

Developing Web Services on the J2EE Platform

Table of Contents:

Abstract	2
Audience.....	2
Web Services Characteristics	2
Exchanging XML Documents in Place of Object Interactions	2
Service Oriented.....	3
Easy Access to Any Web Program.....	3
Dynamic Application Building through Discovery.....	3
Prominent Web Services Standards.....	3
SOAP.....	3
WSDL.....	4
UDDI.....	4
ebXML	5
BTP.....	5
Evolution of J2EE Architecture	5
Simple Web Services Support.....	6
Important Characteristics.....	6
Using Web Services to Integrate Internal Assets	6
Publishing and Discovering Services	6
Motivating Scenario	7
Phase One Development Model	8
Service 1 -- Invoking Service Operation as a Remote Procedure Call (RPC).....	9
Service 2 -- Invoking Service Operation Using One-way Message Exchanges.....	12
Server Side Processing of SOAP Messages	14
XML Transformation	14
Message Listeners	15
J2EE Applications using Advanced Web Services	15
Service Registries.....	15
Service Brokers and Intermediaries.....	15
Impact on J2EE Architecture.....	16
Quality of Service (QoS) Support	17
Web Services Security Model.....	17
J2EE Application Integration with Web Services	18
WorkFlow and Modeling Business Processes.....	20
Business Transactions	20
Conclusion.....	22
Appendix A, Java APIs for Web Services	23
Java API for XML Messaging (JAXM [4])	23
Java APIs for XML based RPC (JAX-RPC [5])	23
Java APIs for XML Registries (JAXR [6])	23
Implementing Enterprise Web Services – JSR 109.....	24
Appendix B, Bibliography	25

Abstract

As the defining framework standard for developing multi-tier enterprise applications, the Java™ 2 Platform Enterprise Edition (J2EE™) is enjoying huge popularity in the vendor marketplace and within a large developer community. J2EE simplifies enterprise applications by basing them on standardized, modular components.

The standalone applications computing model is shifting to a network model, where applications, devices, and services work together. Furthermore, the model is developing from a monolithic to more service-oriented one as a larger percentage of business processes now involve trading partners with whom interaction is increasingly automated.

The new Web Services model for interaction between two programs is based on emerging XML standards using ubiquitous transports such as HTTP. Each program publishes its interfaces and other capabilities (technical or non) using an XML standard that clients can recognize. Clients interact with programs using an XML-based protocol that promotes loose coupling and is ideal for programs that involve multiple parties. One consequence of loose coupling is that any entity with which a Web Service might interact may not exist when the Web Service is developed. New Web Services may be created dynamically just as new Web pages are added to the Web; therefore, Web Services should be able to discover and invoke such services without recompiling or changing any code.

Web Services is a model whose characteristics are ideal for connecting business functions across the Web, both between and within enterprises.

This paper presents a first-phase conceptual overview of the various standards emerging in this new model as well as a more detailed technical view on how Web Services can be implemented in the J2EE platform. Illustrative examples show how J2EE components such as enterprise beans and JMS destinations are exposed as Web services. A second-phase discussion provides information on how the J2EE architecture will evolve to support the more advanced Web Services concepts.

Audience

Two primary audiences can realize the benefits from developing Web services on the J2EE Platform from the information presented in this paper:

- Java developers familiar with the J2EE model and who are interested about how the Web services model fits within J2EE.
- Architects/Managers interested in a high-level technical overview of possible J2EE architecture evolution to support emerging Web services standards.

Web Services Characteristics

Exchanging XML Documents in Place of Object Interactions

J2EE applications support a network-object interaction model where objects of strictly defined types are transferred between components using a request-response interaction pattern. Cross-organizational business interactions do not fit well into this framework for several reasons, as listed below:

- Typically, object interactions are fine grained and complex, requiring an understanding of the semantics behind the interfaces and often the object graph behind the interfaces. Message- or

Developing Web Services on the J2EE Platform

document-driven interfaces are inherently more simplistic and, therefore, more appropriate for general consumption, without having a lot of prior knowledge about the target system's implementation.

- Object interactions require an understanding of the object model within which they live; that is, EJB interactions necessarily imply EJB semantics and an EJB runtime. Likewise, the same applies for COM. Consequently, these interactions are not appropriate for multi-party interoperability.
- Service interfaces often require change, and these changes cannot be captured by simple extensions, which precludes the use of object inheritance to support the necessary interoperability.
- Cross-organizational interactions must be long lived. Therefore, exchanging XML documents asynchronously is better suited for cross-organizational business transactions.

Service Oriented

Application contracts are shifting from a more object-oriented style to a more service-oriented 'pay as you use' approach, forcing application builders to consider service descriptions as much more than just application object interfaces. Now, services must expose technical and business capabilities, which may include quality of service factors such as availability guarantees, response times, etc., and various business-level service qualities such as billing options, pricing, etc.

Easy Access to Any Web Program

Sophisticated distributed computing architectures based on technologies such as CORBA, COM/DCOM, etc. are used to expose Object services. However, a significant percentage of Web applications are based on scripting languages, which do not map well to these distributed architectures. For instance, CORBA's IDL, a neutral language for defining object interfaces, has standard mappings to only a handful of languages. In addition, these technologies use transport protocols typically not allowed to cross-corporate firewalls. Web Services standards work with the most widely used Web transports such as HTTP and face fewer language mapping challenges.

Dynamic Application Building through Discovery

In the J2EE development model, typically, application designers bind software components to one another at development time. These components are usually provided in-house and are well known. Conversely, Web Services are likely to be implemented and provided by different Service Providers to easily change the services, add new or change existing capabilities, and bind to Service Providers dynamically by determining the Provider to use based on the user as well as on application context.

Prominent Web Services Standards

Various Web Services concepts and common standards such as Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL), Universal Description Discovery and Integration (UDDI), Business Transaction Protocol (BTP), and (ebXML) are explained below.

SOAP

SOAP is a lightweight, XML-based protocol for exchanging information in a decentralized, distributed environment. SOAP consists of four parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined data types, a convention for representing remote procedure calls and responses, and a binding convention for

Developing Web Services on the J2EE Platform

exchanging messages using an underlying protocol. Potentially, SOAP can be used in combination with a variety of other protocols such as HTTP, SMTP, etc.

Currently, SOAP 1.2 is a working draft specification in the W3C. The entire specification[4] is available on the W3C XMLP working group Web site.

WSDL

WSDL is an XML-based specification schema for describing a Web service's interface along with endpoint access information such as the network protocol for access, the accepted message formats, and how to reach the service.

Conceptually, a WSDL document is composed of two parts. One part provides the service interface definition, and the second part defines the service implementation definition.

WSDL defines XML grammar for describing contracts between a set of endpoints exchanging messages. A WSDL interface definition part is analogous to an IDL document or an EJB remote interface.

A typical WSDL document contains the following elements:

- Types: XML types corresponding to the various arguments and return types
- Message: Abstract typed definition of the data being exchanged
- Operation: Abstract description of a operation supported by a service
- PortType: Collection of operations supported by one or more endpoints
- Binding: Concrete protocol and data format specification for a particular port type
- Port: Single endpoint defined as a combination of a binding and network address
- Service: Collection of related endpoints

Exhibit 2 provides an example of a WSDL document. A WSDL specification developed by IBM® and a few other companies is currently a 'submission note' to the W3C. The WSDL 1.1 specification[8] is available on the W3C Web site.

UDDI

Universal Discovery Description and Integration is a specification for a distributed, Web-based information registry of Web Services. These registries are publicly accessible and currently support service type registration for software companies, individual developers, standards bodies, and business registration types for describing company-supported services.

UDDI includes the shared operation of a business registry on the Web. For the most part, programs and programmers use the UDDI Business Registry to locate information about services and, in the case of programmers, to prepare systems that are compatible with advertised Web services or to describe their own Web services for others to call. The UDDI Business Registry can be used at a business level to check whether a given partner has particular Web service interfaces, to find companies in a given industry with a given type of service, and to locate information about how a partner or intended partner has exposed a Web service in order to learn the technical details required to interact with that service.

From a J2EE developer's perspective, a UDDI registry is simply a repository containing specific and associated Web Services information (usually from a WSDL file), as in JNDI, but not the Web Service itself. J2EE developers primarily use a UDDI registry to locate services information, prepare systems for Web Services compatibility, and to describe their own Web Services.

ebXML

ebXML is a modular suite of specifications that enables enterprises of any size and in any geographical location to conduct business over the Internet. ebXML defines standards to exchange business messages, conduct trading relationships, communicate data in common terms, and define and register business processes. Currently, OASIS (www.open-oasis.org) manages ebXML.

The ebXML specifications introduce the following key concept standard mechanisms for:

- describing a business process and its associated information model.
- registering and storing business process and information meta models for sharing and reusing them.
- discovering information about each participant, including the business processes they support, the business service interfaces they offer in support of the business process, the business messages exchanged between their respective business service interfaces, and the technical configuration of the supported transport, security, and encoding protocols.
- registering the aforementioned information for discovery and retrieval.
- describing the execution of a mutually agreed upon business arrangement which can be derived from information provided by each participant from above (Collaboration Protocol Agreement – CPA).
- standardizing a business messaging service framework that enables interoperable, secure, and reliable exchange of messages between trading partners.
- configuring the respective messaging services to engage in the agreed upon business process in accordance with the constraints defined in the business arrangement.

BTP

BTP is an XML dialect for orchestrating long-running, inter-enterprise business transactions that addresses the unique business-to-business (B2B) transaction requirements. BTP is based on the multi-level transaction model, which provides the independence required by the participating resource managers. In this case, companies' business-to-business servers are engaging in business transactions.

BTP's goal is to reliably manage business transactions' propagation results (success or failure) to all the involved resources. BTP does not specify the business protocol governing the business transaction. Conversely, it simply provides facilities and semantics for a reliable termination mechanism to achieve a shared agreement on the business transaction outcome.

Alone, BTP cannot guarantee atomicity, consistency, or durability. Systems using this protocol must manage their local resources accordingly to achieve these attributes. For example, upon termination due to failure, systems must execute the appropriate compensating action.

BTP is a business-to-business protocol stack agnostic, so it can be easily implemented in conjunction with other standards such as ebXML or SOAP. For example, a header can be added to the ebXML message envelope to carry the transaction context defined by BTP. BTP system messages such as *startTransaction* or *terminateTransaction* can be sent as standard ebXML messages.

Evolution of J2EE Architecture

Today's J2EE server products provide basic Web services support such as accessing J2EE components using the SOAP 1.1 protocol. Future J2EE products eventually will support, and possibly standardize, how Web services will work that are part of complex business processes participating in business transactions spanning many parties. Most of the higher-layer Web services standards, such as business process standards, and long running transactions, are still immature but are evolving rapidly. This paper explores the evolution occurring in two distinct phases pertaining to the J2EE platform.

Developing Web Services on the J2EE Platform

For the first phase, the primary focus is on the characteristics of simple Web services for driving a candidate J2EE architecture to these services. Practical application scenarios describe how the characteristics are realized in a J2EE server. The discussion explains the developer's activities as well as the programming model and architectural concepts.

The second phase discussion introduces service publishing and discovery, service intermediaries, a Web Services security model, and workflow standards as well as how a J2EE environment realizes these concepts.

Simple Web Services Support

Important Characteristics

- J2EE applications expose EJBs and JMS destinations as Web Services.
- Exposed services use WSDL as the service description language and provide access to components using the SOAP 1.1 protocol over an HTTP transport.
- Services are published and distributed through ad-hoc means.
- Private registries (possibly based on UDDI) are used to integrate with partners by some applications.
- Typical enterprise application integration is based on the J2EE Connector architecture.
- Web Services will not change existing trading partner policy and convention agreements.

Using Web Services to Integrate Internal Assets

Today, enterprises have automated a significant percentage of their processes to reduce costs and improve process execution quality and speed. Application building has moved from monolithic development to a task of rapid integration. In place of building custom adapters to meet the new requirements and link functions together whenever a requirement is added, organizations can now expose existing application functions at an enterprise level as Web Services.

J2EE applications can access these exposed services using standardized XML protocols, allowing organizations to create a veneer over existing applications by exposing services as stateless Web Services. These Web Services are readily accessible because of the HTTP transport ubiquity that supports the essential SOAP as the XML protocol.

The facades created over these applications define the stable contract of the services. The application exchanges XML documents, and a service description document defines the document structure. This service description remains intact even if the underlying implementation changes. Using a service description provides for a more loosely coupled, resilient application that can be more easily changed in response to business requirements. Today, service descriptions are implemented using WSDL and will also likely be implemented in phase two of the Web Services evolution.

Publishing and Discovering Services

Initially, applications will be built with services discovered during design time. When application integration is enabled through Web Services, the service-finding process will be ad-hoc. For instance, Service Providers will describe the services using WSDL and distribution will be by E-mail, FTP, or simply by making it available on an internal Web site. Text documents would disclose other service semantics and capabilities.

An internal or private UDDI-based registry (see the *UDDI* section for additional information) could also be used. These private registries may allow visibility to partner organizations so that the organizations can discover and publish services. The Service Providers and these services will be well known.

Developing Web Services on the J2EE Platform

Designers and developers can leverage these registered services by browsing through the registry using tools or programmatically using the UDDI “Inquiry” API. Additionally, during design time, they can use the service descriptions and any pointers to other technical specifications to statically bind and use the service. The designers and developers would verify the service's protocol and semantics before deployment.

Applications involving integration with partner processes will evolve to use Web Services on a case-by-case basis. Partners will publish their service interfaces using WSDL, and applications consuming these services will bind statically at application build time. In this phase, service selection and invocation are still performed in an ad-hoc manner and will require preliminary agreements from a business, legal, and technical perspective between the cooperating companies.

Figure 1 provides a high-level view of the following candidate J2EE deployment process:

1. Upstream partner receives user request to run a credit check
2. Upstream partner sends request through a browser or Java™ client to the application hosting the Credit Check Web Service
 - a. If the original request is for just good/bad, the application may access local storage
 - b. If the original request is for detailed report, the application may access Vendor X service
3. Request is sent to Vendor X for detailed report (non-static information)
 - a. (Vendor X is known; discovery is not on the fly)

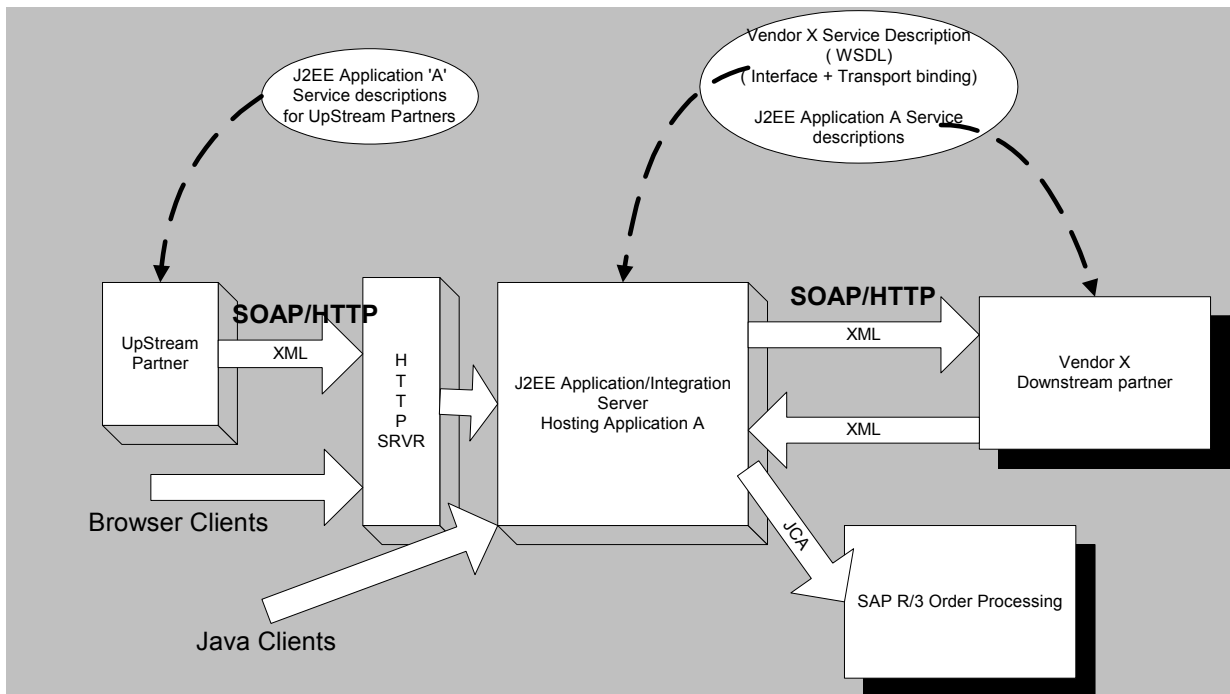


Figure 1. High-level View of a Candidate J2EE Deployment

This view extends connections to partner and internal services using WSDL for service descriptions and SOAP over HTTP for binding and transport, respectively. All the parties consume the service descriptions either statically or at design time, as illustrated by the dotted lines.

Motivating Scenario

Developing Web Services on the J2EE Platform

To exemplify developing Web Services from J2EE applications, consider a checking application as a motivating example. The sections below refer to this example.

A bank or agency provides businesses with up-to-the-instant credit standings. Before completing a business transaction, a business may need to check a potential trading partner's credit standing. In this scenario, a request would be sent to the bank's Web Service, processed, and then a returned response would indicate a credit rating of either 'Good,' 'Bad,' or 'Unknown.'

The credit checking application offers two services:

- Service 1 – Upon request, the service instantly provides a credit status for any organization using its internal up-to-the-instant credit databases.
- Service 2 – The service can return a detailed credit report for any organization given the organization's identification information; for example, Dun & BradStreet "DUNS ID." In this case, the application essentially acts as a broker and satisfies the request using third-party services. This involves aggregating information obtained from an external source, so the service response is not immediate.

Phase One Development Model

This section provides an overview of the developer/ assembler/ deployer tasks to implement the credit-check application as a Web Service.

For exemplification, we oversimplify the design and assume that the first simple credit check service is implemented as a stateless session bean. The second service, which requires the server side component to access a third-party credit checking service, is implemented using a message driven bean (MDB), and the client dispatches service requests as asynchronous "one-way messages" to a JMS queue.

A list of developer activities necessary to expose a credit checking Web Service is as follows:

Service 1

1. Develop the bean's remote interface.
2. Implement the bean class exposing one or more methods that can return a credit rating for any given organization (see Exhibit 1).
3. Use a server-provided tool to generate WSDL from the remote interface. As part of this task, the developer must supply a URI for the service as well as information for binding the service to a specific transport (see Exhibit 2).
4. Deploy the stateless session bean. The deployment engine will probably generate a Web application containing all the SOAP processing machinery.

Service 2

1. Code 'getCreditReport' functionality in an MDB.
2. The MDB accesses a remote service, hosted by Dun & BradStreet, to fetch the credit report for a given organization. Access to the remote service could be a synchronous operation (RPC) or an asynchronous one (Messaging). The sections below explore both programming models.
3. The MDB is associated with a JMS queue declaratively using an EJB descriptor.
4. Server-provided tools can be used to generate a WSDL document for the JMS destination.
5. At deployment time, the server will generate (or update) a Web application to service asynchronous messages being dispatched to the JMS destination.

The next few sections explore this model in detail for service access.

Exhibit 1

```
BE /** Bean remote interface for Service 1 */
public interface CreditRating_Service extends EJBObject {
    public String getCreditRating(String orgID ) throws RemoteException ;
}
```

Exhibit 2

```
/**** WSDL SAMPLE DOCUMENT *****/
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://www.creditratingservice.com/CreditRating"
xmlns:xsd="http://www.w3.org/1999/XMLSchema"
targetNamespace="http://www.creditratingservice.com/creditRatingService-interface">

<types>
<xsd:schema targetNamespace="http://www.creditratingservice.com/creditRatingService"
xmlns="http://www.w3.org/1999/XMLSchema">
</schema>
</types>

<message name="InOrgID">
<part name="ordID" type="xsd:string"/>
</message>
<message name="OutCreditRating">
<part name="rating" type="xsd:string"/>
</message>

<portType name="CreditRating_Service">
<operation name="getCreditRating">
<input message="InOrgID"/>
<output message="OutCreditRating"/>
</operation>
</portType>

<binding name="CreditRating_ServiceBinding" type="CreditRating_Service">
<soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
<operation name="getCreditRating">
<soap:operation soapAction=""/>
<input>
<soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:CreditRating" use="encoded"/>
</input>
<output>
<soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:CreditRating" use="encoded"/>
</output>
</operation>
</binding>
<service name="CreditRating_Service">
<documentation></documentation>
<port binding="CreditRating_ServiceBinding" name="Demo">
<soap:address location="http://localhost:7001/CreditService/CreditRatingService"/>
</port>
</service>
</definitions>
```

Service 1 -- Invoking Service Operation as a Remote Procedure Call (RPC)

Developing Web Services on the J2EE Platform

Although the SOAP protocol was fundamentally designed to be a simple and lightweight mechanism to exchange XML documents, one of the design goals for SOAP is also to accommodate an invocation model, which allows a service client to invoke a service operation as a remote procedure call. Using SOAP for RPC is orthogonal to the SOAP protocol binding. When using HTTP as the protocol binding, an RPC call maps naturally to an HTTP request, and an RPC response maps to an HTTP response. This request/response model is well suited for situations where an instantaneous response is necessary but requires that services are readily available. As previously mentioned, access to the stateless session bean typically would use the RPC invocation model.

Several approaches to providing RPC style access to Web Services from a Java client are given in the following sections.

Using a proxy class generated at design time

In this approach, the developer provides the URI to a WSDL service description document to a vendor-provided tool and generates a proxy class corresponding to the service. The proxy class exposes public methods that map to operations defined in the WSDL document. The method arguments and return types are derived using a convention that maps the ‘messages’ defined for each operation to equivalent Java types. The types used in messages follow the XSD [4] type system for maximum interoperability. The vendor’s tool most likely will support mapping most of the XSD types to Java types, but sophisticated tools could be expected to support mapping custom types used by the service to the Java type system. The client-programming model consists of simply instantiating the Class and invoking the various service methods synchronously.

Using JNDI to fetch the proxy

Alternatively, the client uses the JNDI API to construct proxies from WSDL documents at runtime. Exhibit 3 provides a code sample demonstrating the dynamic construction of the application interface from WSDL. This coding style closely follows the J2EE client-programming model and does not bind the client code to any SOAP implementation classes. Exhibit 3 illustrates the client-programming model using the BEA WebLogic Server 6.1.

Exhibit 3

```
{
    ....
    Properties h = new Properties();
    h.put(Context.INITIAL_CONTEXT_FACTORY,
        "weblogic.soap.http.SoaInitialContextFactory");
    h.put("weblogic.soap.wsdl.interface", CreditRatingService.class.getName() );
    Context context = new InitialContext(h);
    CreditRatingService svc = (CreditRatingService)context.lookup
        ("http://localhost:7001/CreditService/CreditRatingService/CreditRatingService.wsdl");
    String rating = svc.getCreditRating(orgID);
}
```

Dynamic Invocation

A third programming style, shown in Exhibit 4, demonstrates how an application, without prior knowledge of either the service description or the Java interface, can dynamically construct SOAP calls and invoke these on the target service.

Exhibit 4

```
/** Invoking Web Service operations using the Apache SOAP implementation
**/
// Build the call.
Call call = new Call();

call.setTargetObjectURI("urn:CreditRatingService");
call.setMethodName("getCreditRating");

Vector params = new Vector();
params.addElement(new Parameter("OrganizationID", String.class,
                                org_id, null));
call.setParams(params);

// Invoke the call.
Response resp;
try
{
    resp = call.invoke(url, "");
    // A SOAP message encapsulating the request will be sent and
    // SOAP response received will be parsed to return a Response object
}
catch (SOAPException e) // Something went wrong in the SOAP processing
{
    System.err.println("Caught SOAPException (" +
                       e.getFaultCode() + "): " +
                       e.getMessage());

    return;
}
// Check the response.
if (!resp.generatedFault())
{
    Parameter ret = resp.getReturnValue();
    // Get the credit rating
    String rating = (String)ret.getValue();
}
}
```

Irrespective of the API used to fetch the service proxy and invoke the service operations, the proxy or some delegated object is responsible for marshalling the arguments to XML fragments using the SOAP encoding rules (or some other custom encoding format), construct the SOAP document, and transmit the same using a configured transport.

Figure 2 illustrates how a J2EE server typically processes a SOAP RPC request.

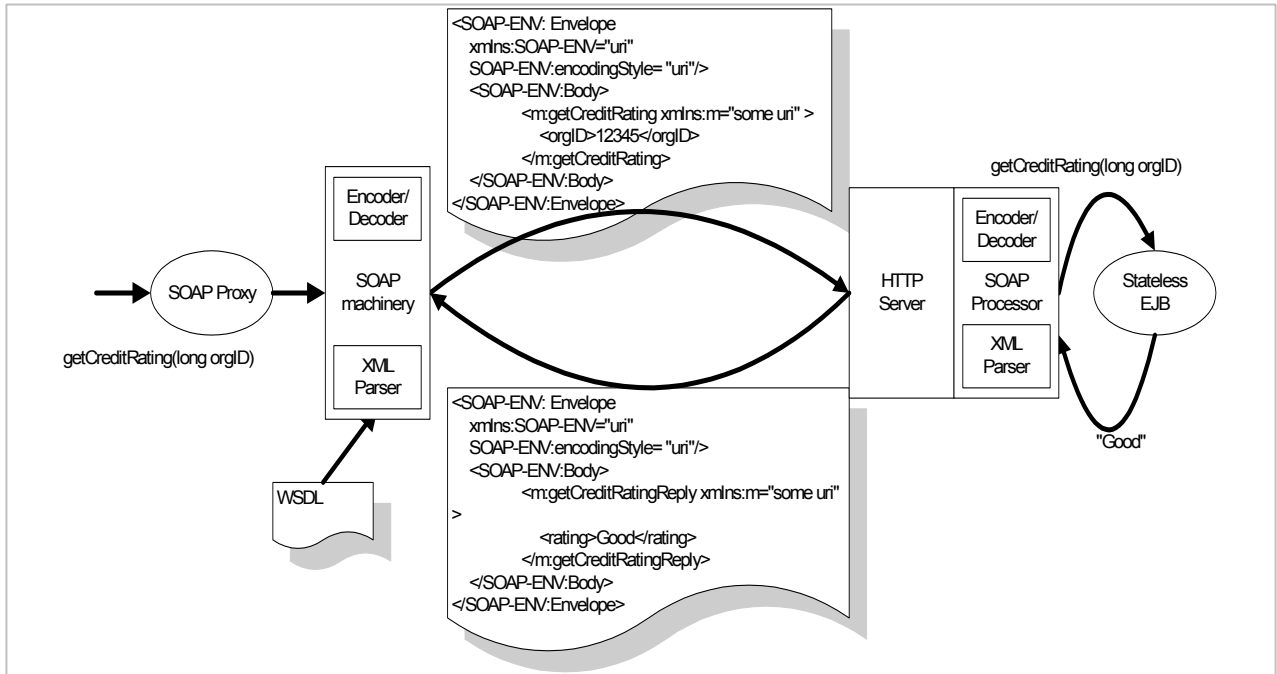


Figure 2. How a J2EE Server Processes a SOAP RPC Request

The transport infrastructure delivers the message to the SOAP processor on the J2EE server (typically, a Web application). The processor maps the request to a stateless EJB, converts the message arguments to Java types using the SOAP decoding rules (or custom decoders provided by the application), and invokes the EJB. The SOAP processor formulates a SOAP message response using the returned result and relays back to the client as an HTTP response.

Service 2 -- Invoking Service Operation Using One-way Message Exchanges

The RPC model lends itself to being a fine-grained interaction model but is not necessarily suitable in business conversations involving multiple parties. Service communications should follow a document exchange model that promotes large-grained interactions well suited for business-to-business scenarios rather than fine-grained method invocation.

For instance, a third-party service accepting orders may accept a single order document containing all the relevant customer, order, product, billing, and shipment information rather than obtain all this information through multiple exchanges.

Developing Web Services on the J2EE Platform

When used with HTTP, SOAP defines a header extension for identifying one-way messages. When a message is sent one-way to a destination, the client does not receive a response immediately nor does it receive a response on the same HTTP connection. The client's Service Provider has the option to deliver the message to the intended target either immediately or asynchronously depending on the availability of the target service. Similarly, the target Service Provider has the option to process the message either immediately or later based on application criteria.

The semantics are those of a messaging system involving functions such as reliable message delivery, message sequencing, correlation, etc. As an XML envelope standard, SOAP does not define any of the advanced messaging semantics. However, standards such as the ebXML messaging specification and the BizTalk Protocol are examples of standards that define the structure of XML messages through SOAP header extensions to support advanced messaging. Recent efforts and announcements from the various standards bodies indicate that there could be convergence to a single, widely-adopted standard for XML messaging.

The J2EE programming model for sending or receiving one-way message documents could depend on the use of proxies (Stubs). This approach is similar to the one previously described for RPC or could also be based on the use of specific messaging APIs such as JMS or JAXM (see the *JAXM* section in *Appendix A, Java APIs for Web Services*, for more information).

Figure 3 illustrates the processing for a service client to send an XML document one way using a protocol such as ebXML and to receive a response as an XML document asynchronously.

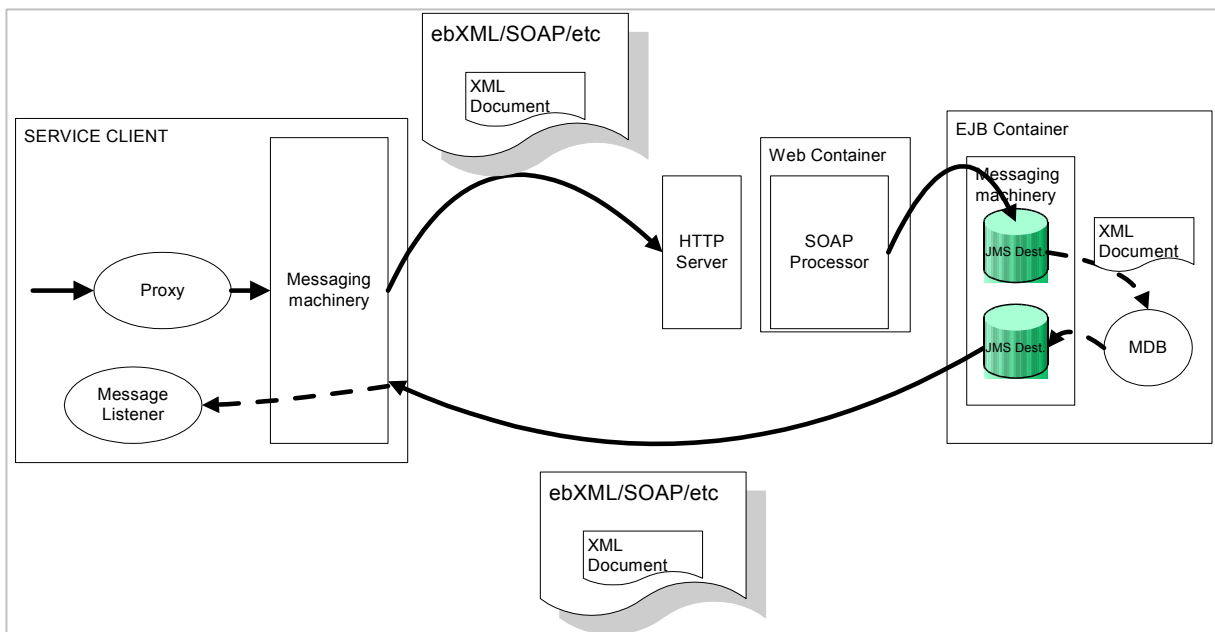


Figure 3. Service Client Process for Sending a Document and Receiving a Response

JMS supports 'point-2-point' messaging using Queues and 'publish-subscribe' style of messaging using JMS topics. JMS implementations could provide an extension of the JMS TextMessage type as XMLMessage type so that XML SOAP messages can be constructed as first-class objects. The application would follow the JMS API and send messages to JMS destinations. The underlying JMS implementation would be responsible for forwarding the messages to the actual destination identified by a URI. The association of the JMS destination to a SOAP endpoint would typically be done declaratively at deployment time. Alternatively, the application can use the JAXM API (see the *JAXM* section in *Appendix*

Developing Web Services on the J2EE Platform

A, *Java APIs for Web Services*, for information) to do XML messaging. JAXM is expected to define an API to construct multipart SOAP multipart messages and send these to messaging endpoints.

Sending messages may involve invoking Stub methods. Using the SOAP encoding rules for the message body, the Stub would marshal the passed argument to XML. The message payload (body) could be wrapped as either an ebXML message or a BizTalk message (the messaging protocol in use); then the message payload would be transmitted to the endpoint. This method immediately returns a void.

The SOAP processor on the receiving end unpacks the ebXML message and dispatches the payload to a JMS destination. The message payload could be transformed to an alternate XML document using an XSLT style sheet or possibly converted to a Java object before it is delivered to the JMS destination. The destination could drive MDBs in the EJB container to process the payload. The enterprise beans may want to respond to the initiator by formulating a reply and sending the reply to a JMS queue. Also, the messaging system could be configured to forward the messages to the originating client.

Server Side Processing of SOAP Messages

The SOAP processor is the key piece of the server side architecture and is responsible for the following functions:

- **Parsing** - The SOAP processor must parse the incoming XML document, ensure the request is well formed, and possibly validate the request.
- **Binding XML payload to Java** – Binding unmarshals the message contents and (optionally) binds the XML contents to a Java object or to a JMS Object message.
- Invoke J2EE components:
 - If the message was an RPC request, the processor must identify the EJB, method, and invocation arguments before invoking the enterprise bean.
 - If the payload contains a one-way message, the target JMS destination is identified and the message is delivered, along with any attachments, to the JMS destination.
- **Engage a security manager to possibly authenticate the sender.** This could involve a third-party authentication service. Web Services security architecture is explained in the Security section.
- **Exception Handling** - In case any exceptions occur during processing, the Java Exception is transformed to a SOAP fault and delivered to the service client.

XML Transformation

Documents exchanged using the asynchronous messaging model typically consist of many XML fragments along with non-XML parts in the message. To allow the J2EE component that processes the message to work with Java objects, the container should provide machinery to transform the XML contents to one of the following:

- **A Java Serializable** - The mapping of an XML schema to a Java class structure could be specified at deployment time. The mapped meta-data could be used by the container to automatically bind the message payload to an object graph.
- **An alternate XML structure using a transformation engine** - In a business-to-business interaction setting, delivered messages may be based on a convention determined by a standards organization or industry taxonomies/ontologies. It may be essential to transform the message payload to an alternate XML structure within the enterprise.
- **Parsed DOM document** - The message could be parsed to a DOM tree using a pluggable XML parser.

A new Java standard, Java API for XML Data Binding (JAXB), provides an API to bind XML documents to Java objects. The mechanism uses an XML schema (only DTD) compiler that binds an XML schema to a set of Java (derived) classes at development time. The JAXB provider also provides a binding framework, a runtime API that, in conjunction with the derived classes, supports unmarshalling,

marshalling, and validating XML documents against the constraints expressed in the schema. Currently, JAXB is under development.

Message Listeners

As messages are delivered to JMS destinations, the message consumption model does not change with Web Services. Message listeners could be MDBs or JMS message consumers. Messages consumed by MDBs could be forwarded to external enterprise information systems through JCA Connectors.

J2EE Applications using Advanced Web Services

As evolution Web Services technologies gains acceptance, we will see Web Services enabling Enterprise Application Integration (EAI). The J2EE architecture will evolve to support B2B applications that require integration with heterogeneous applications either within or outside the enterprise.

The next generation of applications evolves from applications that statically bind to well-known services to applications that are discovered dynamically and perhaps Just-in-time (JIT). The service that actually gets used is determined not only by the service interface but also by a variety of other Quality Of Service (QoS) factors such as price, availability, security, transaction support, etc. Some important aspects that distinguish this next phase of Web Services are explained below.

Service Registries

Service registries allow businesses to register relevant information about their Web Services for easier access by other businesses. Service descriptions explain what the service offers as well as how to access the service. Service registries are based on standards such as UDDI. Although UDDI does not form a full-featured discovery service, the specifications define (and target) a way to publish and discover pertinent service information.

With UDDI-defined facilities, a program or programmer can locate information about services exposed by a partner, find whether a partner has a service that is compatible with in-house technologies, and follow links to the specifications for a particular Web service so that an integration layer compatible with a partner's service can be constructed. J2EE deployment tools could work with UDDI registries to automate the registration process through deployment descriptors.

Service Brokers and Intermediaries

With the proliferation of services discoverable at runtime, application building becomes a task of describing the capabilities needed, finding the right set of services that meet the requirements, and orchestrating the various services found in different ways for different business use cases. Essentially, Service Providers are brokerages, where aggregating services from other Providers fulfill their service and respective contracts. Clients will have a trading agreement with brokers that dictates a certain quality expectation from the brokers. In turn, the brokers guarantee the quality but are free to choose the underlying set of Providers either dynamically or Just-in-time.

For example, in the credit check scenario, a credit check service from a bank is being provided to a broker. In turn, this broker can combine the credit check service with its own stock quote check service and repackage both into a new financial service under its own "brand" that it provides to consumers.

The notion of 'intermediaries' is similar to that of brokers. However, it differs in that the service client converses with the Provider by passing the message through one or more of its intermediate service nodes.

Developing Web Services on the J2EE Platform

These intermediate nodes provide value-added services on top of, or in conjunction with, the actual services such as reliable messaging, auditing, security, etc.

Impact on J2EE Architecture

Two important J2EE applications development changes have occurred.

First, application building moves to a more service-oriented model, where application functions are realized through one or more services. These services are provided by the composition or aggregation of other services.

Second, service discovery and binding are more dynamic. Service clients (J2EE components) connect to service registries securely, query for services that meet certain constraints, and then bind to them. It is important to note that most dynamic connections will be trusted only to private registries, where the dynamic choice of a Service Provider is always a legitimate one.

It is important to note that in the figure below, Application A and Services X and Y could be different services with each request. The criteria will determine the provisioning of these services and may vary from one request to another.

Figure 5 illustrates the following process for using a service registry by J2EE applications:

1. An upstream Service Provider Application, upon receiving a request to check credit, accesses the public registry to locate an appropriate Provider that meets certain criteria.
2. Upon locating an appropriate Provider (Application A), the check credit request is forwarded directly to application A's location.
3. Application A (composite of more than one service – refer to the key) also accesses the public registry to locate services X and Y based on its criteria.
4. Upon locating service X and Y, application A binds and sends the request directly to said services and then receives responses directly from those services.
5. Application A forwards the responses back to the upstream Service Provider who in turn send it to the original requestor.

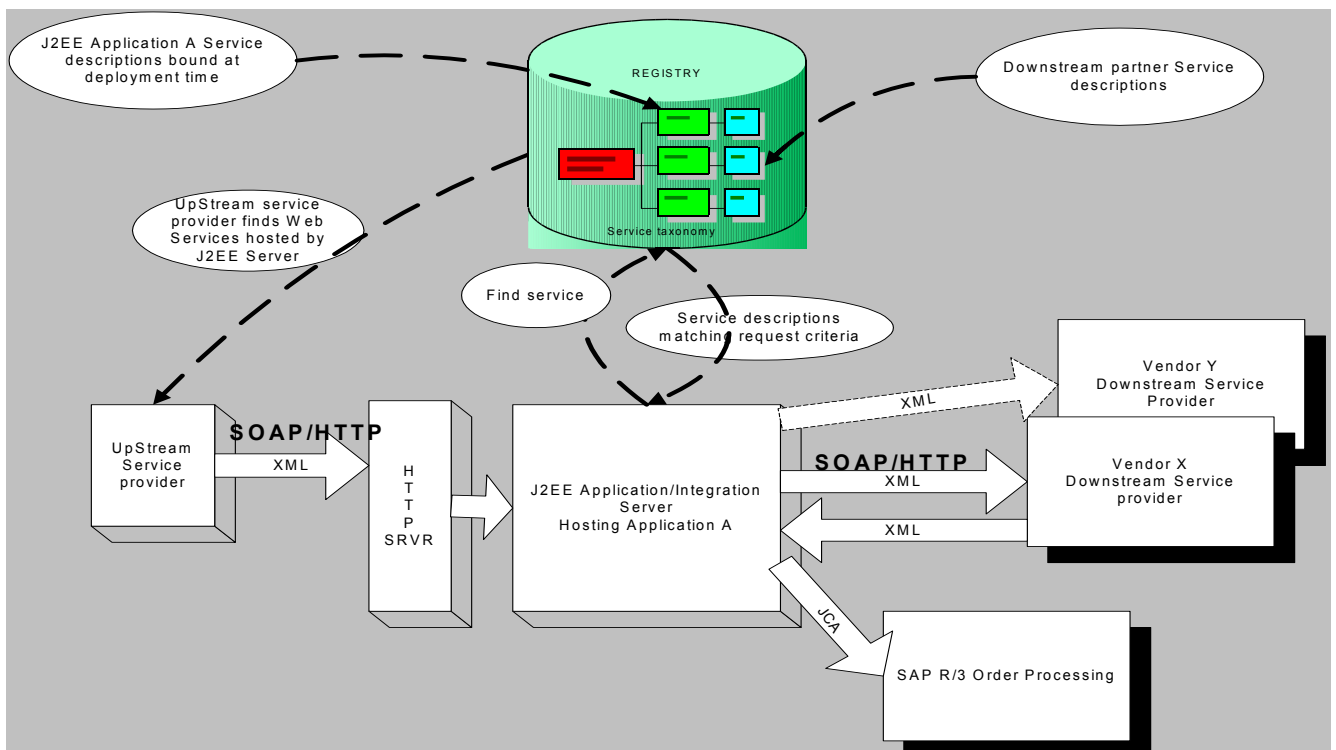


Figure 5. Phase 2 Architecture

Quality of Service (QoS) Support

In the first phase, services are described using WSDL, which primarily specifies only operations, endpoints, and the operations it uses once a service binds to another. In phase two, this service description expands to specify other behavioral and capability aspects of the service such as the supported security protocols, responsiveness (immediate or asynchronous), support for transactions, messaging protocol expectations for individual operations, etc. Service descriptions can also include documents describing business level capabilities such as the pricing model.

Today, service interface definitions are generated from J2EE remote interface definitions. Eventually, the deployment model in J2EE may be expanded to accommodate specifying technical capabilities associated with the service provider. For instance, the supported messaging protocols, message delivery and reliability semantics, security model supported, etc could all be specified through deployment descriptors.

Web Services Security Model

Given the nature of service-oriented applications where a single conversation may potentially involve multiple parties using different security-related technologies, security is an especially complex topic. Most of the XML-based security standards are still evolving.

Initial Web Services implementations used by clients within an organization can rely on the security model supported with HTTP. For instance, the HTTP basic authentication suffices for services that are protected by firewalls and where securing on the wire is not critical. The password is encoded using Base64 encoding for some level of protection on the wire. HTTPS can also be used when the Service Providers require more privacy. HTTPS use an SSL implementation that the service client can use and is supported today with the Web Services-enabled application servers such as the BEA WebLogic Server 6.1.

The SOAP protocol over an HTTP transport will be the most common XML interchange mechanism. With HTTP, we get the security models defined by the standard, viz, the Basic/Digest HTTP authentication, and HTTPS.

With complex Web Services, conversations span multiple services that potentially have been discovered dynamically. Therefore, Web Services provides an end-to-end security model for the entire conversation when highly sensitive data is passed from service to service. These requirements are not met with just HTTP or HTTPS.

Some unique characteristics for complex services addressed by Web Services are as follows:

- The user has a single sign-on facility irrespective of any services involved in a single conversation.
- Sophisticated security models such as those based on Primary Key Infrastructure (PKI) and digital signatures are available for non-repudiation purposes.
- A manageable authorization model enables each Service Provider to grant access to various service functions for each authenticated individual.

To provide single sign-on, a user is identified once and shares the identity among all the participants of the conversation. Given that organizations guard security principals closely and that each organization identifies clients in a proprietary way, this challenge is enormous. The security principal is constantly mapped to a different client depending on the security domain to which the request is directed.

Developing Web Services on the J2EE Platform

Working with a common third-party authentication/Trust service is one way to solve this problem. Organizations can use third-party Trust services to delegate all or part of the Trust management functions, as explained below.

All Service Providers engaging in a conversation rely on a COMMON Trust service to identify individuals through some token form. Message exchange involves the propagation of these tokens. Service Providers use the Trust management service to authenticate requestors. Service Providers publish all the approved Trust services in their service descriptions. Clients can use service descriptions to determine the common Trust Service Provider they want to enable the conversation. Microsoft® Passport is an example of external token-based authentication service.

Trust services can also manage private-public key pairs for services that want to rely on digital signatures. The entire process of registering, fetching, and revocating keys can be delegated to the Trust service. XML Key Management Specification (XKMS) specifies an XML protocol to access the key management functions of a Trust service. Trust service functions are expanded to include all the features necessary to digitally sign and seal documents in the future. Trust service management standards are still evolving. See [x], [y] <<verify references to x & y>> for more information.

Enterprise servers such as the BEA WebLogic Server 6.1 bundle most of the key management functions as part of the server. Services hosted on such a server fetch keys from a local key store and use these keys to sign a document using the syntax specified by the XML DigSig specification [z].

Security Assertions Markup Language (SAML), an OASIS initiative, provides a standard way to profile information in XML documents and to define user authentication, authorizations, and entitlements. SAML implementations provide an interoperable XML-based security solution, where user information and corresponding authorization information can be exchanged by collaborating services irrespective of their existing security implementations. SAML is the key to enabling single sign-on in Web Services. In user-driven transactions for the business-to-consumer market, this standard enables users to travel across sites with their entitlements so that companies and partners in a trusted relationship can deliver single sign-on across sites. Currently, SAML is under development.

Although the details of how this security model works in the J2EE environment is beyond the scope of this document, it is worth mentioning that the XML security standards allow security context to be propagated across entities and directly, and seamlessly, mapped to the J2EE JAAS security architecture. For instance, the digital signature from a signed document can be used to construct, and supply the right credentials in, a JAAS Subject instance in the J2EE server.

J2EE Application Integration with Web Services

Enterprise Application Integration (EAI) was the first era of application integration during which enterprises concentrated on leveraging their existing information technology assets to deploy best-of-breed solutions throughout the enterprise. The growth of EAI was fueled by the adoption of packaged applications such as SAP R/3, PeopleSoft, Siebel, etc. The first generation of EAI solutions has been largely proprietary and based on point-to-point integration, leading to a complex network of one-off integrations.

Today, we are transitioning from the first generation of application integration, primarily aimed at solving single-enterprise problems, to the next generation, focused on integrating applications both inside or outside the enterprise. In addition, standards such as the J2EE Connector Architecture (JCA) are transforming the way enterprises are being integrated.

The J2EE Connector Architecture defines a standard architecture for connecting the J2EE platform to heterogeneous Enterprise Information System (EISs). The J2EE Connector architecture enables an Integration server to provide a standard resource adapter for its EIS. The resource adapter plugs into an

Developing Web Services on the J2EE Platform

application server, providing connectivity between the EIS, the application server, and the enterprise application. An application server and an EIS collaborate to keep all system-level mechanisms such as transactions, security, and connection management transparent from the application components

The J2EE Connector architecture also defines a Common Client Interface (CCI) for EIS access. The CCI defines a standard client API for application components and, using a common client API, enables application components and EAI frameworks to drive interactions across heterogeneous EISs. However, the CCI does not define how the resource adapter communicates with the enterprise application. The protocol and the information exchange are defined in a vendor-specific way.

BEA believes future enterprise applications will begin exposing their functions as Web Services, allowing access to the services using a SOAP-based XML protocol. Designing a resource adapter to interoperate with heterogeneous enterprise resources will be simplified. Enabling bi-directional communication resource adapters will also expose the application's services as Web Services. These services could be used to configure the adapter or listeners that relay enterprise application-generated events to J2EE applications or to other resources connected to the integration server.

Figure 4 illustrates the JCA-based architecture.

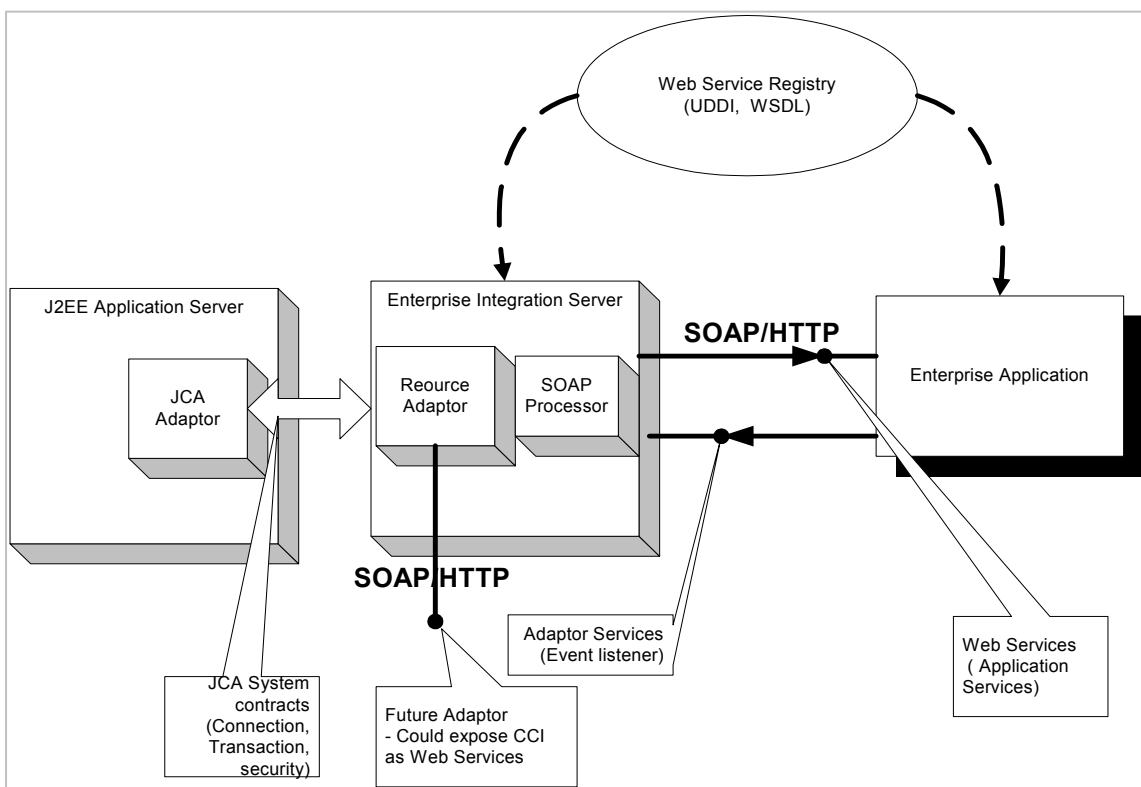


Figure 4. JCA-based Architecture

The adapters will discover the services offered by the enterprise applications using private service registries either statically, at design time, or dynamically.

Developing Web Services on the J2EE Platform

With the maturity of the Web Services, standards will be developed that define how Web Services participate in sessions and in global transactions that may extend beyond the enterprise and define a security model. This will allow defining the JCA system contracts as Web Services. Future JCA adapters could expose all interfaces as Web Services, as shown in the previous figure.

The BEA WebLogic Application Server allows enterprises to create and expose simple Web Services that are point-to-point and non-transactional. The BEA WebLogic Integration supports the creation of more complex Web Services that are multi-party, transactional, involve collaboration and workflow, and provide a higher level of security that is typical for business transactions both within and outside the enterprise.

As part of the BEA E-Business Platform, the BEA WebLogic Integration is a single solution that delivers application server, application integration, business process management, and B2B integration functionality. The BEA WebLogic Integration provides the framework for creating J2EE CA-based adapters using XML schemas as a set of services, adding additional support for XML, metadata, and bi-directional adapters. These adapters are abstracted by defining application views using XML schemas as a set of services, allowing for a common interaction protocol across a variety of applications.

WorkFlow and Modeling Business Processes

Standards such as the Business Process Markup Language (BPML) and ebXML Business process specification define the syntax and conventions for modeling business process flows in terms of XML documents. A typical business process is triggered by a business event and uses the services of one or more Web Services (possibly multi party in a business-to-business setting) to perform a set of tasks. The flow uses contextual data to determine the services exercised at runtime.

BEA expects Service Providers to define the business processes in which individual services participate and register documents containing process descriptions as part of the service description in registries. This will enable service clients to discover the service flows in which they can participate. Based on emerging Web Services standards, support is available for WorkFlow engines to function on the J2EE platform.

The BEA WebLogic Process Integrator (WLPI) is an example of a powerful workflow engine that automates workflow, business-to-business processes, and enterprise application assembly. This type of product built over the J2EE platform leverages the core Web Services offered by the platform. The BEA WLPI model allows parts or complete business processes to be modeled as Web Services.

Business Transactions

Typically, complex Web Services involve conversations spanning multiple parties. In these business-to-business scenarios, the conventional ACID-based transaction model is not suitable. The following characteristics of business transactions make the complex Web Services unique, because they are:

- long running
- capable of involving more than two parties (companies) and multiple resources operating independently by each party such as mainframe applications and ERP systems
- based on a formal trading partner agreement such as RosettaNet PIPs or ebXML Collaboration Protocol Agreements

Business Transaction Protocol (BTP) is a proposal from BEA Systems presented to OASIS to standardize the semantics of business transactions. The specification defines a multi-level transaction model that works with traditional transaction coordinators. A J2EE transaction coordinator (JTA/JTS based) could participate as a subordinate coordinator in a long running transaction. The primary difference is introducing the notion

Developing Web Services on the J2EE Platform

of a compensating transaction that is used by each participating resource manager to compensate for the changes done during the transactions if the outcome is a rollback.

BEA expects future J2EE servers to work with one or more of these business transaction protocols.

Conclusion

Exchanging information in an inter-application manner is moving to a more disconnected or loosely coupled and resilient model, where instead of using custom adapters and bridges to exchange data in proprietary formats, XML documents are exchanged, enabling organizations more flexibility by exposing services as stateless Web Services. New standards such as the J2EE Connector Architecture (JCA) are transforming the way enterprises are being integrated and are defining the exchange protocol in use.

Application building is moving from a monolithic development style to a more service-oriented 'pay as you use' style of approach. This model is forcing application builders to consider service descriptions to be something much more than just application object interfaces. Now, services must expose technical and business capabilities, and they must define the Quality of Services and Service Level Agreements (SLAs).

The J2EE architecture defines a standard for building applications as components. Web services naturally extend this model to allow multi-party interactions containing J2EE components with participating heterogeneous technologies to occur seamlessly. Future J2EE products eventually will support, and possibly standardize, how Web services will work that are part of complex business processes participating in multiple business transactions.

Appendix A, Java APIs for Web Services

While vendors offer custom programming model solutions to access Web Services, significant progress is being made to standardize the APIs for Web Services. This section provides an overview of the important Java specifications for Web Services.

Java API for XML Messaging (JAXM [4])

This specification describes Java APIs designed specifically for exchanging business documents (XML documents plus other arbitrary data). The JAXM implementation is responsible for packaging the document using an XML messaging protocol such as SOAP 1.1 with attachments, ebXML/Transport, Routing and Packaging (TRP), RossettaNet, or BizTalk and for delivering the document to the destination. JAXM starts by defining a low-level Java API for constructing and routing SOAP 1.1 messages then proceeds to define APIs for emerging messaging standards such as ebXML/TRP. The higher-level messaging definitions use the interfaces/classes that define a SOAP message. To a certain extent, the J2EE platform specification already includes JMS as the messaging standard. JAXM APIs merging with JMS remains uncertain.

Java APIs for XML based RPC (JAX-RPC [5])

The goal of this JSR is to develop APIs and conventions for supporting XML-based RPC protocols in the Java platform, addressing three primary requirements:

- APIs for marshaling and unmarshaling arguments and for transmitting and receiving calls. These APIs should permit the development of portable ‘stubs’ and ‘skeletons.’ (A stub is a piece of code that runs on a client computer and maps a language level call into a network call. A skeleton is an analogous piece of code that runs on a server and maps an incoming network call to a language-level call on the server.)
- APIs and conventions for mapping XML-based RPC call definitions into Java interfaces, classes, and methods. The purpose of this ‘forward mapping’ is to allow XML-based RPC interfaces defined in other languages to be mapped into Java. Mapping all XML-based RPC call definitions into Java is highly desirable.
- APIs and conventions for mapping Java classes, interfaces, and methods into XML-based RPC call definitions. The purpose of this ‘reverse mapping’ is to allow programmers to define APIs in Java, and then map them into XML-based RPC. Some constraints may exist regarding which Java methods can be mapped into XML-based RPC.

Java APIs for XML Registries (JAXR [6])

This specification describes Java APIs designed specifically for an open and interoperable set of registry services that enable sharing information between interested parties. The shared information is maintained as objects in a compliant registry. All access to registry content is exposed via the interfaces defined for the registry services.

Using UDDI as the predominant model for XML registries is becoming the preferred industry standard method. JAXR will provide a uniform and standard API for accessing information from these registries within the Java platform.

Implementing Enterprise Web Services – JSR 109

This specification essentially defines how Web Services are implemented in the J2EE platform, including the initial programming model and runtime architecture for implementing Web Services in J2EE. The specification provides a programming and runtime model based on the JAXM, JAXR, and JAX-RPC.

Figure 6 illustrates this relationship between the various Java APIs

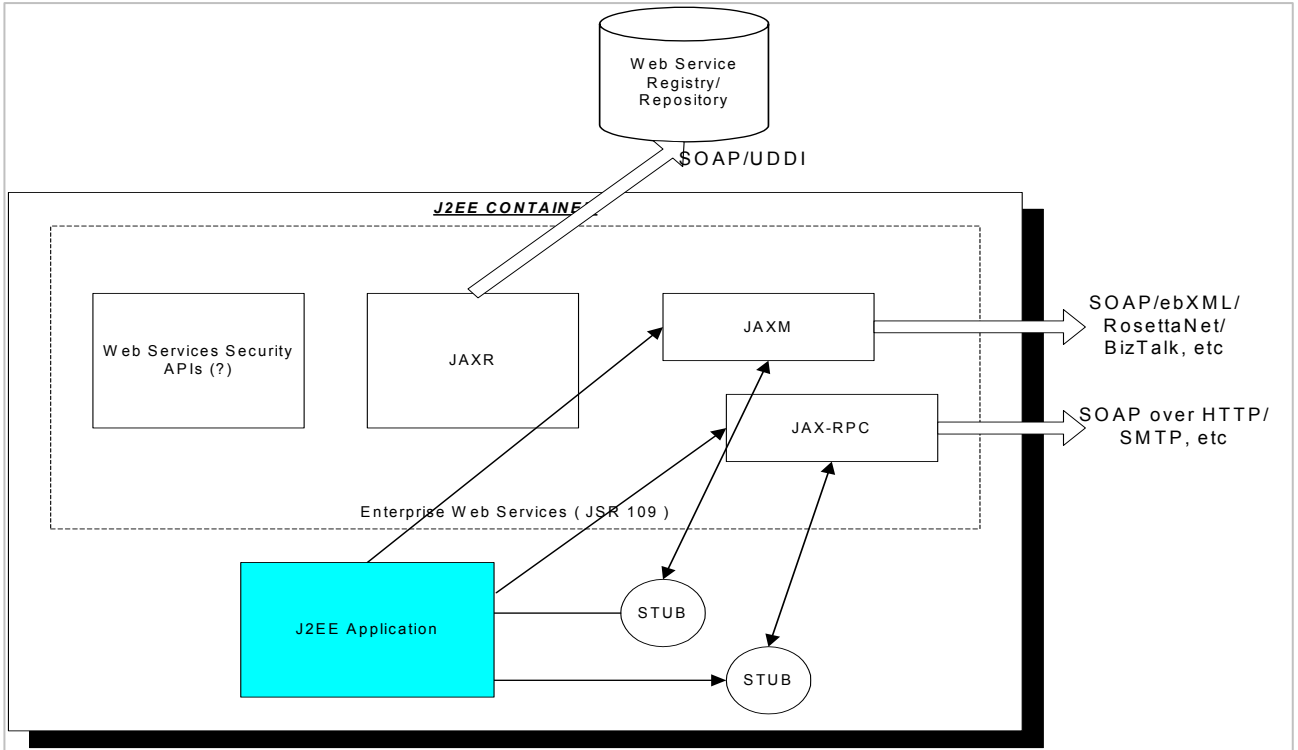


Figure 6. Java API Relationships

Appendix B, Bibliography

Govindarajan, Kannan and Banerji, Arindam. *HP Web Services Architecture Overview*. Submitted to the W3C workshop on Web Services.

Karp, Alan H., and Smathers, Kevin. Hewlett-Packard Laboratories, Palo Alto, California. *Advertising and Discovering Business Services*. Submission to the W3C workshop on Web Services.

EbXML documentation.
(<http://www.ebxml.org/documents/documents.htm>.)

Simple Object Access Protocol (SOAP) 1.2.
(<http://www.w3.org/TR/2001/WD-soap12-20010709/>)

SOAP Security Extensions: Digital Signatures.
(<http://www.w3.org/TR/SOAP-dsig/>)

UDDI Technical White Paper.
(<http://www.uddi.org/>)

Web Services Architecture Overview: IBM Developer Works. Architecture library.

Web Services Description Language (WSDL).
(<http://www.w3.org/TR/wsdl/>)

WebLogic Application Integrator White Paper.

XML Key Management Specification.
(<http://www.w3.org/TR/xkms/>)

XML Schema Part 0, 1, 2.

(<http://java.sun.com/xml/webservices.pdf>) Describes the various Java APIs for Web Services.

1. (<http://www.entrust.com/trustwebservices/>)