a user applies online for a loan by filling out a loan request with necessary financial and personal information. The application's business logic verifies the user information and accepts or rejects the loan request. The use case is summarized in FIGURE 1.

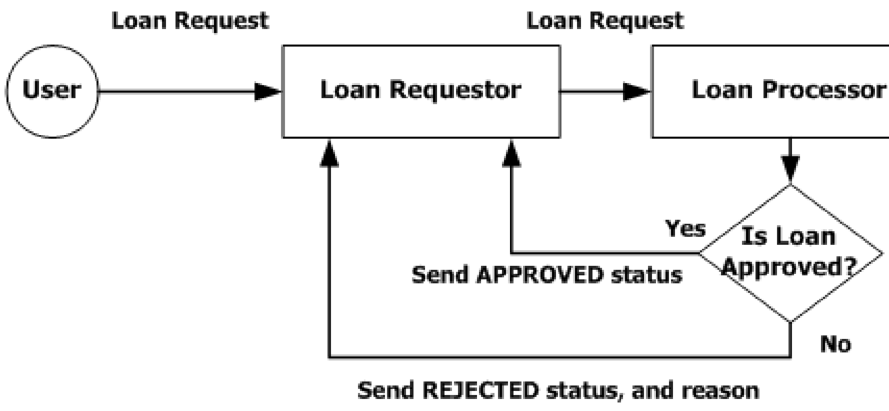


FIGURE 1 Business Use Case Flow Chart for Example Composite Application

To download the files related to this NetBeans™ 5.5 project, refer to the section titled "Resources" on page 62.

The project uses the following components:

- The HTTP/SOAP binding component
- The Business Process Execution Language (BPEL) Service Engine
- The Java EE Service Engine (from the GlassFish project)

Java EE Platform and Web Services

Beginning with J2EE 1.4, the Java EE platform has fully supported both clients of web services and web service endpoints. As a result, a Java EE component can be deployed as a web service at the same time it acts as a client of a web service deployed elsewhere.

JAX-WS (Java API for XML Web Services) is the primary API for web services in Java EE 5. It supports web service calls by using the SOAP/HTTP protocol as constrained by the WS-I Basic Profile specification.

Java EE web services are defined by the specification Implementing Enterprise Web Services (JSR 109). The specification describes the deployment of web service clients and web service endpoints in Java EE releases as well as the implementation of web service endpoints using Enterprise JavaBeans (EJB) components. Java EE web services, along with JBI, provide a strong foundation for implementing a SOA.

JAX-WS 2.0

JJAX-WS 2.0 replaces an older API, JAX-RPC 1.1 (Java API for XML-based Remote Procedure Call), extending it in many areas. The new specification supports multiple protocols, such as Simple Object Access Protocol (SOAP) 1.1, SOAP 1.2, and XML. JAX-WS uses JAXB 2.0 as its data binding model, relying on usage annotations to considerably simplify web service development. It also uses many annotations defined by the specification Web Services Metadata for the Java Platform and introduces steps to plug in multiple protocols instead of HTTP only. In a related development, JAX-WS defines its own message-based session management.

Java EE Web Service Architecture

Web service architecture, in general, allows a service to be defined abstractly, implemented, published and discovered, and used interoperably. You can decouple a web service implementation from its use by a client in a variety of ways-in programming model, logic, and transport. As a consequence, a web service that has been developed with the .NET platform can be used by a Java EE application, and vice versa.

In simplest terms, a service instance, called a Port component, is created and managed by a Java EE container, which in turn can be accessed by the client application. The Port component can also be referenced in client, web, and EJB containers.

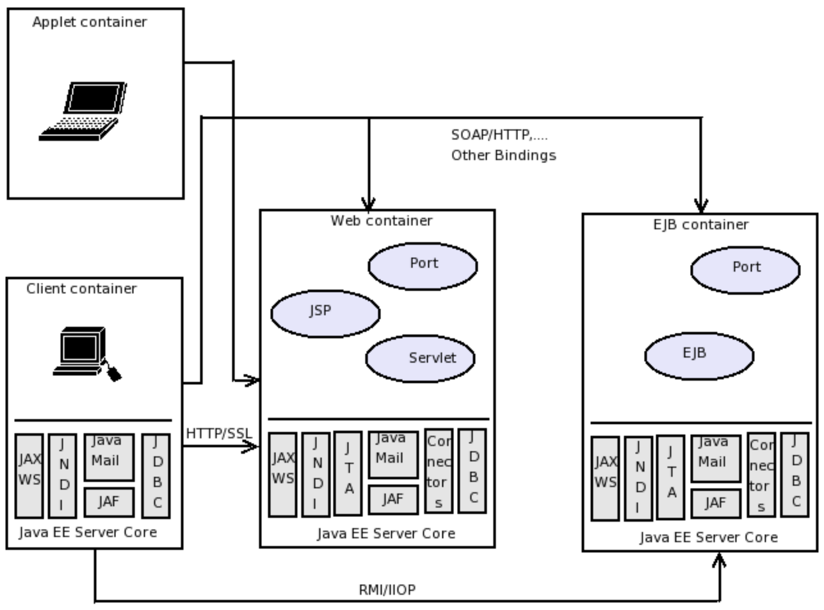


FIGURE 2 Java EE Web Service Architecture

The life cycle of a Port component's implementation is specific to and completely controlled by its container-the two are intimately linked.

The Port component associates a Web Service Definition Language (WSDL) port address with an EJB service implementation bean-a Java class that provides the business logic of the web service and that always runs in an EJB container. Because the service implementation is specific to a container, the service implementation also ties a Port component to its container's behavior. The methods that the service implementation bean implements are defined by the Service Endpoint Interface.

A container provides a listener for the WSDL port address and a means of dispatching the request to the service implementation bean. For example, if you deploy an EJB service implementation bean by using basic JAX-WS SOAP/HTTP transport, the bean will run in a Java EE EJB container and an HTTP listener will be available in the container for receiving the request.

EJB Service Implementation Bean Example

The following code shows an example EJB service implementation bean.

```
package com.sun.jbi.blueprints.loanprocessor;

import javax.ejb.Stateless;
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebParam;

@Stateless()
@WebService()

public class LoanProcessorEJBBean {

   /**
    * Process Loan Application Web service operation
    */

    @WebMethod
    public String processApplication(
                          @WebParam String socialSecurityNumber,
                          @WebParam String applicantName,
                          @WebParam String applicantAddress,
                              @WebParam String applicantEmailAddress,
                          @WebParam int applicantAge,
                          @WebParam String applicantGender,
                          @WebParam double annualSalary,
                          @WebParam double amountRequested) {

        int MINIMUM_AGE_LIMIT = 18;
        int MAXIMUM_AGE_LIMIT = 65;
        double MINIMUM_SALARY = 20000;
        int AVERAGE_LIFE_EXPECTANCY = 70;

        String result = "Loan Application APPROVED.";

        if(applicantAge < MINIMUM_AGE_LIMIT) {
            result =
              "Loan Application REJECTED - Reason: Under-aged "+
              applicantAge+". Age needs to be over "+
              MINIMUM_AGE_LIMIT+" years to qualify.";
            return result;
        }
        if(applicantAge > MAXIMUM_AGE_LIMIT) {
            result = "Loan Application REJECTED - Reason: Over-aged "+
              applicantAge+". Age needs to be under "+
              MAXIMUM_AGE_LIMIT+" years to qualify.";
            return result;
        }

        if(annualSalary < MINIMUM_SALARY) {
         result = "Loan Application REJECTED - Reason: Annual Salary $"+
             annualSalary+" too low. Annual Salary needs to be over $"+
              MINIMUM_SALARY+" to qualify.";
         return result;
        }

        int yearsToRepay = AVERAGE_LIFE_EXPECTANCY-applicantAge;
        double limit = annualSalary*yearsToRepay*0.5;
        if(amountRequested > limit) {
          result =
              "Loan Application REJECTED - Reason:
               You are asking for too much $"+
            amountRequested+". Annual Salary $"+annualSalary+
              ", Age "+applicantAge+" years. Your limit is $"+limit;
          return result;
        }

        return result;
    }
}
```

Note the `@Stateless` and `@WebService` annotations: `@Stateless` annotates the bean as stateless, and `@WebService` annotates the bean as a web service.

When the EJB service implementation bean is packaged and deployed, the application server's deployment tool generates a WSDL according to the rules defined in the JAX-WS specification. It also brings up a listener so that clients can access the service.

Client View of the Java EE Web Service

A Java EE application client, web component, EJB component, or another web service can act as a client of a web service. The client accesses a web service through a Service Endpoint Interface as defined by the JAX-RPC or JAX-WS specification.

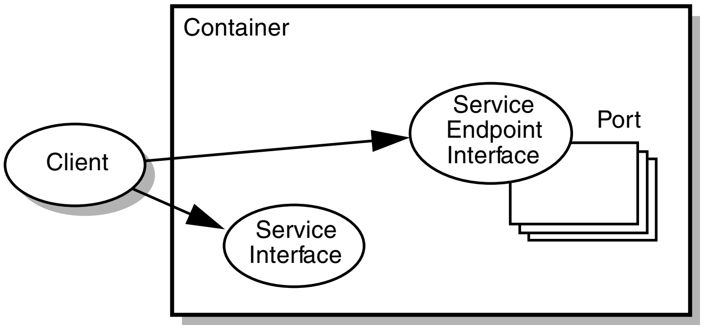


FIGURE 3 Web Services Client View

Because the port can be referenced in the client, the client application can use the `@WebServiceRef` annotation to access a web service, as shown in the following code example.

```
package com.sun.jbi.blueprints.loanrequestor;

import javax.xml.ws.WebServiceClient;
import javax.xml.ws.WebServiceRef;

import com.sun.jbi.blueprints.loanprocessor.LoanProcessor;
import com.sun.jbi.blueprints.loanprocessor.LoanProcessorService;
/**
 * JAXWSClient is a stand-alone Java program
 * that accesses the processApplication
 * method of LoanProcessor. It makes this call
 * It makes this call through a port, a local object
 * that acts as a proxy for the remote service.
 * The port is created at development time
 * by the wsimport tool, which generates JAX-WS portable
 * artifacts based on the WSDL file.
 */
public class JAXWSClient {
    // WebServiceRef using the generated service interface type
    @WebServiceRef
    static LoanProcessorService PROCESSOR_SERVICE =
                      new LoanProcessorService();
    private LoanProcessor loanProcessor;
    public String test(String ssn, String name, String email,
                       String address, int age, String sex,
                       double salary, double loanAmount) {
        this.loanProcessor = PROCESSOR_SERVICE.getLoanProcessorPort();
        // make the actual call
        return this.loanProcessor.processApplication(ssn,
                       name, email, address, age, sex,
                       salary, loanAmount);
    }

    public static void main(String[] args) {
        JAXWSClient client = new JAXWSClient();
        String result = client.test("123-45-6789",
                             "Gopalan Suresh Raj",
                             "gopalan.raj@sun.com",
                                 "800, Royal Oaks Blvd, Monrovia, CA",
                              36, "Male", 9876543, 1234567);
        System.out.println(result);
    }
}
```

The Service object is a factory used by the client to get a stub or proxy that implements the Service Endpoint Interface. The stub is the client representation of an instance of the web service.

Java Business Integration (JBI) – JSR 208

The JBI 1.0 (JSR 208) specification is an industry-wide initiative to create a standardized integration platform for Java and business applications. JBI addresses service-oriented architecture (SOA) needs in integration by creating a standard meta-container for integrated services.

JBI is a messaging-based plug-in architecture. This infrastructure allows third-party components to be "plugged in" to a standard infrastructure and enables those components to interoperate seamlessly. It does not define the pluggable components themselves, but defines the framework, container interfaces, behavior, and common services. The meta- container is itself a service-oriented architecture. JBI components describe their capabilities through the Web Service Definition Language (WSDL).

The major goal of JBI is to provide an architecture and an enabling framework that facilitates dynamic composition and deployment of loosely coupled composite applications and service-oriented integration components. It allows anyone to create JBI- compliant integration plug-in components and integrate them dynamically into the JBI infrastructure.

JSR 208 specifies two deployable archive packages (`.jar` files) called Service Unit and Service Assembly. These archives contain code and standard descriptors, in a manner similar to WAR and EAR packaging in Java EE 5.

FIGURE 1 shows the key pieces of the JBI environment:

- **Service Engines** – Service Engines (SE) are JBI components that enable pluggable business logic.
- **Binding Components** – Binding Components (BC) are JBI components that enable pluggable external connectivity.
- **Normalized Message Router** – The normalized message router (NMR) directs normalized messages from source components to destinations according to specified policies.

- **JBI Runtime Environment** – The JBI runtime environment encompasses the JBI components and the NMR. Because of its container characteristics, it is sometimes called the JBI meta-container.

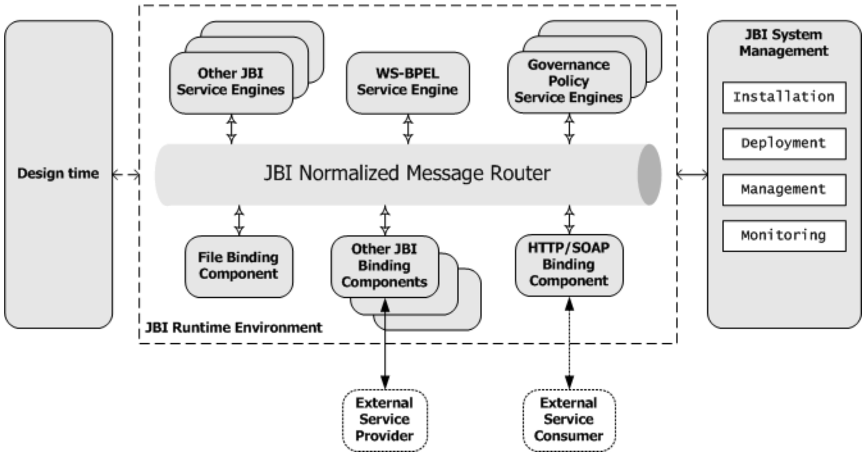

Design time
Other JBI Service Engines
WS-BPEL Service Engine
Governance Policy Service Engines
JBI Normalized Message Router
File Binding Component
Other JBI Binding Components
HTTP/SOAP Binding Component
JBI Runtime Environment
JBI System Management
Installation
Deployment
Management
Monitoring
External Service Provider
External Service Consumer

FIGURE 1 The JBI Environment

BCs and SEs can act as containers, and Service Units can be deployed to installed JBI components. This arrangement allows the deployment (and undeployment) of component- specific artifacts (for example, concrete WSDLs). Service Units can describe what services are provided and consumed by the JBI component. Besides the standard descriptor, the JBI component is responsible for interpreting the contents of the Service Unit `.jar` file.

Multiple Service Units can be packaged in a Service Assembly. The Service Assembly defines the target JBI components to which to deploy the Service Units.

The JBI Meta-Container

The JBI meta-container hosts multiple JBI components (SEs and BCs). When sending and receiving messages outside the JBI environment, the SEs communicate using the NMR and pass messages out to the client through a binding component. When communication is entirely within the JBI environment, no protocol conversion, message serialization, or message normalization is necessary because all messages are already normalized and are in standard WSDL 2.0 format.

Service Engines

The JBI architecture as shown in FIGURE 2 consists of a JBI meta-container, or runtime environment, that can run SEs, which in turn host Service Units (also called service deployments). SEs provide services such as business logic, processing, transformation, and routing services.

For example, you can use a WS-BPEL Service Engine to orchestrate business processes that use WS-BPEL. Similarly, you can preserve your current investment in existing Java EE Application Servers by wrapping your existing EJB or servlet code as a Web service, then reusing it in a SOA with a Java EE Service Engine.

Binding Components

In the JBI environment, protocol independence is provided through BCs. BCs provide transport or communication protocols as well as access to remote services from within the JBI environment. They also provide access to other services in the JBI framework. Protocol-binding components provide a proxy for services in the JBI environment to access service providers that require a particular protocol.

BCs are custom built for each external protocol and are plugged in to the JBI meta-container. This architecture allows any JBI component to communicate over any protocol or transport (SOAP, JMS, and so on) as long as a BC that handles the particular transport or protocol is plugged in to the JBI meta-container.

BCs provide interoperability between services by using protocols such as SOAP, Simple Mail Transfer Protocol (SMTP), Java Message Service (JMS), and so on. Using BCs, you do not need to implement those protocols in your business logic. BCs enable loose coupling, a hallmark of SOA, by decoupling the service implementation from the access mechanism.

The BC consumes protocol-specific message data, converts it into a JBI-specified "normalized message," and hands it off to the NMR for consumption by any SE. Similarly, the BC picks up the normalized message received from the NMR, "denormalizes" the message into a protocol-specific message, and sends it back to the consuming client.

Normalized Message Router

The NMR provides the abstract mediated message exchange infrastructure that is the key to interoperation between JBI components (both SEs and BCs). The NMR receives message exchanges from all SEs and BCs and routes them to the appropriate JBI component for processing. Message exchange between the NMR and the JBI components are done with JBI normalized messages, which provide the interoperability between components, as illustrated in FIGURE 2.

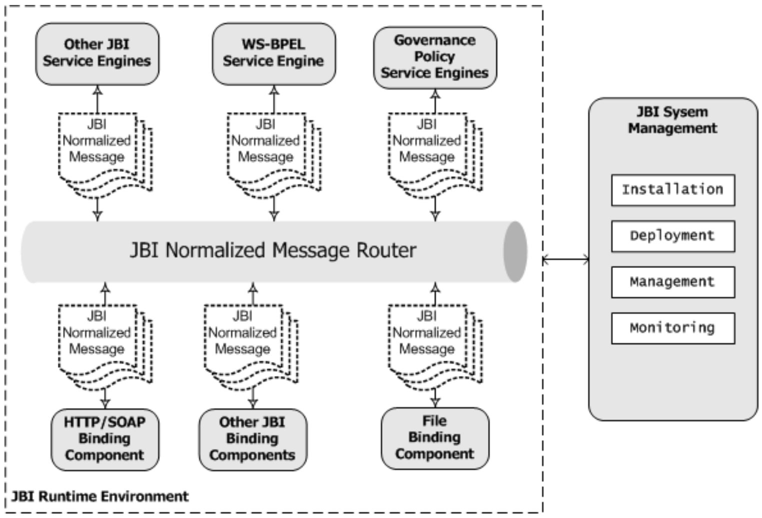


FIGURE 2 Messaging Between JBI Components

The messages passing through the NMR are JBI normalized messages. The NMR focuses on sending and receiving messages, and providing a means of transferring and propagating the transactional context and security context between the BCs and the SEs.

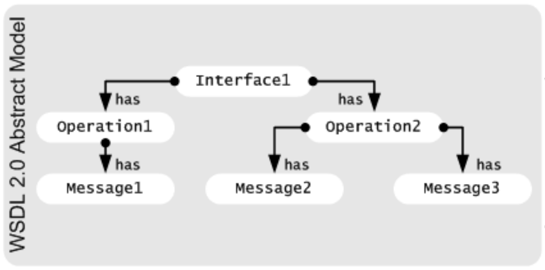
JBI Normalized Message

A JBI normalized message is an XML document. A typical JBI normalized message consists of two parts:

1. Message metadata – Also known as the message context data, the message metadata includes context information such as:

- Protocol-supplied context information
- Security tokens
- Transaction context information
- Data specific to other components

2. Message payload – The message payload is a generic source abstraction that contains all the message data. The payload conforms to an abstract WSDL message type (see FIGURE 3), with no protocol encoding or formatting.



```
<interface name="IRequestLoan">
  <operation name="requestLoan">
    <input name="input"
        message="tns:requestLoanMessage">
    </input>
    <output name="output"
        message="tns:requestLoanResponseMessage">
    </output>
  </operation>
</interface >
```

FIGURE 3 WSDL 2.0 Abstract Message Model

Delivery Channel

The delivery channel is a bidirectional communication pipe used by all JBI components (SEs and BCs) to communicate with the NMR. A service consumer uses the delivery channel to initiate service invocations, as shown in FIGURE 4.

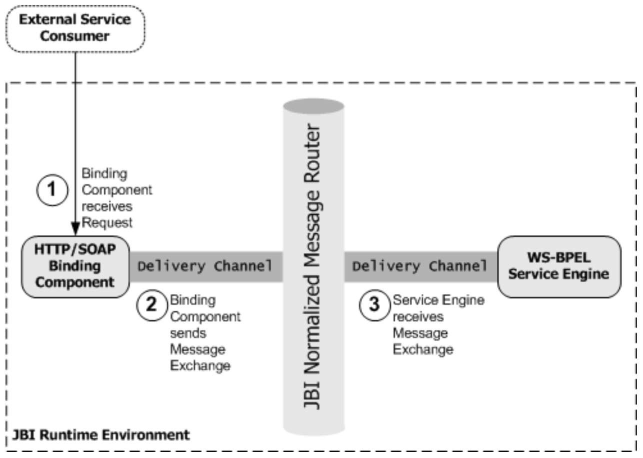

External Service Consumer
1 Binding Component receives Request
HTTP/SOAP Binding Component
Delivery Channel
2 Binding Component sends Message Exchange
JBI Normalized Message Router
Delivery Channel
3 Service Engine receives Message Exchange
WS-BPEL Service Engine
JBI Runtime Environment

FIGURE 4 External Service Consumer Initiates Service Request

A service provider uses its delivery channel to receive such invocations, as shown in FIGURE 5.

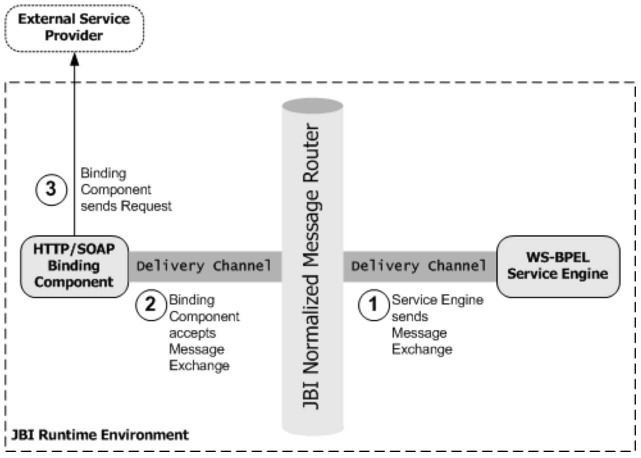

External Service Provider
Binding Component sends Request
3
HTTP/SOAP Binding Component
Delivery Channel
2
Binding Component accepts Message Exchange
JBI Normalized Message Router
Delivery Channel
1
Service Engine sends Message Exchange
WS-BPEL Service Engine
JBI Runtime Environment

FIGURE 5 External Service Provider Receives Request

Any JBI component that acts as both a service consumer and a service provider uses the same delivery channel for both roles. Each JBI component is provided with a single delivery channel.

JBI Message Exchange Patterns

JBI Message Exchange Patterns define an end-to-end interaction that deterministically completes a message exchange between a service provider and a service consumer The patterns define the sequence and cardinality of abstract message exchange.

JBI message exchange patterns are modeled after the WSDL 2.0 message exchange patterns. Message exchange patterns are defined from the service provider's perspective.

Service Invocation Patterns

Service invocation is an instance of end-to-end interaction between a service provider and a service consumer.

JBI mandates four message exchange patterns, as shown in the following table:

Service Invocation	Message Exchange Pattern (Provider View)
One-Way	In-Only
Reliable One-Way	Robust In-Only
Request Response	In-Out
Request Optional-Response	In Optional-Out

In-Only Message Exchange Pattern

The In-Only message exchange pattern is used for one-way exchanges.

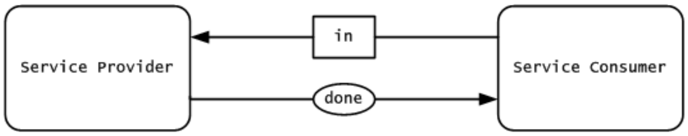


FIGURE 6 In-Only Message Exchange Pattern

1. Service consumer initiates a message exchange with an In-Only message.
2. Service provider responds with a status message to complete the message exchange.

Robust In-Only Message Exchange Pattern

The Robust In-Only message exchange pattern is used for reliable, one-way message exchanges.

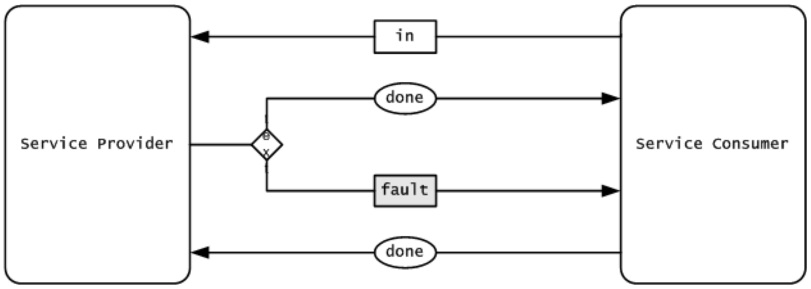


FIGURE 7 Robust In-Only Message Exchange Pattern

1. Service consumer initiates a message exchange with a Robust In-Only message.
2. Service provider may respond with status or fault message.
   - If provider response is status, the message exchange is complete.
   - If provider response is a fault, consumer responds with status to complete message exchange.

In-Out Message Exchange Pattern

The In-Out message exchange pattern is used for two-way exchanges.

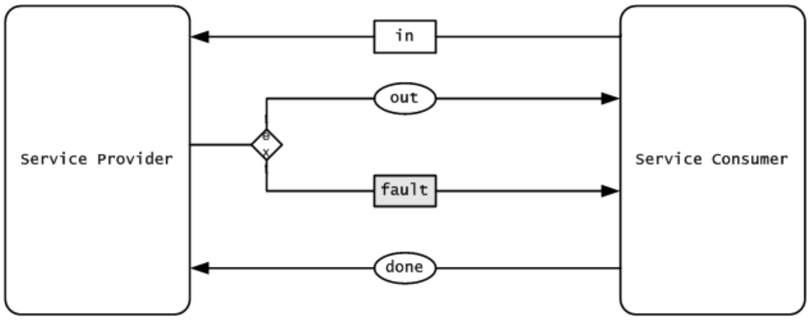


FIGURE 8 In-Out Message Exchange Pattern

1. Service consumer initiates a message exchange with an In-Out message.
2. Service provider responds with message or fault.
3. Service consumer responds with status to complete the message exchange.

In Optional-Out Message Exchange Pattern

The In Optional-Out message exchange pattern is used for a two-way exchange when the provider's response is optional.

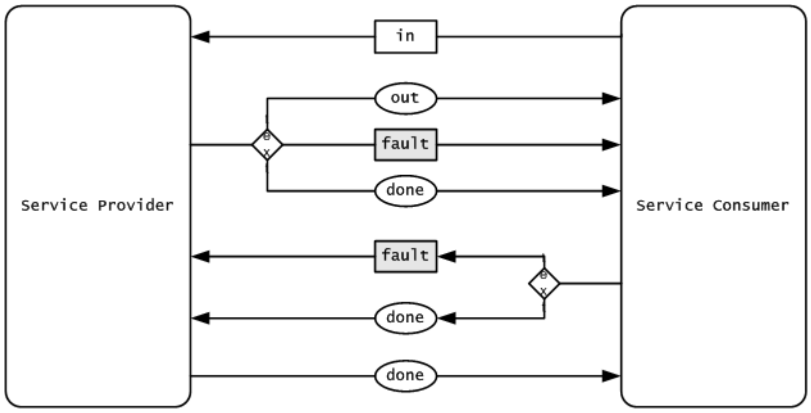


FIGURE 9 Optional-Out Message Exchange Pattern

1. Service consumer initiates a message exchange with an In Optional-Out message.
2. Service provider responds with message, fault, or status.
   - If provider responds with status, the exchange is complete.
   - If provider responds with fault, consumer responds with status. Status completes the message exchange.
   - If provider responds with message, consumer may respond with fault or status. Status completes the message exchange.
     - If consumer responds with fault, provider responds with status to complete the message exchange.

JBI Message Exchange Routing

Service Assemblies may contain deployment-provided routing information called service connections and service link types. Service connections provide explicit mappings from a consumer-provided address to the actual provider service endpoint. Service link types provide additional information about a component's expectations as a service consumer. These expectations concern the use of service connections to influence routing decisions made by the JBI Framework.

Each consumer declares, in its service unit metadata, its expectations concerning how service connections are to be applied to it when it consumes a particular service. This exchange of information is known as provider linking.

As an example, consider the HTTP/SOAP BC and the BPEL SE:

At Service Unit deployment time, the BC finds and reads the Service Unit `jbi.xml` file and locates the WSDLs with all the associated information for the external endpoint names provided. It activates all `<provides>` (outbound) endpoints. For the `<consumes>` endpoints, it starts an HTTP server on the port specified in the `<soap:address/>` tag.

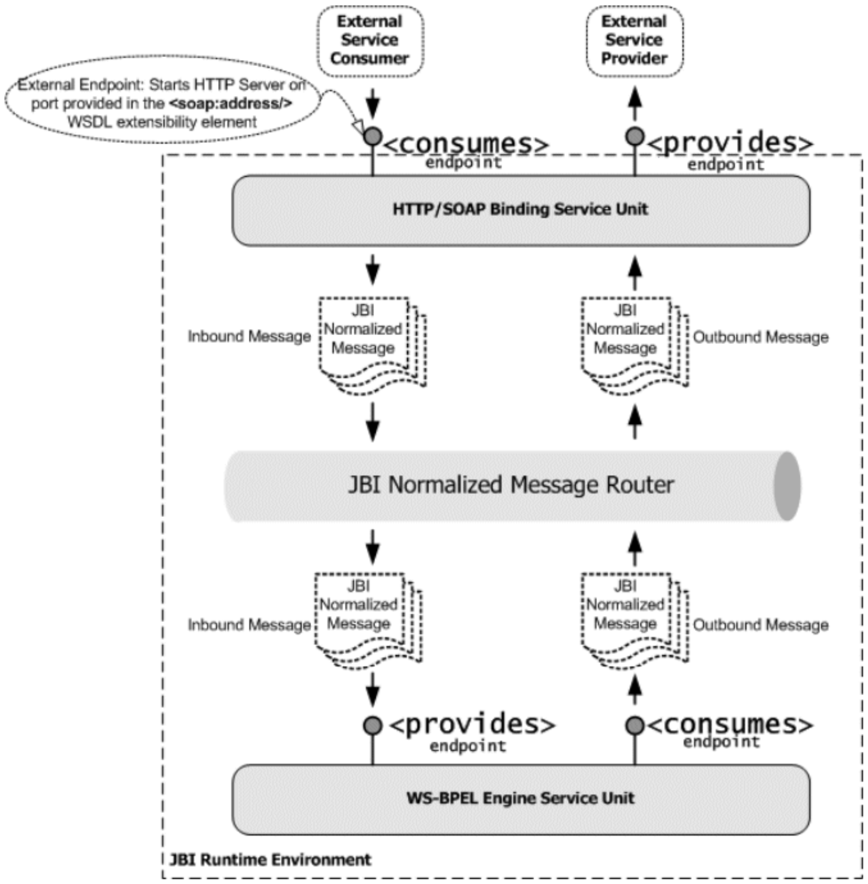


FIGURE 10 Routing Exchange Between the HTTP/SOAP Binding Component and the BPEL Service Engine

When the WS-BPEL Engine Service Unit is deployed to the BPEL SE, it reads the `jbi.xml` and activates the endpoints (named after the partner link and role) for the `<provides>` tag. It then finds all matching artifacts (BPEL constructs) associated with these partner links-role pairs.

For an inbound request, when a request arrives at the port that matches the context in a `<soap:address/>` tag, the BC sends the message to the NMR, giving the `<consumes>` service/endpoint details. In this case, the service/endpoint details are the same as the external endpoint details.

The framework determines from the descriptor that the link is a soft link and looks up the consumer address in the service connections that have been declared in the Service Assembly. It then maps the external endpoint name to the concrete endpoint the BPEL SE has activated. The endpoint is named after the partner link and role.

The BPEL SE receives the request and matches the partner link and role name of the endpoint on which it received the request to the matching BC to execute.

For an outbound request, essentially the same thing happens in the other direction. The SE sends the message to the NMR, using the `<consumes>` details (partner link and role). These details are then mapped to the concrete internal endpoint name (in this case, the same name as the external endpoint) of the BC through the service connection mapping in the Service Assembly descriptor.

Information in the Service Units and Service Assemblies Routing

Routing information in Service Units and Service Assemblies involves the following activities.

- Creating individual Service Units for each binding component, depending on the WSDL extensions used.
- Creating a `jbi.xml` descriptor for each Service Unit with the `<consumes>` and `<provides>` attributes declared.
- Creating a service connection section in the Service Assembly descriptor to resolve all the mappings among partner links and endpoints.

Service Unit Deployment for a Service Engine

When Service Units are intended for a Service Engine, you must create the Service Unit deployment descriptor (`jbi.xml`) as defined in Section 6.3 of the JBI specification.

According to the specification, the content of the Service Unit jbi.xml must conform to the following rules:

- The attribute binding-component must be false.
- For each partner link with the myRole attribute, an element, provides, must be created with the partner link name as service-name, myRole as the endpoint-name, and the port type of the partner link type as the interface-name.
- For each partner link with the partnerRole attribute, an element, consumes, must be created with the partner link name as service-name, partnerRole as the endpoint-name, and the port type of the partner link type as the interface-name.

The basic format of the Service Unit jbi.xml descriptor is as follows:

```
<?xml version='1.0'?>
<jbi version="1.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://java.sun.com/xml/ns/jbi"
        xsi:schemaLocation="http://java.sun.com/xml/ns/jbi jbi.xsd">
    <services binding-component="false">
        <provides interface-name=QNAME
                  service-name=QNAME
                  endpoint-name=text/>
        <consumes interface-name=QNAME
                  service-name=QNAME
                  endpoint-name=text
                  link-type="standard"/>
    </services>
</jbi>
```

JBI Composite Application Service Assembly

The JBI composite application Service Assembly project must create the Service Assembly deployment descriptor (jbi.xml), as defined in Section 6.3 of the JBI specification. There are two major enhancements:

Service Connections

- The Service Assembly jbi.xml must include a connections element specifying the mapping between consuming and providing service endpoints.

- Unconnected service endpoints must be resolved by matching the port type (designated by `interface-name`) of the consuming endpoints with that of the providing endpoints.

Connection Elements

- A connection element must be added for each resolved pair of endpoints.

Service Unit Deployments Intended for a Binding Component

You must create a separate service unit deployment package for each BC needed by the project. A BC is considered needed if it implements one or more endpoints specified in the connections section of the Service Assembly descriptor.

The BC deployment package must include all referenced WSDLs, XSDs, and associated data files.

The Service Unit `jbi.xml` must specify all consuming and providing endpoints that are implemented or used by the binding component. Note that this descriptor does not contain any mapping to or from the concrete endpoint of an SE. Therefore, it can be generated by static inspection of the deployment artifacts (BPEL, WSDL, and so on) and does not rely on the port mapping, which is done at a higher level in the Service Assembly connection section.

Each BC names the internal endpoints it provides. The consumer addresses the endpoint according to the external endpoint, port, or interface name. Note the contrast to SE naming, which uses partner link details.

To associate a given endpoint with a target Service Unit (and therefore target component), look at the WSDL extensions used. For example, if a `<file:binding/>` tag is present in the binding, then the endpoint is directed toward the file BC.

Sample Descriptors

Consider the following SE Service Unit `jbi.xml` descriptor example. For the inbound message, the SE is the provider; for the outbound message, the SE is the consumer.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<jbi xmlns="http://java.sun.com/xml/ns/jbi"
     xmlns:ns1="http://localhost/loanRequestorBpel/loanRequestor"
     xmlns:ns2="http://www.seebeyond.com/eInsight/loanRequestorBpel"
     xmlns:ns3="urn:LoanProcessorEJB/wsdl"
     version="1.0">
  <services binding-component="false">
    <provides endpoint-name="loanRequestorRole_myRole"
              interface-name="ns1:IRequestLoan"
              service-name="ns2:bpelImplemented"/>
    <consumes endpoint-name="loanProcessorEJBPartnerRole_partnerRole"
            interface-name="ns3:LoanProcessorEJBSEI"
            service-name="ns2:ejbInvoked"/>
  </services>
</jbi>
```

The following code example is a binding component Service Unit `jbi.xml` descriptor, targeted at only one binding component. For the inbound message, the BC is the consumer; for the outbound message, the BC is the provider.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<jbi xmlns="http://java.sun.com/xml/ns/jbi"
     xmlns:ns1="http://localhost/loanRequestorBpel/loanRequestor"
     xmlns:ns2="http://www.seebeyond.com/eInsight/loanRequestorBpel"
     xmlns:ns3="urn:LoanProcessorEJB/wsdl"
     version="1.0">
  <services binding-component="true">
    <provides endpoint-name="LoanProcessorEJBSEIPort"
              interface-name="ns3:LoanProcessorEJBSEI"
              service-name="ns3:LoanProcessorEJB"/>
    <consumes endpoint-name="port"
              interface-name="ns1:IRequestLoan"
              service-name="ns1:loanRequestorService"/>
  </services>
</jbi>
```

The following example is an excerpt from the connection section in the Service Assembly `jbi.xml`. Note how it essentially maps the endpoints declared in the above descriptors. For inbound messages, it maps from the BC consumer address (external `endpoint-name`) to the concrete SE endpoint (partner link names). For outbound messages, it maps from the SE consumer address (partner link names) to the concrete BC endpoint (external `endpoint-names`).

```
<connections>
  <connection>
    <consumer endpoint-name="port"
              service-name="ns1:loanRequestorService"/>
    <provider endpoint-name="loanRequestorRole_myRole"
              service-name="ns2:bpelImplemented"/>
  </connection>
  <connection>
    <consumer endpoint-name="loanProcessorEJBPartnerRole_partnerRole"
              service-name="ns2:ejbInvoked"/>
    <provider endpoint-name="LoanProcessorEJBSEIPort"
              service-name="ns3:LoanProcessorEJB"/>
  </connection>
</connections>
```

JBI Management, Monitoring, and Administration

The JBI environment is administered with Java Management eXtensions (JMX). All JBI components (SEs and BCs) must provide specified management interfaces. In addition, the JBI Framework provides specified JMX Management Beans (MBeans) to help manage the JBI environment-provided infrastructure as well as all the JBI components.

Standard interfaces for management monitoring and administration are provided for the following activities in any JBI environment:

- Installation of JBI components and shared libraries
- Deployment of JBI artifacts (Service Assemblies and Service Units) to installed components
- Start/stop/shutdown of JBI components (SEs and BCs)
- Start/stop/shutdown of composite services (a group of related services) referred to as a Service Assembly.

Development of a JBI Component (Service Engine or Binding Component)

Before a component can be plugged in to a JBI Framework, you must implement the following interfaces and perform related tasks.

1. Implement a Bootstrapper (implements the `javax.jbi.component.Bootstrap` interface). The bootstrapper allows the installation of a component to be customized (for example, creation of database tables as part of installation) It traps install and uninstall events.

2. Implement your Component (implements the `javax.jbi.component.Component` interface). The component functions as a single point for the JBI implementation to query the other component-supplied interfaces as well as metadata for component-supplied services. This implementation provides access to the component's life cycle implementation and Service Unit manager.

3. Implement a Component Life Cycle (implements the `javax.jbi.component.ComponentLifecycle` interface). The component life cycle manages component life cycle states such as stopped, shutdown, and running.

4. (Optional) Implement a Service Unit Manager (implements the `javax.jbi.component.ServiceUnitManager` interface). Implement a service unit manager for components that support deployment of Service Units to the component.

5. Create a `jbi.xml` deployment descriptor for the component, and package the JBIComponent `.jar` file.

6. Install the new JBI component in the JBI meta-container.

Service Unit and Service Assembly Creation and Packaging

FIGURE 11 shows how Service Units and the Service Assembly are packaged.

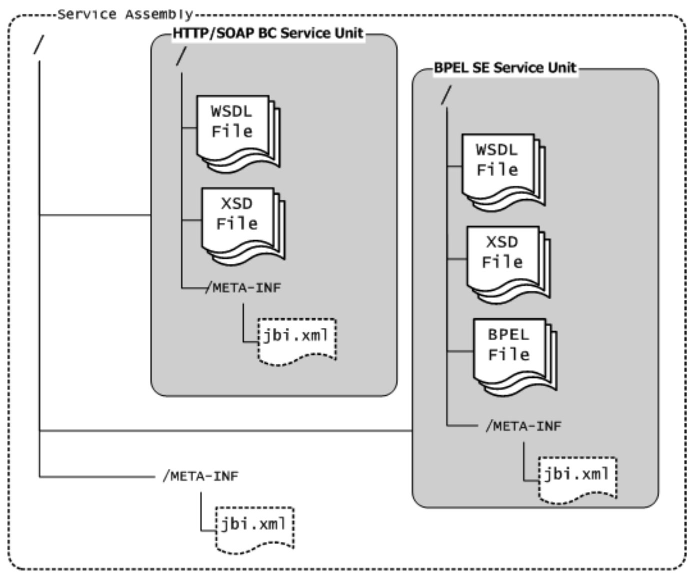


FIGURE 11 Packaging Service Units and Service Assembly

Each Service Unit contains the following elements:

- A JBI descriptor – Descriptors for services consumed and provided in a `jbi.xml` file.
- Component-specific artifacts that can be deployed to a JBI component (SE or BC)

The Service Units are packaged in a composite Service Assembly. Each Service Assembly contains the following elements:

- A JBI descriptor listing each Service Unit intended to be deployed on an SE or a BC in a `jbi.xml` file
- One or more Service Units

Service Assembly Deployment to the JBI Environment

A Service Assembly is a deployment unit that wraps multiple Service Units intended to be deployed on multiple target JBI component containers. A Service Assembly provides a convenient way to group many Service Unit deployments into one unit so that they can be managed as a whole.

When you deploy a Service Assembly, the JBI deployment service breaks the composite Service Assembly into its constituent Service Units and `jbi.xml` deployment descriptors.

Based on the information provided in the `jbi.xml` deployment descriptors, the deployment service deploys the Service Units to the target JBI component containers, as shown in FIGURE 12.

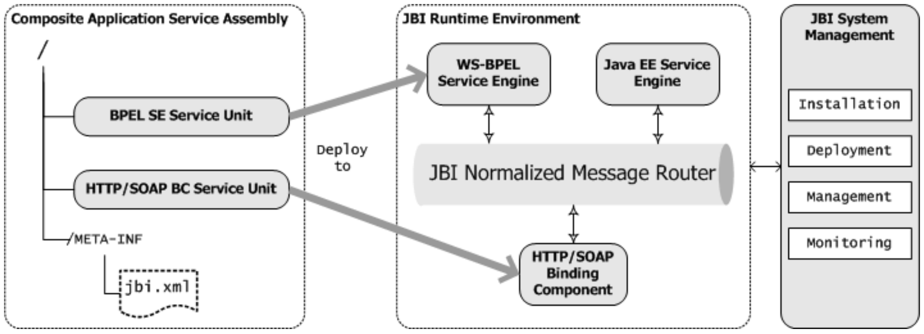


FIGURE 12 Deploying From a Service Assembly to JBI Component Containers

Java EE Service Engine

Sun Java System Application Server 9.0, Platform Edition (hereafter Application Server) contains a JSR 208-compliant JBI component that connects Java EE web services and Java EE web service clients with the normalized message router (NMR). Thus, the Java EE Service Engine can act as both a service provider and service consumer in NMR. It is automatically installed in the JBI runtime in every Java EE SDK installation that contains JBI.

Role of Java EE Service Engine As a Service Provider

The Java EE Service Engine functions as a service provider by enabling an endpoint in NMR (see FIGURE 13).

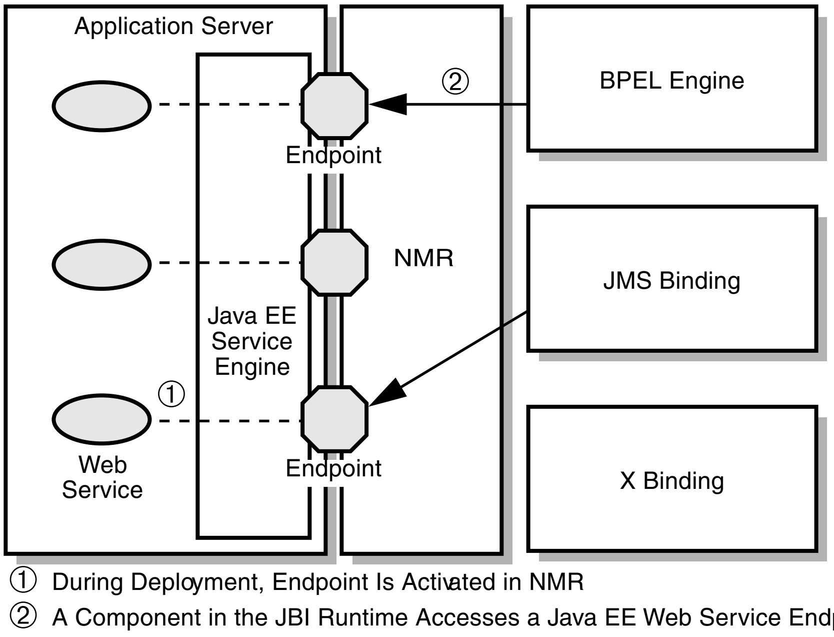

Application Server
②
BPEL Engine
Endpoint
NMR
JMS Binding
Java EE Service Engine
①
Web Service
Endpoint
X Binding
① During Deployment, Endpoint Is Activated in NMR
② A Component in the JBI Runtime Accesses a Java EE Web Service Endpoint

FIGURE 13 Java EE Service Engine As Service Provider

When a Java EE web service is deployed, the deployment runtime of Application Server notifies the Java EE Service Engine so that an endpoint is enabled in the NMR of the JBI runtime. The notification enables any component deployed in NMR to access the Java EE web service. For example, a BPEL application running inside the BPEL engine can access the Java EE web service by sending a normalized message to the endpoint enabled by the Java EE Service Engine.

This way of accessing Java EE web services is an alternative to the normal web service client access defined by JAX-WS.

Role of Java EE Service Engine As a Service Consumer

When a Java EE application needs to access an endpoint of a service provider deployed in the JBI runtime environment, the Java EE Service Engine acts as a bridge between the Java EE container and the NMR. In this case, the Java EE Service Engine normalizes the SOAP message that otherwise would have been used in the JAX-WS communication and sends it to the NMR. This normalized message is handled by a service that has been enabled by a service provider deployed in the JBI runtime environment.

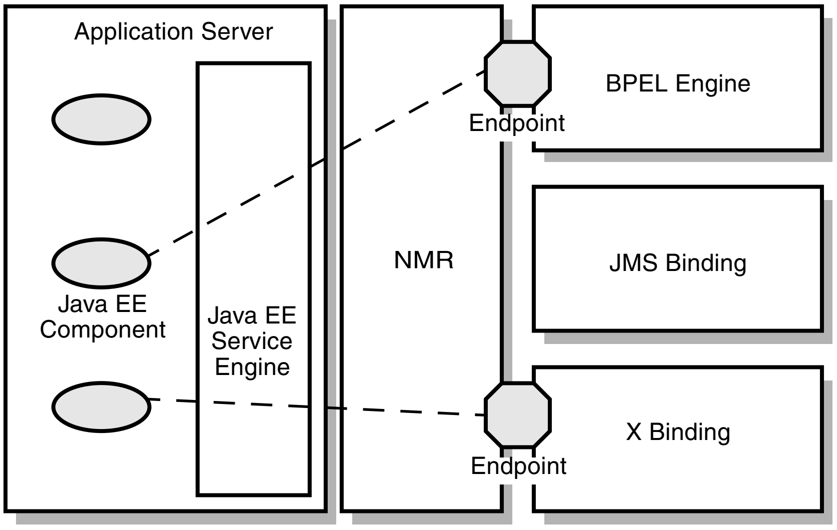


FIGURE 14 Java EE Component Accesses an Endpoint Registered in NMR

To route the JAX-WS requests through NMR, the `service-ref` element in the Sun-specific deployment descriptor file needs to be modified as shown in the following code example.

```
<sun-web-app>
   <service-ref>
     <service-ref-name>
       sun-web.serviceref/calculator
     </service-ref-name>
     <port-info>
       <wsdl-port>
          <namespaceURI>
              http://example.web.service/Calculator
          </namespaceURI>
          <localpart>
            CalculatorPort
```

```
           </localpart>
        </wsdl-port>
        <service-endpoint-interface>
           service.web.example.calculator.Calculator
        </service-endpoint-interface>
        <stub-property name="jbi-enabled" value="true"/>
      </port-info>
    </service-ref>
  </sun-web-app>
```

By changing the stub-property `name="jbi-enabled"` value to `"false"`, you can prevent the Java EE Service Engine from intercepting the request.

HTTP/SOAP JBI Binding Component

HTTP/SOAP (HTTP/SOAPBC) is a JBI 1.0 BC that supports the SOAP protocol over HTTP.

The HTTP/SOAPBC complies with the WSDL 1.1 and SOAP 1.1 specifications (the reference implementation example uses WSDL 2.0, SOAP 1.2). Message exchanges to and from the HTTP/SOAPBC use the JBI WSDL 1.1 wrapper for the normalized message.

SOAP binding from the WSDL 1.1 specification is implemented (it does not use HTTP Get/Post or MIME bindings). SOAP binding follows WS-I 1.0 conventions and adds additional support for nonconforming components. It supports document- and RPC-style web services and literal use.

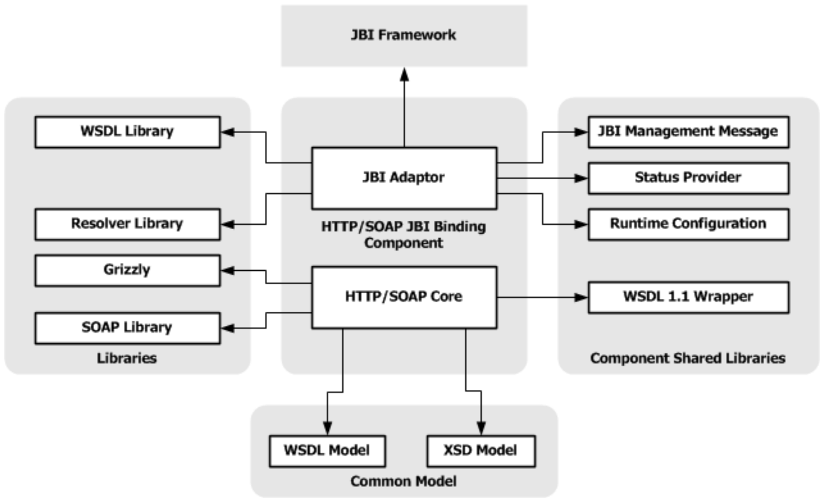


FIGURE 15 HTTP/SOAP Binding Component

The HTTP/SOAPBC supports the common convention of WSDL retrieval by appending `?wsdl` to the URI. It uses XML catalogs following the OASIS Committee Specification. XML catalogs allow the component to resolve schemas locally without resorting to network access. HTTP/SOAPBC packages an embedded HTTP server (Grizzly). HTTP/ SOAPBC uses asynchronous input/output in the server to service thousands of concurrent incoming requests. Outbound requests are handled through SOAP with Attachments API for Java (currently, SAAJ 1.2).

The HTTP/SOAPBC currently only handles ports that are not serviced by the application server. The HTTP/SOAPBC supports JBI Service Unit deployments to define the web services that will be provisioned or consumed. It makes use of the WSDL extensibility (standard SOAP extensions) to define external communication details for the web services to provision or consume.

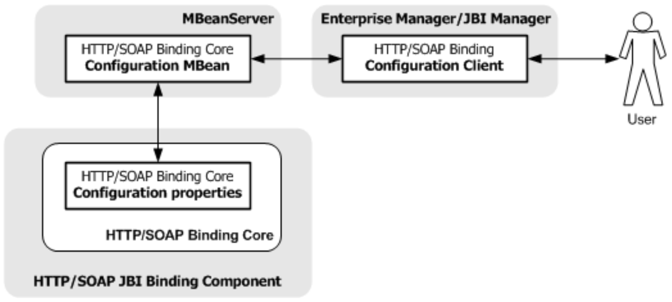

MBeanServer
HTTP/SOAP Binding Core Configuration MBean
Enterprise Manager/JBI Manager
HTTP/SOAP Binding Configuration Client
User
HTTP/SOAP Binding Core Configuration properties
HTTP/SOAP Binding Core
HTTP/SOAP JBI Binding Component

FIGURE 16 HTTP/SOAP Runtime Configuration

WS-BPEL JBI Service Engine

The WS-BPEL JBI Service Engine (hereafter called BPEL Service Engine), shown in FIGURE 17, is a JBI engine component that provides services for executing business processes.

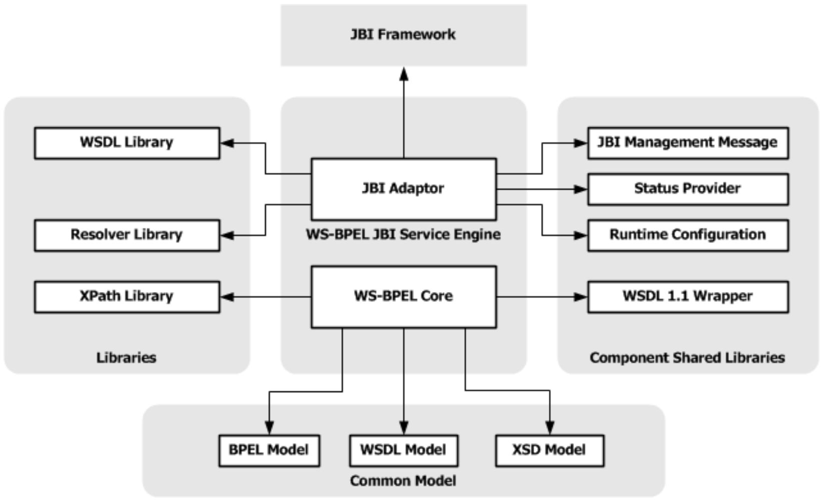


FIGURE 17 BPEL Service Engine

The BPEL Service Engine is a standard JBI 1.0 service engine component. It supports business processes that conform to the Web Services Business Process Execution Language (WS-BPEL) 2.0 specification. It provisions and consumes web services described in WSDL1.1 and exchanges messages in JBI-defined XML document format for wrapped WSDL 1.1 message parts.

The BPEL Service Engine supports request/reply, asynchronous one-way invocations, and direct invocation between two business processes. It can monitor endpoint status. It offers a command-line facility for building a Service Assembly and testing the deployed service.

The BPEL Service Engine can be configured in one of three modes: static, deployment, or runtime:

- **Static** – Parameter values, once loaded, can only be modified by reinstalling the engine.
- **Deployment** – Parameter values can be changed without reinstallation, but only until the engine is started or restarted; they remain in effect throughout business process execution.
- **Runtime** – Parameter values can be changed even while business processes are running. See FIGURE 18.

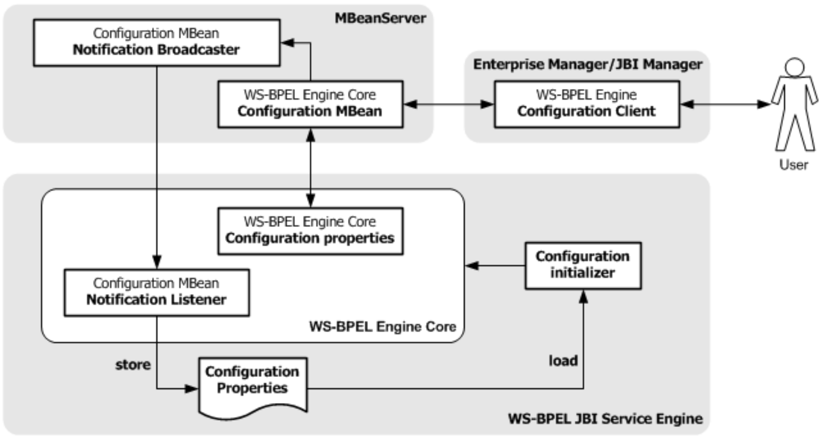


FIGURE 18 BPEL Service Engine Runtime Configuration

Putting It All Together: The Loan Processing Composite Application

This section presents an example of how a composite application can be created and deployed in the JBI environment. It uses the HTTP/SOAPBC, the WS-BPEL SE, and the Java EE SE components. It shows how these components can be orchestrated to solve a business problem.

Download the files related to this NetBeans 5.0 project from the links given in the section titled "Resources" on page 62 of this document.

The following JBI components are used in this example:

- WS-BPEL Service Engine
- HTTP/SOAP Binding Component (no security)
- Java EE Service Engine

The composite application is an Enterprise JavaBeans application exposed as a WebService. The JBI Service Assembly consists of the following Service Units:

- WS-BPEL engine provider, consuming all the other provided services
- HTTP/SOAP binding consumer, for receiving requests

Business Use Case

The composite application satisfies the following business use case, illustrated in FIGURE 19.

The user applies for a loan by filling out a loan request, including information such as personal identifying information, amount of loan requested, and credit history.

When the loan request is received, the personal information supplied by the user is verified across an existing database, and approval is granted or rejected based on the information and the amount requested.

After certain formalities are fulfilled, a report, in the form of an approval letter, is generated and sent to the user, confirming the approval of the loan. If the loan is rejected for some reason, then a report showing the reason for the rejection is generated and displayed.

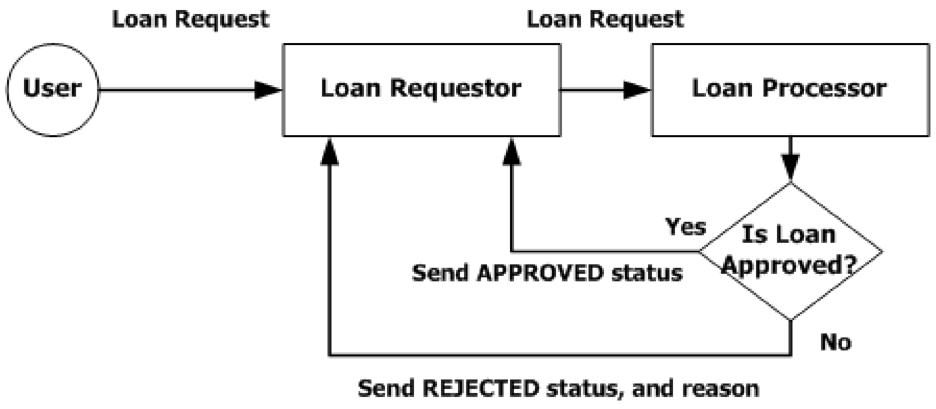


FIGURE 19 Business Use Case Flow Diagram

Implementation With JBI

The above business case can be implemented according to SOA principles and the JBI specification. The Service Assembly (composite application), which consists of all the Service Units, must be deployed. FIGURE 20 describes such a scenario. The description of all the activities that are performed at each step follows the figure.

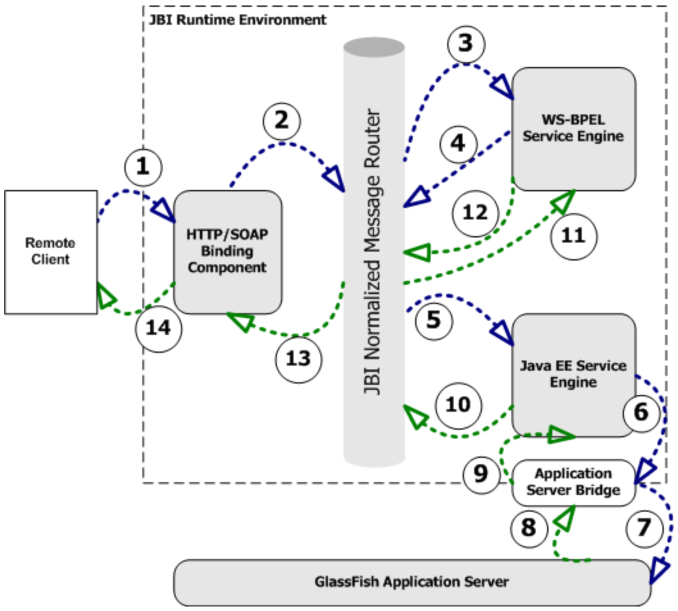
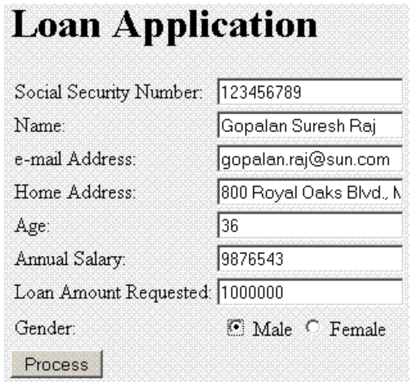

JBI Runtime Environment
Remote Client
HTTP/SOAP Binding Component
JBI Normalized Message Router
WS-BPEL Service Engine
Java EE Service Engine
Application Server Bridge
GlassFish Application Server
1
2
3
4
5
6
7
8
9
10
11
12
13
14

FIGURE 20 Sequence of Steps Describing Message Flow in the Loan Processing Application

Step 1 – The user submits the information that is required for obtaining a loan. The information can be supplied though a web services client or an application client. For this scenario, we assume a web client. The user fills out the form (see FIGURE 21), from which information such as name, age, salary, e-mail address, and so on, is collected. When the form is submitted, a request is made to the appropriate URL.

Step 2 – The HTTP/SOAP BC provides a consumer endpoint for the URL. The HTTP/SOAP BC can process this request because the WSDL that corresponds to this service (`http://localhost:22000/loanRequestor`) has already been deployed in the HTTP/SOAP BC, and the port corresponding to this service is the correct port. Any other information that the BC requires (security-partner link-related information) is supplied along with the Service Unit.

When the HTTP/SOAP BC receives this request, it unmarshals the SOAP message, removes the SOAP headers, constructs a normalized message (based on the JBI specification), and addresses it to the component that provides this service in the JBI system. This message is then sent on the delivery channel. The normalized message router (NMR) routes the message to the BPEL service engine with service name `http://localhost/loanRequestorBpel/loanRequestor` and port `loanRequestorService`.



Loan Application

Social Security Number:	123456789
Name:	Gopalan Suresh Raj
e-mail Address:	gopalan.raj@sun.com
Home Address:	800 Royal Oaks Blvd., N
Age:	36
Annual Salary:	9876543
Loan Amount Requested:	1000000
Gender:	[✓ Male] Female

FIGURE 21 Loan Application Request Form

Step 3 – The BPEL SE receives a request on its delivery channel. The service information and the BPEL script that must be executed for this request is obtained from the deployed Service Unit. The BPEL SE starts executing the BPEL script corresponding to this service and port. It also validates the message received with the message type described in the WSDL.

Step 4 – The first activity in the BPEL script happens to be an invocation of another service. The BPEL SE constructs a normalized message and updates the message content with the loan request message and the message headers, with the service name `urn:LoanProcessorEJB/wsdl}LoanProcessorEJB` and port `LoanProcessorEJBSEIPort`. This message, too, is sent on the delivery channel. The BPEL service engine has no knowledge of the provider of the service other than the service definition in the WSDL file.

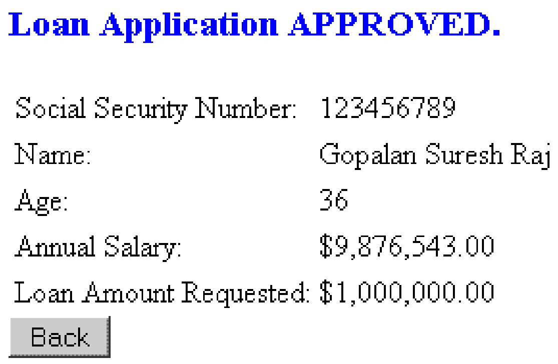
Steps 5 to 10 – Because the endpoint is an internal endpoint (that is, provided by some engine in the JBI runtime system) and has been activated by the Java EE SE, the message is delivered to the delivery channel of the Java EE SE. A deployment to the Java EE SE is not a Service Unit but could be any application server component, such as an EJB component or a servlet. A special provision in Sun Java System Application Server specifies that any EJB exposed as a Web Service automatically has its service endpoints registered with the JBI normalized message router (NMR).

Similarly, a provision in the `web.xml` deployment descriptor for the Application Server lets you indicate whether or not a servlet is JBI enabled. If it is not JBI-enabled, then the servlet request processing proceeds normally; if it is JBI enabled, then any request to the servlet is trapped by the Java EE Service Engine and a request is sent to the NMR. The Java EE SE also normalizes the request message.

In this example, Java EE SE receives a request for a service that is provided by a JBI-enabled EJB component deployed in the application server. The SE passes the request to the EJB component after denormalizing the request and applying the SOAP wrappers to it (because the EJB component is a web service implementation, it expects a SOAP message). The response returned by this web service is likewise stripped of SOAP wrappers and normalized into a JBI normalized message. The resulting message is returned as a response to the consumer (the BPEL SE). The web service is the `LoanProcessor` web service, which approves or rejects loan requests.

Steps 11, 12 – The BPEL SE receives the response from the provider and validates the response to determine whether or not it is a fault. If the response is valid, the BPEL SE proceeds with the execution of the next service; otherwise, it executes the fault handlers. The response is merely an approved or rejected status message from the `LoanProcessor` corresponding to the service name `{urn:LoanProcessorEJB/wsdl}LoanProcessorEJB` and port `LoanProcessorEJBSEIPort`. The BPEL SE sends the reply back through the NMR to the consumer.

Steps 13, 14 – The HTTP/SOAP BC that initiated the Loan request receives this response. It denormalizes the response message and creates a SOAP response (see FIGURE 22) to be sent to the web-service client that invoked it.


Loan Application APPROVED.

Social Security Number:	123456789
Name:	Gopalan Suresh Raj
Age:	36
Annual Salary:	$9,876,543.00
Loan Amount Requested:	$1,000,000.00

FIGURE 22 Loan Request Response Presented to User

The following sections of this document provide programming details to give you a deeper understanding of the example application.

Web Services Business Process Execution Language (WS-BPEL)

The Web Services Business Process Execution Language (WS-BPEL) is used to create processes capable of invoking other processes, all of which correspond to a business workflow. Coordinating the interaction of these processes is known as orchestration. WS-BPEL enables orchestration by providing standardized integration logic and process automation between web services.

Using constructs derived from the Web Service Definition Language (WSDL), WS-BPEL describes inbound and outbound process interfaces so that a process can easily be integrated into other processes or applications. These interface descriptions enable consumers of a process to inspect and invoke a WS-BPEL process just as they would any other web service.

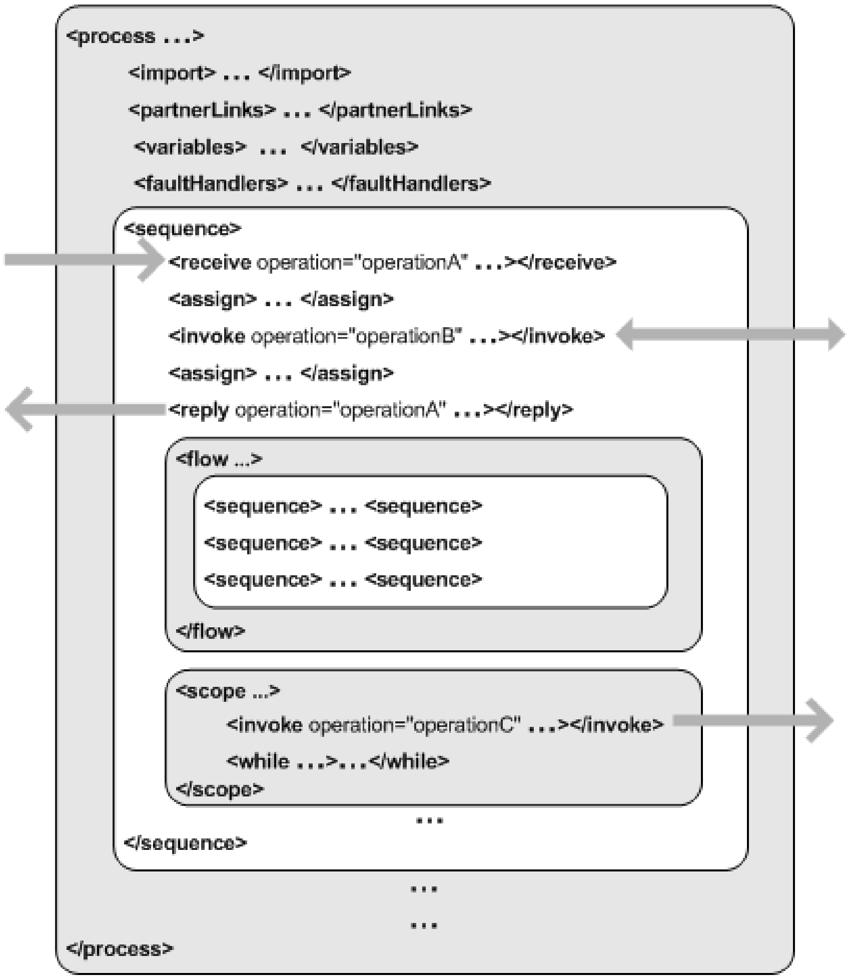
BPEL – The Language in a Nutshell

BPEL is a rigorous language that extends web services for interacting processes. BPEL business processes build stateful, conversational orchestrations from web services. BPEL processes are expressed in XML notation.

To design a composite application using WS-BPEL, you must become acquainted with the BPEL language. While this article cannot describe all the details of the BPEL language and its capabilities, it introduces features of the language now as background for understanding the example program presented later.

Users of the Sun Java Enterprise Pack can use the Sun BPEL editor and the many tools that come with it to create their orchestrations and therefore will not have to create BPEL XML code by hand. However, knowledge of the language elements is still useful since the Sun BPEL editor references these constructs and language elements in its graphical widgets.

FIGURE 23 shows a typical BPEL process definition skeleton.



```
<process ...>
        <import> ... </import>
        <partnerLinks> ... </partnerLinks>
        <variables> ... </variables>
        <faultHandlers> ... </faultHandlers>
    <sequence>
        <receive operation="operationA" ...></receive>
        <assign> ... </assign>
        <invoke operation="operationB" ...></invoke>
        <assign> ... </assign>
        <reply operation="operationA" ...></reply>
        <flow ...>
            <sequence> ... <sequence>
            <sequence> ... <sequence>
            <sequence> ... <sequence>
        </flow>
        <scope ...>
            <invoke operation="operationC" ...></invoke>
            <while ...>...</while>
        </scope>
                        ...
    </sequence>
                        ...
                        ...
</process>
```

FIGURE 23 BPEL Process Definition Skeleton

The process Definition

The root element of any BPEL process is the `<process/>` definition. Each process definition has a `name` attribute and definition-related namespaces, as shown in the following example.

```
<?xml version="1.0" encoding="utf-8" ?>
<process name="loanRequestorBpel"
 targetNamespace="http://www.seebeyond.com/eInsight/loanRequestorBpel"
 xmlns:tns="http://www.seebeyond.com/eInsight/loanRequestorBpel"
 xmlns:bpws="http://schemas.xmlsoap.org/ws/2004/03/business-process/"
 xmlns:ns0="http://localhost/loanRequestorBpel/loanRequestor"
 xmlns:ns1="http://localhost/loanRequestorBpel/
                                        LoanProcessorEJBWrapper"
 xmlns="http://schemas.xmlsoap.org/ws/2004/03/business-process/"
 xmlns:ns3="urn:LoanProcessorEJB/wsdl"
 xmlns:ns2="http://www.w3.org/2001/XMLSchema">
    <import ...>
    </import>
    <partnerLinks>
       ...
    </partnerLinks>
    <variables>
       ...
    </variables>
    <sequence>
       ...
    </sequence>
       ...
</process>
```

The import Element

The `<import/>` element, illustrated in the following example, indicates a dependency on external XML Schema or WSDL definitions. Each `<import/>` element contains three mandatory attributes:

- The `namespace` attribute specifies the URI namespace of the imported definitions.
- The `location` attribute contains a URI giving the location of a document that contains relevant definitions in the namespace specified. The document located at the URI must contain definitions belonging to the same namespace as indicated by the `namespace` attribute.
- The `importType` attribute identifies the type of document being imported by providing the URI of the encoding language. The value must be set to `"http:// www.w3.org/2001/XMLSchema"` when XML Schema 1.0 documents are imported, and to `"http://schemas.xmlsoap.org/wsdl/"` when WSDL 1.1 documents are imported.

```
<import namespace="http://localhost/loanRequestorBpel/loanRequestor"
        location="loanRequestor.wsdl"
        importType="http://schemas.xmlsoap.org/wsdl/">
</import>
<import namespace="http://localhost/loanRequestorBpel/
                                      LoanProcessorEJBWrapper"

 location="LoanProcessorEJBWrapper.wsdl"
 importType="http://schemas.xmlsoap.org/wsdl/">
</import>
```

The partnerLinkType and role Definitions in WSDL Files

The `<partnerLinkType/>` element is embedded directly within the WSDL file of every partner process and service process involved in a BPEL orchestration.

`<partnerLinkType/>` elements defined in the partner's WSDL file identify the WSDL `portType` (interface) element referenced by the `partnerLink` in the service process's BPEL `process` definition.

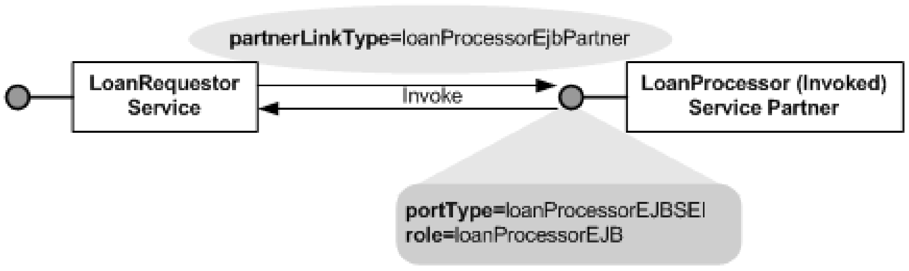

partnerLinkType=loanProcessorEjbPartner
LoanRequestor Service
Invoke
LoanProcessor (Invoked) Service Partner
portType=loanProcessorEJBSEI
role=loanProcessorEJB

FIGURE 24 partnerLinkType Behavior

The `<partnerLinkType/>` element contains one `<role/>` element for each role (service provider or service consumer) the service process can play, as defined by its `partnerLink` element—`myRole` (indicating a service provider) and `partnerRole` (indicating a service consumer) attributes—in the service process's BPEL process definition. Therefore, a `partnerLinkType` can have either one or two `<role/>` child elements. The following example shows one `<role/>` child element.

```
<definitions name="LoanProcessorEJBWrapper"
targetNamespace="http://localhost/loanRequestorBpel/
                                LoanProcessorEJBWrapper"
  xmlns:tns="http://localhost/loanRequestorBpel/
                                LoanProcessorEJBWrapper"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:plink="http://schemas.xmlsoap.org/ws/2004/03/partner-link/"
  ...
  >
...
    <plink:partnerLinkType name="loanProcessorEjbPartner">
        <plink:role name="loanProcessorEJB"
                    portType="ns:LoanProcessorEJBSEI">
        </plink:role>
    </plink:partnerLinkType>
...
</definitions>
```

In situations when a service process has the same relationship with multiple partner processes, the service process's BPEL `partnerLink` elements can reference the same `partnerLinkType`.

The partnerLinks and partnerLink Definitions

The `<partnerLink/>` definition defines the `portType` (interface) of the partner process that will participate in BPEL orchestrations of the business process being defined. These partner processes can act as clients to the service process being defined or can be invoked by the service process.

The `<partnerLink/>` element encodes communication exchange information between the service process and its partner processes. The role of the service process will vary according to the nature of communication with its partner process.

For the example in FIGURE 25, when the `LoanRequestor` service process invokes a `LoanProcessorEJB` partner process, it may act in a `LoanRequestor` role and the `LoanProcessorEJB` partner process may act in a `LoanProcessorEJB` role.

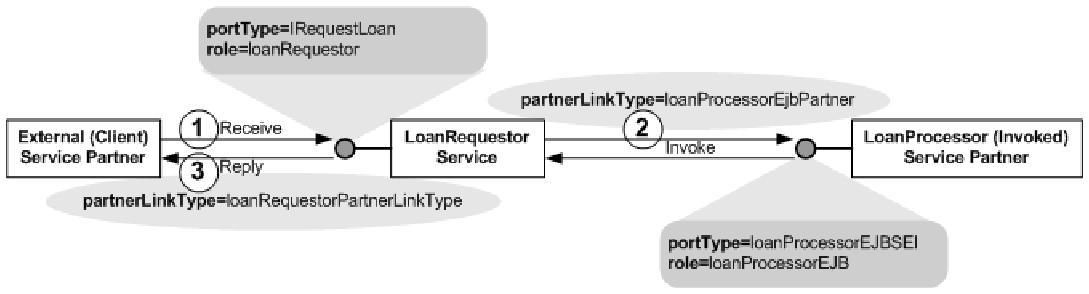

portType=IRequestLoan
role=loanRequestor
partnerLinkType=loanProcessorEjbPartner
External (Client) Service Partner
1 Receive
3 Reply
LoanRequestor Service
2 Invoke
LoanProcessor (Invoked) Service Partner
partnerLinkType=loanRequestorPartnerLinkType
portType=loanProcessorEJBSEI
role=loanProcessorEJB

FIGURE 25 partnerLinkType in Loan Processor Example

The `<partnerLink/>` element contains the `myRole` and `partnerRole` attributes that establish the roles for the service process and its partner process, respectively, as illustrated in the following code example.

```
<partnerLinks>
    <partnerLink name="bpelImplemented"
                 partnerLinkType="ns0:loanRequestorPartnerLinkType"
                 myRole="loanRequestor"/>
    <partnerLink name="ejbInvoked"
                 partnerLinkType="ns1:loanProcessorEjbPartner"
                 partnerRole="loanProcessorEJB"/>
</partnerLinks>
```

The `myRole` attribute is used when the service process acts as the service provider and is being invoked by a service consumer partner process client. Similarly, the `partnerRole` attribute identifies the service provider partner process that the service process is invoking when acting in a service consumer client role.

The variables and variable Definitions

The `<variable/>` element is used by the service process to store state information related to immediate workflow logic. Entire XML messages can be stored in a variable and retrieved later by a process. Only predefined XSD schema type data can be stored into these variables.

As defined by the following attributes, several data types can be stored in a process variable:

- The `messageType` attribute allows the variable to contain an entire WSDL-defined message, as illustrated in the following code example.

- The element attribute allows a variable to contain an XSD element.
- The type attribute allows a variable to contain any XSD simpleType.

In the following example, variables with the messageType attribute are defined for each of the input and output messages handled by the process definition. The value of this attribute is the message name from the partner process definition.

```
<variables>
  <variable name="requestLoan_Input"
            messageType="ns0:requestLoanMessage">
  </variable>
  <variable name="requestLoan_Output"
            messageType="ns0:requestLoanResponseMessage">
  </variable>
  <variable name="processApplication_Input"
            messageType="ns3:LoanProcessorEJBSEI_processApplication">
  </variable>
  <variable name="processApplication_Output"
     messageType="ns3:LoanProcessorEJBSEI_processApplicationResponse">
  </variable>
</variables>
```

Structured Activity: The sequence Element

Structured activities combine primitive activities into more complex processes. The <sequence/> element defines an ordered sequence of activities so that they are executed in the order in which they are listed, as illustrated in the following example. <sequence/> elements can be nested, so sequences can be defined within sequences.

```
<sequence>
    <receive>
       ...
    </receive>
    <assign>
       ...
    </assign>
    <invoke>
       ...
    </invoke>
    <assign>
       ...
```

```
      </assign>
      <reply>
         ...
      </reply>
  </sequence>
```

Web Service Activity: The receive Element

Web service activities are defined by BPEL to create WebService compositions. The `<receive/>` element defines the information expected by a service process acting as a service provider waiting to receive a request from an external client partner process acting as a service consumer.

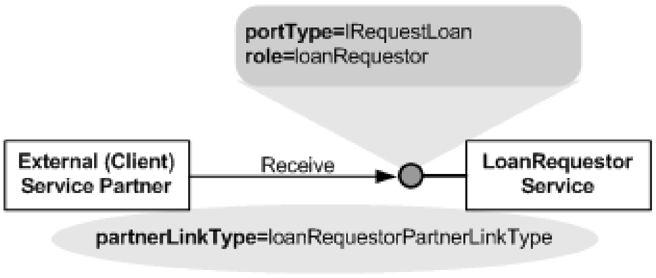

portType=IRequestLoan
role=loanRequestor
External (Client) Service Partner
Receive
LoanRequestor Service
partnerLinkType=loanRequestorPartnerLinkType

FIGURE 26 Receive Task

The attributes that are required for defining a receive task are as follows:

- `partnerLink` attribute – This attribute value points to the client partner process (the service consumer).
- `portType` attribute – This attribute value contains the service process's (service provider) port type (or interface) that will be waiting to receive the service request from the client partner service consumer.
- `operation` attribute – This attribute value contains the service process's operation that will be receiving the client request.
- `variable` attribute -This attribute value contains the process definition variable that will store the incoming request message.
- `createInstance` attribute – This attribute value is set to `"yes"` if, on receiving a client request, a new service process instance must be created and launched. If this attribute value is set to `"no"`, a new service process instance is not created when a client request is received.

```
<receive partnerLink="bpelImplemented"
         portType="ns0:IRequestLoan"
         operation="requestLoan"
         variable="requestLoan_Input"
         createInstance="yes">
</receive>
```

The `<receive/>` element is also used to receive callback messages during an asynchronous message exchange.

Basic Activity: The assign, copy, from, and to Elements

Basic activities are defined by BPEL to create WebService compositions. The `<assign/>`, `<copy/>`, `<from/>`, and `<to/>` elements let you copy data from one process variable to the other throughout the service process.

```
<assign>
    <copy>
        <from>$requestLoan_Input.requestPart/aSSN</from>
        <to>$processApplication_Input.parameters/String_1</to>
    </copy>
    <copy>
        <from>$requestLoan_Input.requestPart/aName</from>
        <to>$processApplication_Input.parameters/String_2</to>
    </copy>
    <copy>
        <from>$requestLoan_Input.requestPart/aAddress</from>
        <to>$processApplication_Input.parameters/String_3</to>
    </copy>
    <copy>
        <from>$requestLoan_Input.requestPart/aEmailAddress</from>
        <to>$processApplication_Input.parameters/String_4</to>
    </copy>
    <copy>
        <from>$requestLoan_Input.requestPart/aAge</from>
        <to>$processApplication_Input.parameters/int_5</to>
    </copy>
    <copy>
        <from>$requestLoan_Input.requestPart/aGender</from>
        <to>$processApplication_Input.parameters/String_6</to>
    </copy>
    <copy>
        <from>$requestLoan_Input.requestPart/aAnnualSalary</from>
        <to>$processApplication_Input.parameters/double_7</to>
    </copy>
    <copy>
        <from>$requestLoan_Input.requestPart/amountRequested</from>
        <to>$processApplication_Input.parameters/double_8</to>
    </copy>
</assign>
```

Web Service Activity: The invoke Element

Basic activities are defined by BPEL to create WebService compositions. The `<invoke/>` element allows the service process to invoke a one-way or a request-response operation on a port type (or interface) offered by a partner process.

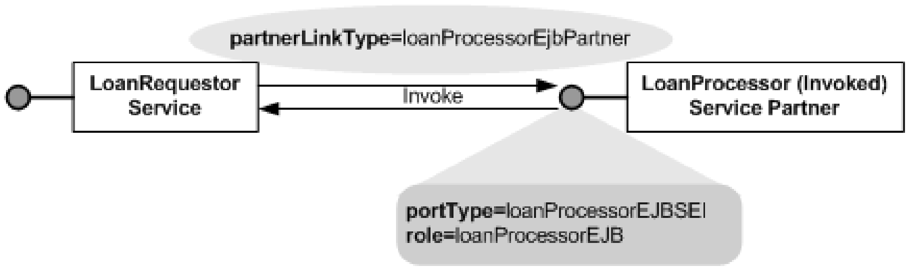

partnerLinkType=loanProcessorEjbPartner
LoanRequestor Service
Invoke
LoanProcessor (Invoked) Service Partner
portType=loanProcessorEJBSEI
role=loanProcessorEJB

FIGURE 27 Invoke Task

The five common attributes of the `<invoke/>` element are as follows:

- `partnerLink` attribute – This attribute value identifies the partner process through this link.
- `portType` attribute – This attribute value identifies the port type (or interface) of the partner process.
- `operation` attribute – The attribute value identifies the operation of the partner process to which the invocation request has to be sent.
- `inputVariable` attribute – This attribute value indicates the input message that is used to communicate with the partner process service operation. This attribute represents a variable element in the process definition with a messageType attribute.
- `outputVariable` attribute – This attribute is used when a request-response message exchange pattern is being used to communicate with the process partner. The return value is stored in a separate variable element.

The following example illustrates the use of the <invoke/> element.

```
<invoke partnerLink="ejbInvoked"
        portType="ns3:LoanProcessorEJBSEI"
        operation="processApplication"
        inputVariable="processApplication_Input"
        outputVariable="processApplication_Output">
</invoke>
```

Web Service Activity: The reply Element

BPEL defines basic activities needed to create WebService compositions. When a synchronous message exchange is being created, each <receive/> element has a corresponding <reply/> element. The <reply/> element allows the service process to send a message in reply to a received message from a process partner client service because the element is associated with the same partnerLink element as its corresponding <receive/> element.

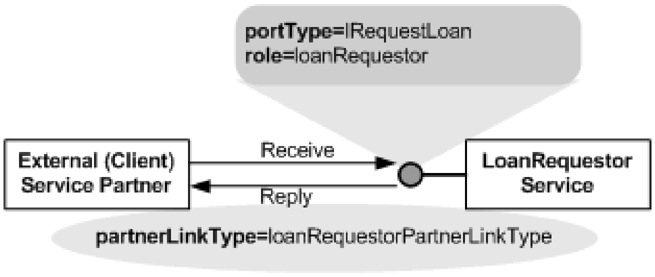

portType=IRequestLoan
role=loanRequestor
External (Client) Service Partner
Receive
Reply
LoanRequestor Service
partnerLinkType=loanRequestorPartnerLinkType

FIGURE 28 Replying to Received Message

The <reply/> element has almost the same set of attributes as the <receive/> element:

- partnerLink attribute – This attribute is the same partnerLink attribute present in the corresponding <receive/> element.
- portType attribute – This attribute is the same portType attribute present in the corresponding <receive/> element.
- operation attribute – This attribute is the same operation attribute present in the corresponding <receive/> element.
- variable attribute – This attribute is the service process variable attribute that is defined in the process definition to hold the message that is returned to the partner process service.

- `messageExchange` attribute – This optional attribute allows the `<reply/>` element to be explicitly associated with the message activity capable of receiving a message such as the `<receive/>` element.

The following example illustrates the use of the reply element.

```
<reply partnerLink="bpelImplemented"
       portType="ns0:IRequestLoan"
       operation="requestLoan"
       variable="requestLoan_Output">
</reply>
```

Basic Activity: The wait Element

The `<wait/>` element makes the service process wait for a period of time. It is used to introduce an intentional delay within the service process. Its values can set either a time or a specified date.

Basic Activity: The throw Element

The `<throw/>` element lets you explicitly trigger a fault in response to a specified condition to indicate that an error has occurred in the service process execution.

Basic Activity: The exit Element

The `<exit/>` element terminates the entire service process orchestration instance by destroying it.

Basic Activity: The empty Element

The `<empty/>` element lets you specify that no activity should occur for a specified condition.

Structured Activity: The while Element

The `<while/>` element lets you define a loop. You can define a group of activities that need to be repeated while a condition is satisfied. The element contains a condition attribute that, as long as it evaluates to `"true"`, will continue to execute the list of activities defined within the `<while/>` element.

Structured Activity: The if Element

The `<if/>`, `<elseif/>`, and `<else/>` elements lets you select an execution branch based on conditional logic.

Structured Activity: The pick Element

The `<pick/>` element blocks and waits for a suitable message to arrive or for a timeout alarm to go off. When either of these triggers occurs, the associated activity is executed and the `<pick/>` completes. The element lets you add an `<onMessage/>` child element and an `<onAlarm/>` child element. Use the `<pick/>` element to respond to external events for which service process execution is suspended.

Structured Activity: The flow Element

The `<flow/>` element enables parallel execution of a collection of sequences. It lets you define multiple activities that can occur concurrently. The element finishes after all branches have finished executing. The child `<link/>` element present in the `<flow/>` element lets you define dependencies between activities.

Structured Activity: The compensationHandler Element

The `<compensationHandler/>` element lets you group a list of activities that define a compensation process that can be executed when certain conditions that require a compensation occur.

Structured Activity: The correlationSets Element

The `<correlationSets/>` element implements correlation, used to associate messages with process instances. A message can belong to multiple correlation sets. Message properties can be defined in WSDL files.

Structured Activity: The eventHandlers Element

The `<eventHandlers/>` element lets you enable process instances that can respond to events during execution of the process logic. You can define `<onMessage/>` and `<onAlarm/>` child elements that trigger process execution upon the arrival of specific types of messages, either after a predefined period of time or at a specified date and time.

Structured Activity: The scope Element

The `<scope/>` element lets you subdivide regions of logic within your process definition into scopes. You can then define `variables`, `faultHandlers`, `correlationSets`, `compensationHandler`, and `eventHandlers` elements that are local to the scope.

Project OpenESB and JBI Support

JSR 208 defines a JBI implementation in terms of a single Java Virtual Machine (JVM), where components, service assemblies, and runtime support (for example, management) are coresident in a JVM implementation. For many enterprise environments, the ability to distribute components and service assemblies across process, machine, and network boundaries is a vital requirement. While this requirement can be met by deploying multiple, independent instances of JBI and linking them through standard communication protocols, this approach has two major drawbacks:

- There is no single point of administration for the entire system. Individual components and service assemblies must be managed by each JBI instance's local management interface.
- Each operation in the system (deploy, install, start, stop, and so on) requires knowledge of the system topology. For example, deploying a service assembly requires knowledge of the physical location of targeted components in the system.

Project OpenESB addresses these problems by introducing a Java Open Enterprise Service Bus built with JBI technology. In essence, OpenESB enables a set of distributed JBI instances to communicate as a single logical entity that can be managed through a centralized administrative interface.

Centralized Management

OpenESB includes a Centralized Administration Server (CAS), which serves as a single point of administration for the entire system. The CAS provides an interface to all standard JBI operations, including instance management facilities such as adding and removing an instance from the ESB. Because OpenESB supports both heterogeneous and homogeneous system topologies, an administrator can partition components and service deployments according to environment-specific requirements—performance, security, or licensing, for example.

Location Transparency

When using OpenESB, the ESB administrator is not burdened with the responsibility of tracking the physical location of each deployment and installation across every JBI instance in the system. The CAS presents an interface to the entire ESB as a single JBI system. With this approach, starting a component is a system-level operation, which is translated by the CAS into instance-specific operations that are executed behind the scenes.

Location transparency also simplifies the provisioning of components and service assemblies. When a component activates a service endpoint in one instance of the ESB, that activation is visible to all instances. Further, ESB supports the concept of a distributed message exchange, whereby two components in separate JBI instances can communicate with each other as if they were colocated in the same JBI instance.

Conclusion

The advantage of the implementation described in the example application is that it offers location transparency and segregates the logic in the business process. These features let you replace or upgrade individual components in the composite application without affecting other components or the process as a whole. Location transparency gives you the flexibility to choose alternative message paths by adding, removing, or modifying the Service Units deployed on the components.

References

OpenESB

- OpenESB home page on java.net (https://open-esb.dev.java.net/)

Service-Oriented Architecture

- SOA Offerings from Sun Microsystems (http://www.sun.com/products/soa/offerings.jsp)
- Developing a Service Engine Component (http://java.sun.com/integration/reference/techart/jbi/)

Resources

Programming Tools

- Download the Java EE 5 Tools Bundle (http://java.sun.com/javaee/downloads)

Example Application

- Download the files related to the example Java NetBean Enterprise Pack 5.5 project (http://java.sun.com/developer/technicalArticles/WebServices/soa3/loanProcessing.zip)

Demonstration Videos for Creating Example Application

You will need the Flash Player plugin to view these videos.

- Creating the LoanProcessor EJB WebServices project (http://java.sun.com/developer/technicalArticles/WebServices/soa3/loanprocessorejb.htm)
- Creating a new BPEL Module project and packaging the required WSDLs and XSDs (http://java.sun.com/developer/technicalArticles/WebServices/soa3/loanprocessing.htm)
- Creating the BPEL-JBI composite application project and deploying to the JBI meta-container (http://java.sun.com/developer/technicalArticles/WebServices/soa3/loanprocessingBpelJBI.htm)
- Testing, debugging, and switching between the BPEL and JPDA debuggers for the deployed JBI composite application project (http://java.sun.com/developer/technicalArticles/WebServices/soa3/loanprocessingJBITest.htm)

Index