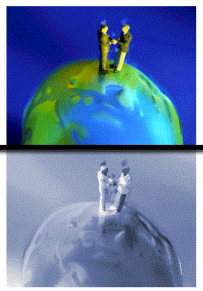

EJB, J2EE, and XML Web Services Expertise

Developer's Guide to Building XML-based Web Services

with the Java 2 Platform, Enterprise Edition (J2EE)

By James Kao
June 2001

Prepared for Sun Microsystems, Inc.

Table of Contents

I. Executive Summary

Web services using XML standards is a new paradigm in the way B2B collaborations are modeled. It provides a conceptual and architectural foundation which can be implemented using a variety of platforms and products. Today, developers can use the Java 2 Enterprise Edition (J2EE) to build XML-based web services. They can leverage existing J2EE technologies to build a complete and fully interoperable web service that complies with XML standards. Without radical reengineering, and without rebuilding a proven J2EE system, developers can construct complex and powerful web services applications.

II. Introduction

A web service is an application that accepts requests from other systems across the Internet or an Intranet, mediated by lightweight, vendor-neutral communications technologies. These communications technologies allow any network-enabled systems to interact. As technologies mature, a web service will encompass additional special functionality geared towards performing multiparty B2B collaboration.

Web services are evolving and beginning to operate in an extremely intelligent and dynamic way. These *smart* web services will understand the context of each request and produce dynamic results based on each specific situation. The services will adapt their processes based on the user's identity, preferences, location, and reason for the request. Multiple services will be combined on the fly, collaborating to produce a unique, customized solution. The mechanics of this collaboration will be completely transparent to the consumer, who will experience only the collective benefit delivered by the end result.

The XML standards which a web services system is built upon allows for an implementation-neutral approach to performing business collaborations. There are many possible implementations developers can use, including a variety of products, platforms, and standards. By using a standards-based approach, developers can build a system that provides maximum interoperability for their web services.

This white paper describes the portable Java and XML technology approach for implementing a web services architecture. It explains each of the key web services technologies and how they fit together. You will gain a better understanding of the concepts that underlie a XML web services architecture, and how they fit together with J2EE.

We begin with a 30,000-foot birds-eye view of how to build web services using J2EE. This section will give you a high-level understanding of the building blocks of a web services system. We will elaborate on each functional area later in this white paper.

III. Overview

Traditionally, there have been many barriers to two or more businesses collaborating in electronic transactions. Widely disparate systems, security issues, and incompatible data formats have made large-scale B2B integration the sole domain of large businesses and their large partners. Web services will change the field of play, and allow collaboration to occur between businesses of all sizes, significantly reducing the development and maintenance costs of building business webs.

There are three major challenges in building a web service that participates in a business web:

1. **Build client-tier connectivity** to allow applets, applications, business partners, web browsers, and PDAs connect and make use of a web service.
2. **Implement the web service** including any workflow logic, data transformation logic, business logic, and data access logic. This is the functionality behind the web service that performs work on behalf of the clients.
3. **Connect to back-end systems** which may include one or more databases, existing enterprise information systems, business partners that publish their own web services, and a shared context repository for user information shared across many systems.

You can achieve these three goals in building web services by using the Java 2 Platform, Enterprise Edition (J2EE). While J2EE has historically been used to build traditional packaged applications, user-interface driven deployments, and other enterprise-class systems, it is also a viable web services platform. The web services development model with J2EE relies on the following two standard technologies:

> **XML technologies.** The use of XML standards is very important in the overall scheme of the web services universe. XML is a data format that represents data in a serialized form that can be transported over the network from one endpoint to another. These various XML standards are primarily wire-level protocols (with a few exceptions) along with specified processes designed to support a particular semantic behavior.
>
> **Java technologies.** Developers use J2EE APIs to author business and presentation logic, access XML documents, and perform XML operations. Reliance on proven Java technology is important because it allows developers to leverage existing infrastructure to achieve a whole new level of functionality. Developers continue to embrace the J2EE paradigm of using standard APIs with many possible implementations to create systems built from best-of-breed components. Today, developers have the APIs necessary to build a web service using J2EE. A critical component of this is the Java API for XML Parsing (JAXP) which we will describe later. The future will bring a few new JAX* APIs, mainly for dealing with the XML data formats and services. These future JAX* API's will allow for greater ease and speed in development.

Figure 1 depicts an architectural overview of the heart of a web services system based on J2EE. Note that many APIs are not shown in this diagram, such as those used for parsing and messaging. However, standards, protocols, and major subsystems in a web services deployment based on J2EE are depicted.

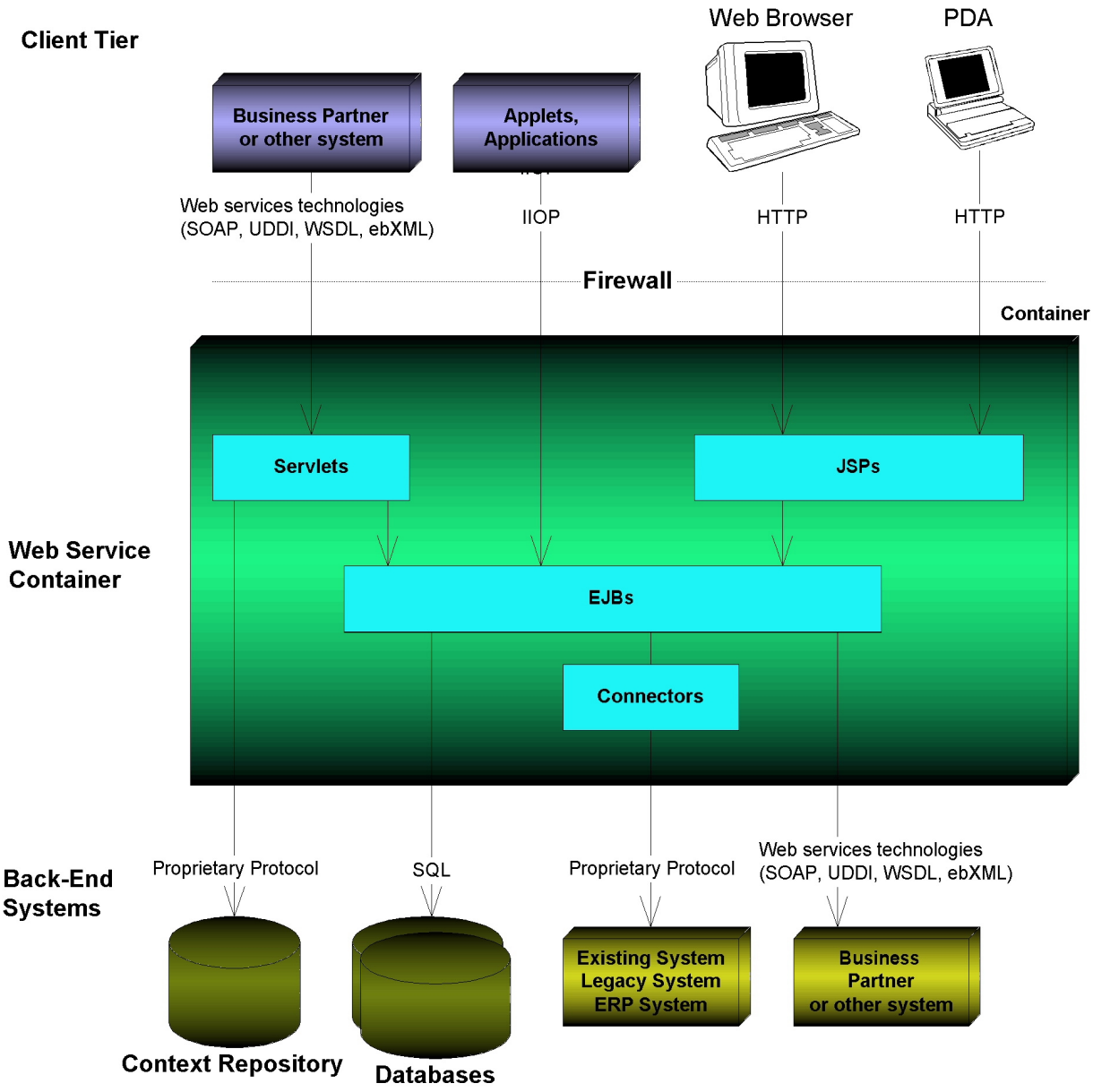

Figure 1. Major subsystems and protocols in a J2EE-based web services environment.

Let's now investigate in greater detail how we can build J2EE web services to meet eBusiness challenges.

IV. Client Tier Connectivity

Client Tier Connectivity refers to how consumers of web services access your system. Table 1 shows the three major types of clients that can connect to a web service.

Type of client	Examples	How this client connects
Business Partners	Distributors, resellers, large customers	XML-based web services technologies (SOAP, UDDI, WSDL, ebXML)
Thin Clients	Web browsers, PDAs, wireless devices	Lightweight protocol (HTTP)
Thick Clients	Applets, applications, existing systems	Heavyweight protocol (IIOP)

Table 1 Types of clients connecting to a web service.

Business Partner Connectivity

The first type of client that could access a particular web service is a business partner. Business partners could be using a variety of programming languages, middleware, and hardware. So when a business partner calls your system, the web service request arrives in the form of an XML document. XML is a standard meta-markup language for business data and allows heterogeneous systems to communicate.

Java Servlets

When a business partner issues a request to a web service, the recipient of the XML document is a Java servlet. A servlet is a request/response-based Java object that runs within the managed container environment. It can respond to requests using any protocol, such as HTTP, FTP, or POP. In this case, servlets are used to respond to HTTP requests, since web service requests utilize the HTTP protocol to enable firewall navigation.

When a request comes into a J2EE web service deployment, the following order of operations ensue, as shown in Figure 2.

1. The XML document is received by a Java servlet.
2. The servlet processes the incoming XML-based request.
3. The servlet then calls one or more Enterprise JavaBeans (EJB) components to perform business data processing.
4. The EJB components perform their processing, possibly calling external systems.
5. The EJB components return data to the servlet.
6. The servlet then marshals this return value into an XML document.
7. The servlet returns XML to the client on a response.

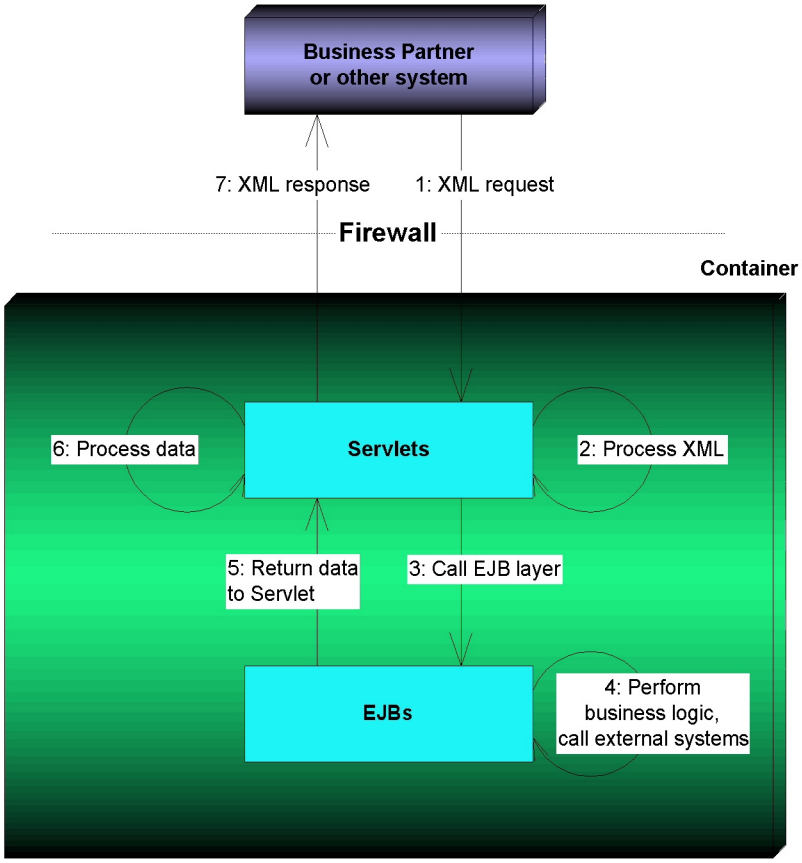

Figure 2 Processing a business partner request.

To achieve this level of business partner connectivity, there must be a way to publish, describe, locate, and call a web service. We now describe how this is achieved.

UDDI

Before a partner can make a web service call to a business, it must first locate a business with the service needed, discover the call interface and semantics, and write or configure software on their end to collaborate with the service. Thus we need a vehicle to publish our web service.

UDDI (Universal Description, Discovery, and Integration) is an important new project aimed towards providers and seekers of web services. The members of the UDDI Project operate a web service called the UDDI Business Registry (UBR), which is global, public directory of businesses and services. Web service providers can register and describe their services in the UBR. Users can query the UBR to discover web services and to locate information needed to interoperate with the services.

UDDI is a mechanism to direct systems looking for certain services to documentation that describes them. UDDI contains the standard “white pages”-type business search and “yellow pages”-type topical search, as well as a “green pages”-type service type search. It is this “green pages” search that will allow a developer to find all services that match a particular service type.

UDDI utilizes SOAP messaging (typically XML/HTTP) for publishing, editing, browsing, and searching for information in a registry. It also contains an XML schema for encapsulating the various types of data that may be returned or sent to the registry service.

JAXR

To support the functionality of UDDI on the Java platform, the Java APIs for XML Registries (JAXR) is a forthcoming API specification that developers can use to access registries. Note that JAXR is not required to build web services today; you can still use the more general XML APIs to interact with the protocols directly. JAXR is a convenience API which provides a Java API to perform the various publishing, querying, and editing tasks these registries support. It focuses exclusively on XML web services being used for B2B applications, and addresses issues such as complex content queries and support for publish/subscribe XML messaging. It can be used to access other types of registries as well, such as an ebXML Registry (described later).

These registry operations are themselves web services and such, can be accessed using current web service tools (e.g. 3rd party SOAP and ebXML messaging tools). However, when JAXR emerges, it will provide a consistent and specialized API for these kinds of registry operations that will make the developer's life much easier.

WSDL

For a business to discover a service it wants to use, it needs to understand the call syntax and semantics prior to actually making a call. The WSDL (Web Services Description Language) specification is an XML document which describes the interface, semantics, and administrivia of a call to the web service. This allows for simple services to be quickly and easily described and documented.

Here's an example WSDL definition:

```
<?xml version="1.0"?>
<definitions name="StockQuote"
          targetNamespace="http://example.com/stockquote.wsdl"
          xmlns:tns="http://example.com/stockquote.wsdl"
          xmlns:xsd1="http://example.com/stockquote.xsd"
          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <schema targetNamespace=http://example.com/stockquote.xsd
            xmlns="http://www.w3.org/2000/10/XMLSchema">
      <element name="TradePriceRequest">
        <complexType>
          <all>
            <element name="tickerSymbol" type="string"/>
          </all>
        </complexType>
      </element>
      <element name="TradePrice">
        <complexType>
          <all>
            <element name="price" type="float"/>
          </all>
        </complexType>
      </element>
    </schema>
  </types>
  <message name="GetLastTradePriceInput">
    <part name="body" element="xsd1:TradePriceRequest"/>
  </message>
```

```
    <message name="GetLastTradePriceOutput">
      <part name="body" element="xsd1:TradePrice"/>
    </message>
    <portType name="StockQuotePortType">
      <operation name="GetLastTradePrice">
        <input message="tns:GetLastTradePriceInput"/>
        <output message="tns:GetLastTradePriceOutput"/>
      </operation>
    </portType>
    <binding name="StockQuoteSoapBinding"
             type="tns:StockQuotePortType">
      <soap:binding style="document"
                    transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="GetLastTradePrice">
        <soap:operation
                 soapAction="http://example.com/GetLastTradePrice"/>
        <input>
          <soap:body use="literal"/>
        </input>
        <output>
          <soap:body use="literal"/>
        </output>
      </operation>
    </binding>
    <service name="StockQuoteService">
      <documentation>My first service</documentation>
      <port name="StockQuotePort" binding="tns:StockQuoteBinding">
        <soap:address location="http://example.com/stockquote"/>
      </port>
    </service>
  </definitions>
```

It contains the following key pieces of information:

- A description/format of the messages that can be passed (via embedded XML Schema Definitions) within the `<types>` and `<message>` elements
- The semantics of the message passing (e.g. request-only, request-response, response-only) within the `<portType>` element
- A specified encoding (various encodings over a specified transport such as HTTP, HTTPS, or SMTP) within the `<binding>` element
- The endpoint for the service (a URL) within the `<service>` element

WSDL is often mentioned along with UDDI, as the format of technical interface descriptions. While UDDI is the most common and recommended place to register a WSDL specification, the UDDI spec does not restrict what type or format of description may be linked to from its registry. It may be WSDL, a regular web page with human-oriented documentation, or even just an e-mail address to contact for information.

There is a Java API for WSDL (JWSDL) specification currently in the works in the Java Community Process (JCP). When released, it will provide an API for manipulating WSDL documents without

interacting with the XML documents directly. While you can currently achieve the full range of functionality using JAXP, using JWSDL will be much easier and faster, simplifying the developer's task.

WSDL and UDDI are shown Figure 3.

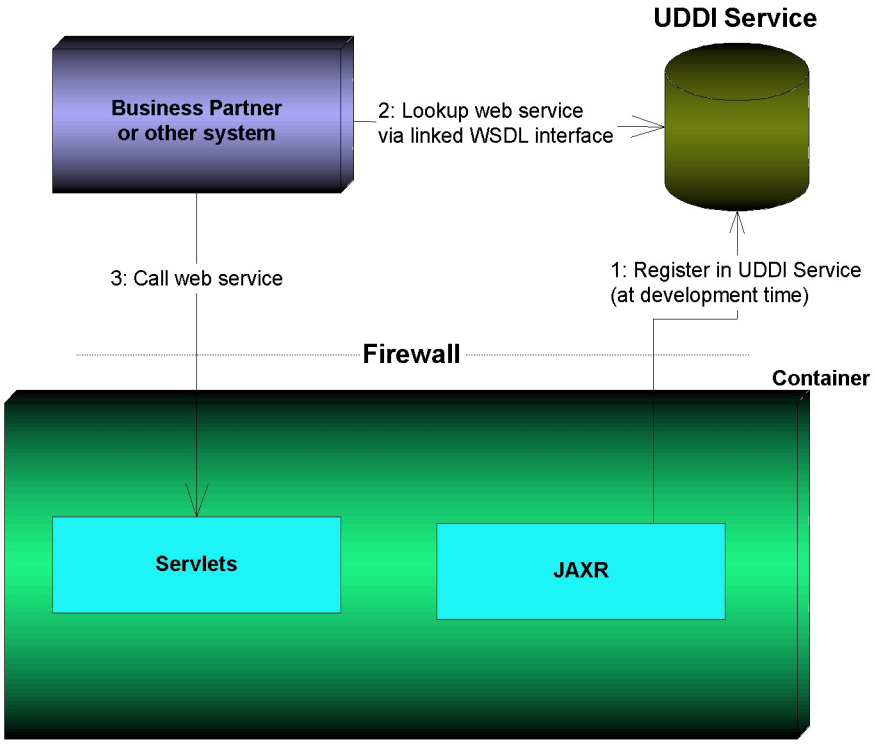

Figure 3 Using JAXR,UDDI, and WSDL.

SOAP

Once a business partner looks up your WSDL description using UDDI, it can call one or more operations on your web service using the Simple Object Access Protocol (SOAP).

SOAP is a specification for performing business method requests as XML documents, and can support a variety of lower level protocols such as HTTP(S) or SMTP. XML is used because of its programming language-neutrality, extensibility, and massive industry support. HTTP is used because any Internet-enabled system can communicate on a socket, because it is a simple protocol that can interoperate with any system, and because it can navigate through firewalls using port 80, which is typically accessible.

The power behind SOAP lies in its simplicity. SOAP is a lightweight and very easy-to-understand technology, and is also easy to implement. It has industry momentum and buy-in from all major eBusiness platform vendors.

From the technical perspective, SOAP specifies how to represent various pieces of "call administrivia," as well as how to encode parameters. A SOAP envelope surrounds the optional header and the body and is most commonly transported as an HTTP POST action to an http server, although other forms of transport (such as SMTP) are also possible. SOAP supports both message-passing and RPC call semantics. This is a sample SOAP call as it appears on-the-wire.

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"
```

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
  <SOAP-ENV:Header>
    <t:Transaction xmlns:t="some-URI" SOAP-ENV:mustUnderstand="1">
      5
    </t:Transaction>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <m:GetLastTradePrice xmlns:m="Some-URI">
      <symbol>SUNW</symbol>
    </m:GetLastTradePrice>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

JAX/RPC

To aid developers in building XML-based requests such as SOAP requests, The JCP is developing the Java APIs for XML based RPC (JAX/RPC). JAX/RPC is used for sending and receiving (including marshalling and unmarshalling) method calls using XML-based protocols such as SOAP, or others such as XMLP (XML Protocol. For more information, see http://www.w3.org/2000/xp/). JAX/RPC isolates you from the specifics of these protocols, enabling rapid application development. There is no longer any need for developers to interact directly with the XML representation of the call.

Currently, there are a variety of 3rd party SOAP implementations, which developers can use to make SOAP calls with varying levels of automation, and developers can tap into those APIs today. In the future, JAX/RPC will supersede these APIs and provide a unified interface to the variety of implementations we have today. JAX/RPC will provide a standard interface for constructing and consuming SOAP RPC requests and automatically manage the marshalling and unmarshalling of method parameters.

In the case of receiving a SOAP request from a business partner, a Java servlet uses JAX/RPC to receive the XML-based request. Once this request is received, the servlet can perform the business processing and return results back to the business partner.

ebXML

For more extended business exchanges where there is a need for an agreed-upon structure for business transactions, multi-request transactions, schemas, and document flow, application requirements often stretch the limits of a purely SOAP based implementation. While SOAP provides a low-level foundation which you can build these extended business exchanges on top of, one might hope for a more advanced framework which already has these issues in mind.

This is the motivation for *ebXML*, a suite of XML specifications and related processes and behavior designed to provide an e-infrastructure for B2B collaboration and integration. It is a full-featured specification which will revolutionize the way businesses adopt partners and conduct business with each other. These are the key components of the ebXML standard:

Collaboration Protocol Profile (CPP)

A CPP describes a company's offerings in a standard, portable way. Specifically, it describes the message-exchange capabilities and business collaborations that a company supports. It also describes the company's business processes, including how partners interact with this company. An interesting facet of a CPP is that a business collaboration includes both sides of a two-party B2B transaction. For example, in a buyer-seller situation, the CPP would describe not only the

selling process and semantics of the seller, but also the buying process and semantics of the buyer.

Collaboration Protocol Agreement (CPA)

A CPA describes the exact requirements and mechanisms for the transactions that two companies perform with each other. It is formed from a manually or automatically derived intersection of their CPPs, which has been reviewed and agreed upon by both sides. This CPA becomes a contract between the two parties and specifies the "rules of engagement" for a particular collaboration.

Examples of a CPP and a CPA, and details of their specifications, can be found at:
http://ebxml.org/project_teams/trade_partner/cpp-example.xml
http://ebxml.org/project_teams/trade_partner/cpa-example.xml
http://www.ebxml.org/specs/ebCCP.pdf

Business Process and Information Modeling

ebXML also includes specifications for describing a business process in XML. This can include transactions, document flow, binary collaborations, data encapsulation formats, and more. These specifications are used by authors when constructing CPPs and are also used to describe and share business processes or information formats.

Core Components

Another crucial part of the ebXML standard are a set of XML schemas, also called core components. These schemas contain formats for business data, such as dates, taxation amounts, account owner, exchange contract, and more. They are specific business constructs and entities, but not aligned to any particular vertical industry.

Messaging

ebXML messaging is a format that encapsulates a message with all the related message-oriented middleware semantics (e.g. asynchronous/synchronous, reliability options). In particular, an ebXML message represents the visible part of the execution of a CPA, and has features specified to enforce the "rules of engagement" specified therein.

ebXML messaging is built on top of SOAP-encapsulated message-passing invocations (as opposed to RPC-style invocations). It extends the SOAP protocol by adding layered frameworks that support attachments, security, and reliable delivery.

Registry/Repository

The ebXML registry/repository is a service that stores CPPs, CPAs, ebXML core components, and other ebXML documents and fragments. It contains powerful query abilities to allow users to search for relevant components and potential business partners. The JAXR API can also be used to access ebXML registries. Business services defined in CPPs and stored in an ebXML registry/repository can then be published to UDDI . A key concept here is while UDDI provides a universal singleton source of **references** to web service information, an ebXML repository is a local container for the actual information itself. Thus in a wide-area web service discovery scenario, a potential business partner would first search for a service in UDDI, which would contain a reference to a CPP or other documentation that is actually stored in an ebXML repository.

JAXM

When receiving a web service request from a business partner, we need a Java API to process ebXML messages from within our servlet, in a similar way to how we processed SOAP requests using JAX/RPC.

The Java API for XML Messaging (JAXM) is a forthcoming specification for interacting with XML messaging standards such as ebXML messaging and SOAP messaging. This API is designed to facilitate the processing of XML message protocols, particularly those where a predetermined "contract" exists (ebXML in particular) to determine the format and constraints of the message. This API will handle all the "envelope" information, such as routing information and the "cargo" manifest, in an intuitive way separate from the actual payload of the message. This allows developers to focus on interacting with the payload and not worry about the other message administrivia.

While developers can currently use JAXP to implement the full range of functionality which JAXM will provide, JAXM will bring a dedicated API specially tailored to the kinds of interaction you need in order to perform XML-based message passing. This will greatly simplify the code which developers will need to write in order to accomplish this task, and bring a unified and standard interface for such code.

The difference between JAXM and JAX/RPC is analogous to the difference between message-oriented middleware (MOM) and remote procedure calls (RPCs). JAXM is geared toward message-oriented middleware-type applications, while JAX/RPC is designed specifically for RPC behavior. JAX/RPC and JAXM are shown in Figure 4.

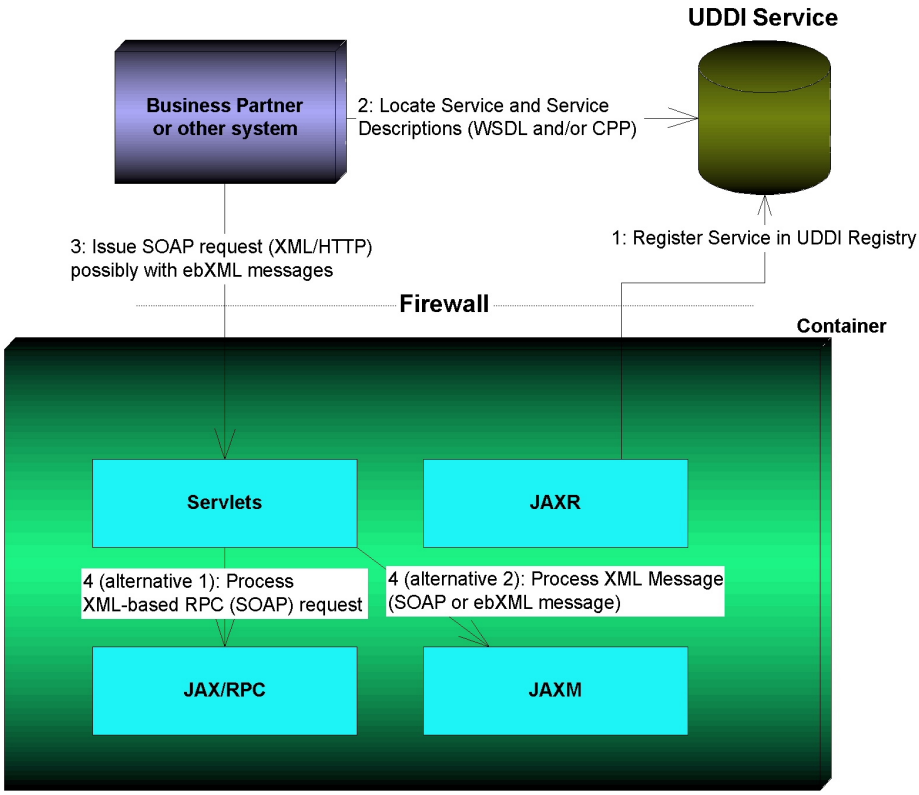

Figure 4 Using JAX/RPC and JAXM from a Servlet.

Note that until mature implementations of JAXM and JAX/RPC are available, developers will need to rely on third-party SOAP APIs, such as Apache SOAP, IdooXOAP, and GLUE. When JAXM and JAX/RPC are released, they should provide a uniform interface to the current array of different SOAP and ebXML messaging implementations. This is analogous to how JDBC provides a uniform interface to relational database drivers.

This completes the discussion of connecting business partners to your web services. Next, let's move on to thin clients and thick clients.

Thin Client Connectivity

Thin clients (such as web or wireless browsers) represent people who are interested in viewing web pages. The web service is responsible for performing any necessary processing on behalf of that web page request, such as executing a B2C transaction, and then showing an order confirmation page.

To achieve this, developers write dynamic web pages using JavaServer Pages (JSP) technology. JSP components are a dynamic page technology that can generate content (often HTML or XML) based upon back-end data processing results. They run within the managed environment of the container, which provides services to the JSP components, such as translating them into executable code.

JSP components act as a front-end "presentation" interface into a back-end layer of business logic, which may be implemented in a variety of ways (e.g. EJBs, ordinary Java objects, or regular JavaBeans). They then generate results, typically as HTML or XHTML (an XML-compliant version of HTML).

JSP components represent an interactive user interface, rather than a programmatic interface that other systems call. For example, a stock quote service might be called as a web service programmatically from an application that is calculating statistical averages of stock quotes. That same back-end stock quote service might return stock quotes via web pages to end users using JSP technology.

The role of JSP components is shown in Figure 5.

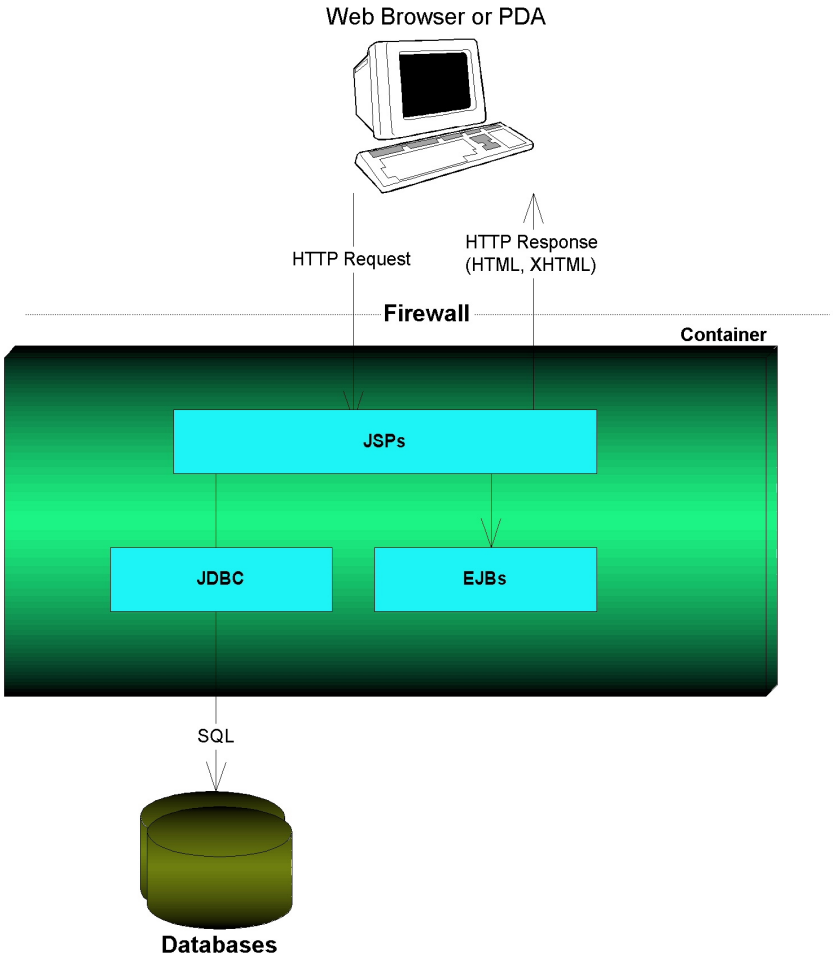

Figure 5 JSP components responding to a web request

Note that the usage of JSP components in this context is illustrative and not prescriptive. Developers may also use Java servlets (described later) to achieve similar results.

Thick Client Connectivity

Some types of clients that connect to a web service are better suited as thick clients. For example, within a company's internal LAN topology, issues such as download times, security, and heterogeneous technologies become less important issues. Responsiveness and functionality of the user interface may become more important, especially if the applications are used on a regular basis, such as an internal call center application.

A thick client can connect to a web service in a variety of ways. For example, it can use web services technologies, such as UDDI, WSDL, SOAP, and ebXML. This paradigm is less performance-efficient, since there is little need for XML translating and processing since both the client application and web service implementation are typically authored by the same development group.

A higher performing approach is for the thick client to connect via a more efficient protocol such as Java RMI-IIOP (Java Remote Method Invocation over the Internet Inter-ORB Protocol). The Object Management Group's IIOP is the standard protocol used to communicate with EJB components, and thick clients can exploit this to connect to the EJB layer directly.

V. Implementing Web Services

So far, we've covered how to connect any type of client to our web service. The next step is to see how to implement the internals of the web service.

Data Translation and Transformation

Before entering a web service, the first challenge we must overcome is building an interface layer to translate the incoming XML data into a format suitable for processing by our business service, and then translating the results of the business service into an XML format to return to the client. A developer thus needs robust mechanisms for parsing XML, binding them to Java objects, generating XML, and transforming various XML formats. Sometimes, because of the variety of interfaces our application supports (e.g. SOAP from B2B partners, HTML from web browser based customers, and WML from wireless browser based customers all going to the same set of services) we may need different routines for each client with their own dedicated web service interface to deal with the semantic differences associated with the different types of client environments.

JAXP

The Java API for XML Processing is a native Java interface to the industry standard XML parsing APIs, SAX (Simple API for XML) and DOM (Document Object Model), along with a pluggable interface to an XSLT (XML Stylesheet Language Transformations) engine. These form the basic foundation for parsing and processing XML documents. While these APIs are a bit primitive for the needs of a web services system, they represent the best software for dealing with XML that has been released so far. JAXP gives us a full-featured API for accessing, modifying, and creating XML documents in Java and is the "foundation" API which our web service interfaces are built with.

For more information, please see:

> http://java.sun.com/xml/tutorial_intro.html
> http://java.sun.com/xml/xml_jaxp.html

JAXB

The Java API for XML Binding is a forthcoming specification for converting an XML document into a Java object and vice versa. Using JAXB, you can automatically convert XML documents into Java

objects for processing by a back-end EJB layer. You can also take Java objects from the EJB layer and convert them back into XML, which is then returned to business partners.

This JAXB interface provides a higher-level way of dealing with an XML document than parsing it with a SAX or DOM parser, while still retaining general applicability. This specification allows for a mapping between an XML schema and a Java class, providing a simple way to transform an XML document into a Java object instance and vice-versa. This is much simpler than parsing documents one tag at a time.

XSLT

XML documents that arrive from business partners may not be of the appropriate schema to be used internally. For example, a business partner may use the term "OrderNum" while internally we use the term "OrderID".

In the opposite direction, we often need to reformat a return value into different formats of XML depending on what kind of client made the call. For example, a call coming from a business partner might be returned in SOAP form, while the same call coming from a web browser would need to be transformed into XHTML. In more complex systems, we may have a variety of presentation formats we need to support, such as WML from a wireless device or VoiceXML from a voice-response system. This requires us to have some mechanism to transform some “base” XML response format into a variety of different XML variants to support the different possible interfaces to our system.

XML Stylesheet Language Transformations (XSLT) is a mechanism to convert an XML document from one schema to another. A stylesheet specifies a number of template-matching rules and applies them in a recursive tree-traversal similar to the DOM paradigm. An XSLT engine can then use this stylesheet to convert XML documents. An XSLT stylesheet’s syntax is very expressive and contains a full repertoire of loops, conditionals, and mathematical expressions, along with function-like constructs and the concepts of scope and recursion.

Shared Context

When two businesses conduct a transaction, a context is usually associated with it. The context can take the form of special agreements (e.g. discount rules, preferential pricing rules) or business rules that apply only to particular partners, thus making the processing of a transaction different for various partners. Furthermore, a given business collaboration may involve several calls spanning a short range a time. Each of these separate calls may be tied together in a shared context that will need to span the entire lifetime of the collaboration instance.

In a J2EE web service, having a discrete location for this context is a recommended part of an implementation. As a developer, you should expect the need for context in a complex web service and plan for a discrete component of your architecture to handle it. Currently this context would typically be implemented manually through normal database code using JDBC (Java Database Connectivity). However, there are plans for an upcoming Context API that will streamline access to the types of context needed in a web service system. Shared context data can be accessed from any type of component, such as a servlet, JSP, or EJB component.

Business Layer

Once incoming XML data has been translated into Java objects, the data is ready to be sent to an EJB business layer for processing. EJB technology is a standard for building business components in Java. Using EJB components, you can gain high-end services from the container, such as security, transactions, persistence, connection pooling, load-balancing, and failure recovery services.

There are three types of EJB components specified in the EJB 2.0 standard:

Session Beans perform work on behalf of clients (they are *verbs*). Session beans are generally short-lived and perform fairly quick actions, such as trading a stock, submitting a purchase order, or calculating taxes on a transaction.

Entity Beans represent business data (they are *nouns*). They are generally long lived and map to an underlying storage, such as an RDBMS or OODBMS system. There are two subtypes of entity beans: *bean-managed persistent* (where you write the persistence logic) and *container-managed persistent* (where the container generates the persistence logic for you). Some examples of entity beans include stocks, orders, customers, employees, and accounts.

MessageDriven Beans are message-oriented components. They receive messages using message-oriented middleware, such as IBM MQSeries or TIBCO Rendezvous. Messages can also be sent from Java clients using the Java Message Service (JMS) standard. When a message arrives for a message-driven bean, it is similarly accessed using the JMS API. Examples of message-driven beans include a logging service, a stock trading service, or an order submission service.

Typically, session beans call entity beans to achieve their desired actions. For example, a pricing engine that computes prices of orders is a session bean which delegates to one or more product and order entity beans. Message-driven beans receive messages and route those messages to session beans or entity beans.

A sample EJB component interaction is shown in Figure 6.

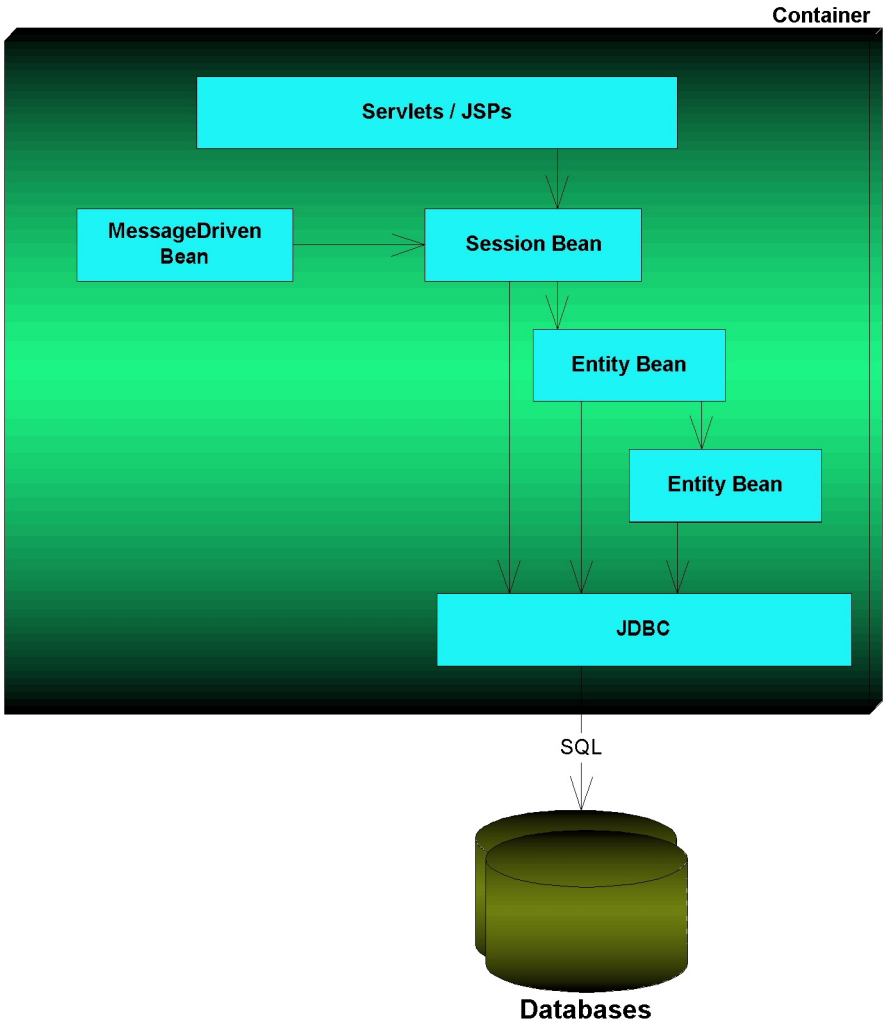

Figure 6 An EJB business layer

You create, find, and destroy EJB components using the Java Naming and Directory Interface (JNDI) API. This API is typically used to access many types of external resources in a J2EE deployment, including database drivers, message-oriented middleware drivers, or EJB component factories.

For more information on EJB, see:

- http://java.sun.com/products/ejb/white/white_paper.html
- http://java.sun.com/products/ejb/
- http://www.theserverside.com
- "Mastering Enterprise JavaBeans" by Ed Roman, published by John Wiley & Sons.

VI. Performing Back-End Integration

The last challenge to overcome when developing a web service using J2EE is connecting to back-end systems, such as databases, legacy systems, and other business partners.

Database Connectivity

To connect to relational databases, developers have a choice of APIs:

> **The JDBC API** is a relational database API used to access any SQL-compliant database. JDBC isolates you from the specific protocol and syntax of a particular database, yielding more portable code. JDBC is a widely accepted standard, and a myriad of JDBC drivers are available for download today.
>
> **SQL/J** is a standard for embedding SQL logic directly into Java code. This is somewhat analogous to how JSP components interlace Java code within HTML tags.

Legacy System Connectivity

Connecting to existing systems has historically been one of the most challenging and burdensome tasks of creating any enterprise deployment. Most enterprises comprise a hodgepodge of existing systems such as SAP R/3, Siebel, i2, and custom systems. Integration has been a manual task, because there are very few adapters available for existing systems. ISVs have been required to write custom adapters for every platform, but this lack of a standard platform has left little or no incentive for ISVs to do this.

The J2EE Connector Architecture (JCA) is an industry movement that is spawning a marketplace of adapters to existing systems. Using the JCA, you can download or purchase an off-the-shelf adapter to connect to an existing system. You can also write your own, if no such adapter exists. These adapters can run in any J2EE-compliant environment.

With the JCA, you gain the benefits of the "write once, run anywhere" paradigm - write the adapters once, and run them in any J2EE environment. For ISVs of existing systems (such as SAP), this creates a unique opportunity for them to solve integration problems for their clients. Indeed, adapters are already being developed as we speak, and this is what makes the JCA so exciting for end developers.

Business Partner Connectivity

The final type of back-end system that we might connect to is another business partner's web service. This business partner system exposes itself using the same universally agreed-upon XML standards that we would use when publishing our own web service. Namely, UDDI as a web service registry, WSDL for describing the web service, and SOAP and ebXML for performing business transactions.

Your EJB component layer can invoke business partners' web services using the JAX* suite of APIs, described earlier in this white paper.

> **Use the Java API for XML Registries (JAXR)** to look up the business partner's web service that is published in a UDDI registry.
>
> **Use the Java API for XML RPC (JAX/RPC)** to perform RPC requests to the external web service.
>
> **Use the Java API for XML Messaging (JAXM)** to send SOAP or ebXML messages to the external web service.
>
> **Use the Java API for XML Parsing (JAXP) and the Java API for XML Binding (JAXB)** for transforming Java data (such as that imported from a database) into an XML format suitable for the partner. You can use the same APIs to convert the received XML data back into a Java language construct, and to perform XSLT transforms to convert schemas.

Using these Java standard APIs with a J2EE web services architecture, we can build powerful cross-platform systems which we can share with our partners, thus providing a complete end-to-end web services solution. This is shown in Figure 7.

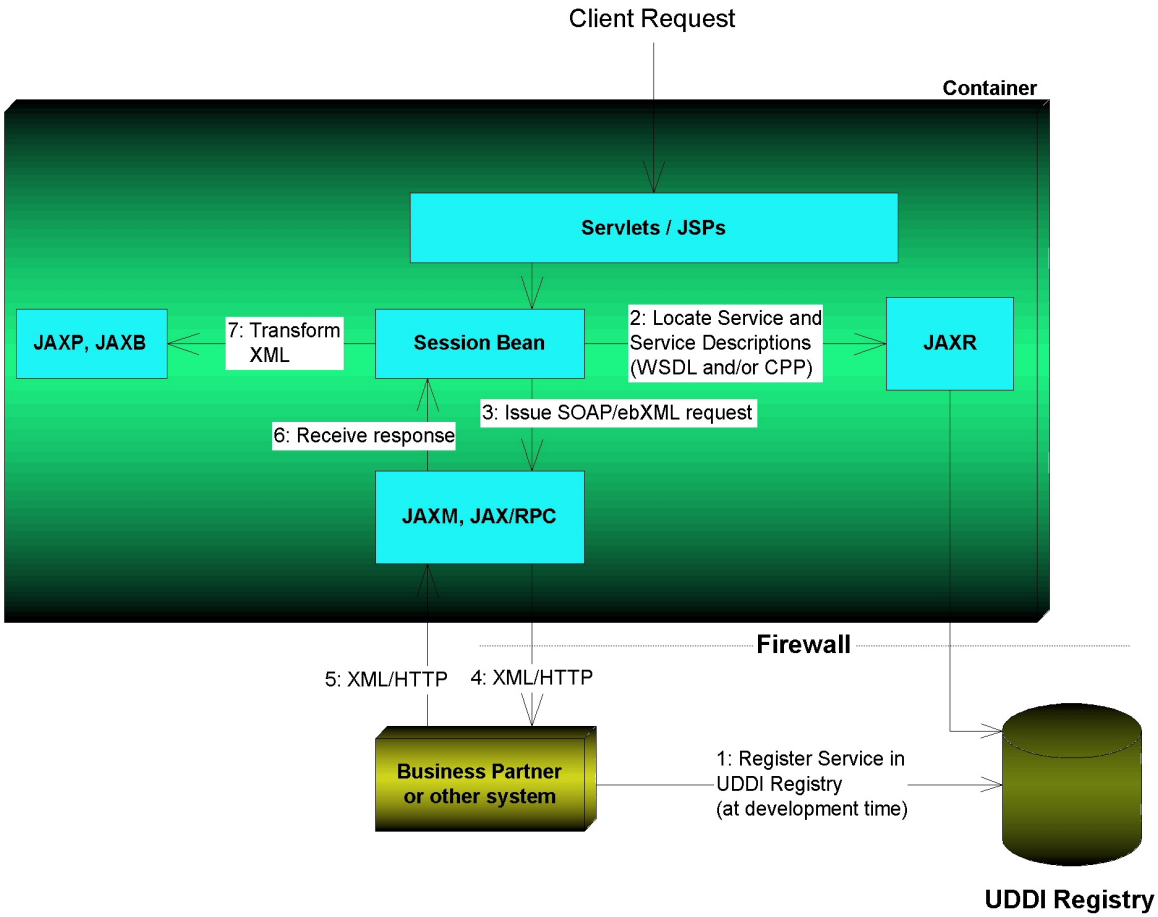

Figure 7 Using the JAX APIs to invoke a business web service.*

VIII. Conclusion

In this whitepaper, we have reviewed how to build a web service using J2EE. A crucial advantage of a J2EE based web services system is the ability to use the existing J2EE infrastructure for 70 to 90 percent of a web services system's functionality. Developers retain all the benefits of a J2EE system's standards based best-of-breed approach, and continue to leverage hard-won J2EE skills. A web services architecture allows for wide flexibility in how it is implemented. Developers who consider thoughtfully the high-level architecture before delving into implementation specifics can build a system largely out of standard and proprietary components selected to best fit the enterprise's needs.

Developers who adopt a J2EE web services architecture can look forward to continually expanding standardization and functionality. Developers can build web services today using existing technology like servlets, JSP, EJB, and JAXP along with a wide selection of SOAP, WSDL, and ebXML tools. The forthcoming release of more advanced JAX API's will further simplify web service development and provide dramatic increases in developer productivity. Advances in both standard and proprietary components will further enhance their ability to quickly design powerful solutions.