Enterprise Java – EJBs and Web Services

Short history
Since its introduction in the mid-90s, the Java platform has been widely successful and accepted both in industry and academia.

Java is now used to program supercomputers down to small tiny computing devices such smart card. However, the place where, arguably, Java has made the greatest stride is in enterprise computing.

The introduction of the Java 2 Enterprise Edition (J2EE) [1] platform has radically changed enterprise computing, by creating a vendor and platform neutral specification that provides a complete set of APIs for almost all aspects of enterprise development.

Today in its 3rd release—with a 4th release in the works—the J2EE specification has been embraced by all major players (except Microsoft) and is widely available for virtually all enterprise computing platform (e.g. Linux, UNIX, Win2K/XP, HP-UX, Solaris, OS400, OS390, etc…)

The introduction of Microsoft .NET [2] as a competitive platform creates an alternative to the J2EE platform but also came about with an industry shift towards easy integration based on XML and the WWW. “Web services” [6] is the moniker used to describe, locatable, describable and programmable network port on the Web using standard protocol based on XML. With Web services the J2EE and .NET platform as well as other legacy environment can easily interoperate.

In this lecture we give an overview of the J2EE platform, specifically the Enterprise JavaBeans server-side components. We then describe the emergent Web services standard as well as the key Java Web services specification JAX-RPC (JSR101) [11].

J2EE Overview
The enterprise edition of the Java 2 platform is a set of specifications describing OO APIs that together with concrete product implementation yield a platform for enterprise distributed object-oriented applications, that is robust and scalable. The industry coined the term “application server” to describe such platform; concrete examples are IBM’s Websphere, BEA’s WebLogic and Sun’s ONE.

In general a J2EE application server constitute of:
1. Web container – used to create presentation information and extending the Web server implementing the HTTP/HTTPS/SSL protocols

Outline for Lecture < XY>
1. J2EE Overview
2. Intro to Enterprise JavaBeans
   a. Session
   b. Entity
3. Other J2EE APIs – Brief summary
4. Web services
   a. WSDL
   b. JAX-RPC
5. References
2. EJB container – used to create scalable, secure and “focused” server-side components
3. Additional enterprise services to: connect to datastore (JDBC), secure authentication and authorization (JAAS), secure enterprise messaging substrate (JMS), scalable directory and naming service (JNDI), transaction support (JTS, JTA) etc…

Figure 1 shows an example multi-tier enterprise Web application implemented with the J2EE platform. This example tries to show the various aspects of the J2EE platform. However, not all APIs are applicable to all J2EE application deployment and also the architecture typically spans more than one server—for fault-tolerance and to better distribute components.
(MVC) [4][5] architecture depicted shows the controller implemented as a Java servlet running in the Web container and the model implemented as a set of EJBs running in the EJB container.

Note also that the other services depicted in the figure such as datastore, naming and directory service can be part of the same application server but this is not a requirement.

The EJBs model components can also be distributed and remotely accessed via RMI, more on this in the next section. For a good overview on designing J2EE application see the J2EE blue prints [3].

Also, the figure shows the consumption of a Web service. The service is shown implemented in .NET and located via a UDDI [7] registry.

**Intro to Enterprise JavaBeans (EJBs)**

When implementing a J2EE application, the model or business layer of the application is implemented using EJBs. They are server-side OO software components that represent the business logic and data of the application [8]. For instance, imagine an eStore application. EJBs would be used to represent domain business things such as Customers, ShoppingCart, CreditCardPayment, Checkout, etc...

Unlike their distant cousin, JavaBeans, EJBs require a container where they are deployed and where they reside. They have very structured lifecycle and participate in various contracts with the container who in turn adorn them with function such as transactions, security and persistence.

In short, EJBs are a component architecture designed to allow focused development of business logic and data while delegating system level aspects such as security, transaction, persistence to the application server provider.

An EJB component is reusable since its system level attributes can be “added” to the component at deployment time. This means that the bean can be deployed in various applications assuming that the application necessitates the domain object and business logic that the bean represents. Figure 2 shows an overview of the various EJB users and their roles.
Note: these roles are only conceptual and are not required in practice. The same user could play all roles for instance. However, these conceptual demarcations of roles speak to the fact that EJBs are designed to be reusable software components.

It’s also worth noting that EJBs are packaged in JAR files which contain all the EJB classes and other necessary supporting Java classes as well as a META-INF directory with the XML deployment descriptor file ejb-jar.xml.

There are 3 kinds of EJBs:

1. **Entity** beans – these represents surrogate for entities in the domain being modeled e.g. Customer and Employer. They can be either bean managed or container managed.

2. **Session** beans – these represent tasks or business functions. They can be *stateless* (pure business service) e.g. CreditCardPayment or *stateful* sessions (keeping conversational state) e.g. ShoppingCart.
3. *Message Driven* beans\(^1\) – these represents the tasks performed in response to an asynchronous message via a messaging infrastructure like JMS.

**Session Beans**

Each EJB (regardless of type) is comprised of a Home (or HomeRemote) and Remote interface\(^2\). The Home interface is used like a Factory class [14] to create and locate the Remote interface of the bean. The Remote interface is the interface with the business methods for the EJB.

Client use the Home interface to locate the Remote interface and then use the latter to make business method calls to the bean. The Remote interface extends java.rmi.Remote and all its methods must throw java.rmi.RemoteException.

In EJB 2.0, a bean can also expose a Local interface and a LocalHome to locate it. The Local interface does not extend the java.rmi.Remote interface and their methods do not need to throw java.rmi.RemoteException. A Local interface, as the name suggest, is intended to be used by co-located clients or beans—that is clients and beans that are running in the same JVM as the bean exposing the Local interface.

Figure 3 shows the UML class diagram [10] for a *ProcessPayment* session bean whose Remote interface (*ProcessPaymentRemote*) contains two business methods:

```
+ creditCard( cc: CreditCard, value: float ) : void
  { throws RemoteException, InvalidCCException }
+ cash( value: float ) : void { throws RemoteException }
```

The *CreditCard* class passed to the creditCard( ... ) method is a simple JavaBeans data object used to contains the information of a credit card as the payment is being processed\(^3\).

---

\(^1\) MDB EJBs were introduced in the 2.0 revision of the EJB spec. They are relatively new and require a messaging infrastructure for their implementation. Their structure is relatively simple in that they do not contain business methods in their bean interface but one method onMessage( msg: Message ): void where the msg is a JMS message. Please see [8][9] for more details on MDBs.

\(^2\) The remote interface is sometimes suffixed with Remote.

\(^3\) The *CreditCard* is not modeled as an entity bean since it is not intended to be persisted. However, in some situation a reference to the *CreditCard* might be useful (to facilitate fast payment) and in this case it would be an entity bean and persisted to the database with a key and relationship to the *Customer* entity for instance.
The Home interface (ProcessPaymentHome) contains just one method:

```
+ create() : ProcessPayment { throws CreateException }
```

It is used by clients to create (accessed) an instance of the ProcessPayment session bean.

To complete the ProcessPayment EJB we need to provide the Bean class. This class implements the correct `javax.ejb.<Session/Entity/Message>Bean` interface and contains methods corresponding to the Remote and/or the Local interface. In these methods the `Enterprise Bean Provider` code the business logic of the bean. This method can be passed references to other beans (e.g. Customer entity bean) or access other beans (via JNDI) to implement its logic.
Access to J2EE services in the Bean methods is also allowed; however, the Bean's methods should not directly worry about system level issues such as synchronization, transactions, threading, security, etc... since these aspects are provided by the container and are attributes that are added when the bean is deployed to the EJB container.

Listing 1 shows part of the XML deployment descriptor (DD) for the ProcessPayment bean. In general, a DD can be used to deploy more than one bean. The Server or Container Provider also typically provides tools to facilitate the creation of DDs as well as the entire deployment process4.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD ...">
<ejb-jar>

<enterprise-beans>
  <session>
    <ejb-name>ProcessPayment</ejb-name>
    <home>com.xyz.ejb.ProcessPaymentHome</home>
    <remote>com.xyz.ejb.ProcessPayment</remote>
    <ejb-class>com.xyz.ejb.ProcessPaymentBean</ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Container</transaction-type>
    <!-- Add ejb-ref -->
  </session>
</enterprise-beans>

<assembly-descriptor>
  <!-- Security attributes -->
  <security-role>
    <description>Everyone calling this bean</description>
    <role-name>everyone</role-name>
  </security-role>
  <method-permission>
    <role-name>everyone</role-name>
    <method>
      <ejb-name>ProcessPayment</ejb-name>
      <method-name>*</method-name>
    </method>
  </method-permission>
  <!-- Transaction attributes -->
  <container-transaction>
    <method>
      <ejb-name>ProcessPayment</ejb-name>
      <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
</assembly-descriptor>
</ejb-jar>

Listing 1 ProcessPaymentBean deployment descriptor

The <enterprise-beans> element of the DD contains a list of the different EJBs deployed; each with its own series of elements to describe the Home, Remote and Bean class as well as other characteristics for the deployment of

4 Assume that the beans are created in a package com.xyz.ejb (not shown in diagram). However, in general, various packages would be used and different beans would be in their own packages.
the bean. The `<assembly-descriptor>` section of the DD contains security and transactional attributes that should be added to each bean at deployment-time by the container.

Once the bean is deployed to the application server, client request can be serviced. Typical clients to a bean using the `Remote` interface need to:

1. locate the bean `Home` interface—via the JNDI API
2. create an instance of the bean via the `Home` interface—if the instance is pooled the application server just returns the instance
3. execute the business methods

For example, the following simple example application shows the various steps for client access to the `ProcessPayment` session bean to execute a `CreditCard` payment.

```java
protected Context getInitialContext() throws Exception {
    if ( initialContext == null ) {
        Hashtable props = new Hashtable();
        props.put( Context.INITIAL_CONTEXT_FACTORY,
                "com.ibm.websphere.naming.WsnInitialContextFactory" );
        props.put( Context.PROVIDER_URL, "iiop://localhost:2809/" );
        initialContext = new InitialContext( props );
    }
    return initialContext;
}

private ProcessPaymentHome getProcessPaymentHome() throws Exception {
    Context jndiContext = getInitialContext();
    Object homeRef = jndiContext.lookup( "ejb/edu/ncsu/csc517/ProcessPaymentHome" );
    ProcessPaymentHome processPaymentHome = (ProcessPaymentHome)PortableRemoteObject.narrow( homeRef, ProcessPaymentHome.class );
    return processPaymentHome;
}

//Some application method using session bean
try {
    ProcessPaymentHome processPaymentHome = getProcessPaymentHome();
    ProcessPayment processPayment = processPaymentHome.create();
    Calendar calendar = Calendar.getInstance();
    calendar.set( 2003, 12, 01 );
    CreditCard creditCard = new CreditCard( "John", "Doe", "1234567890", calendar.getTime() );
    processPayment.creditCard( creditCard );
} catch( Exception e ) {
    //Process error
}
```

Guest lecture, “Enterprise Java™ -- EJBs and Web Services” © 2002, E. Michael Maximilien <maxim@us.ibm.com>
Listing 2 ProcessPayment session bean client example

When the client application code does `processPaymentHome.create()` to access the bean remote interface, does that mean that a new instance of the bean is created?

Actually, usually no, typically application servers will pool EJBs and make them available to the application as they are needed. This brings us to the lifecycle of the beans in the EJB container.

Figure 5 shows the lifecycle of a session bean. At first, all beans are in the Does not exist state, when the application server creates the pool new instances are created and the `setSessionContext()` and `ejbCreate()` methods are called on each. Each bean then transitions to the Method-ready Pool state. When a client does `Home.create()`, the bean returned would be in the ready state and is capable of processing business methods. Since this is a session stateless bean no other state transition occurs until the bean is removed by the application server.

![Stateless session bean lifecycle state diagram](image)

**Figure 5** Stateless session bean lifecycle state diagram [8]

Stateless session beans have the simplest lifecycle characteristics. Stateful and Entity beans follow a more complex lifecycle.

**Entity Beans**

The other kind of bean that is provided by the EJB specification is entity bean. Entity beans usually map to nouns in the application domain e.g. Customer, Address, etc... Since entity beans represent state information that need to be persisted, they usually contain attributes that are persisted in a backend datastore. Also, an entity bean must provide a key field whose value uniquely identifies the bean instance e.g. `Customer.email`.

Like session beans, entity beans, have Home, Remote and Local interfaces. However, the Home interface in an entity bean usually provides finder methods...
that allows client to locate beans using the primary key or some customized query using the EJB QL\(^5\) language.

Listing 3 shows the complete set classes and interfaces for a Customer entity bean; this bean is shown to have both a Local and Remote interface. Only the findByPrimaryKey() finder method is shown.

```java
public interface CustomerHome extends javax.ejb.EJBHome
{
    public Customer create( String email )
        throws javax.ejb.CreateException,
        java.rmi.RemoteException;

    public Customer findByPrimaryKey( String primaryKey )
        throws javax.ejb.FinderException,
        java.rmi.RemoteException;
}

public interface CustomerLocalHome extends javax.ejb.EJBLocalHome
{
    public CustomerLocal create( String email )
        throws javax.ejb.CreateException;

    public CustomerLocal findByPrimaryKey( String primaryKey )
        throws javax.ejb.FinderException;
}

public interface CustomerLocal extends javax.ejb.EJBLocalObject
{
    public String getFirstName();
    public void setFirstName( String newFirstName );

    public String getLastName();
    public void setLastName( String newLastName );
}

public interface Customer extends javax.ejb.EJBObject
{
    public String getFirstName() throws java.rmi.RemoteException;
    public void setFirstName( String newFirstName )
        throws java.rmi.RemoteException;

    public String getLastName() throws java.rmi.RemoteException;
    public void setLastName( String newLastName )
        throws java.rmi.RemoteException;
}

public abstract class CustomerBean implements javax.ejb.EntityBean
{
    //---------------------------------------------------------------
    // Instance variables
    //---------------------------------------------------------------

    private javax.ejb.EntityContext myEntityCtx;

    //---------------------------------------------------------------
    // Lifecycle methods
    //---------------------------------------------------------------

    \(^5\) EJB QL is a query language added to the EJB 2.0 specification. It is based on the SQL-92 standard.
    Guest lecture, "Enterprise Java™ -- EJBs and Web Services" © 2002,
    E. Michael Maximilien <maxim@us.ibm.com>
public void setEntityContext( javax.ejb.EntityContext ctx )
{
    myEntityCtx = ctx;
}

public javax.ejb.EntityContext getEntityContext()
{
    return myEntityCtx;
}

public void unsetEntityContext() { myEntityCtx = null; }

public void ejbActivate() {}

public String ejbCreate( String email )
throws javax.ejb.CreateException
{
    setEmail(email);
    return null;
}

public void ejbPostCreate( String email )
throws javax.ejb.CreateException {}

public void ejbLoad() {}

public void ejbPassivate() {}

public void ejbRemove() throws javax.ejb.RemoveException {}

public void ejbStore() {}

//---------------------------------------------------------------
// Abstract getter and setter for Entity attributes
// Application server will implement when bean is deployed
// and map attribute to correct field in DB table for bean
//
public abstract String getFirstName();
public abstract void setFirstName( String newFirstName );

public abstract String getLastName();
public abstract void setLastName( String newLastName );

public abstract String getEmail();
public abstract void setEmail( String newEmail );
}

Listing 3 Customer entity bean code

Since entity beans represent things with state in an application, they need to be persisted. This persistence is done usually via a backend datastore e.g. a relation database. Figure 6 shows a RDBMS schema for table that represents the customer.

<table>
<thead>
<tr>
<th>CUSTOMER</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRING(40) : FIRST_NAME</td>
</tr>
<tr>
<td>STRING(40) : LAST_NAME</td>
</tr>
<tr>
<td>STRING(40) : EMAIL*</td>
</tr>
</tbody>
</table>

Figure 6 Customer entity bean database schema

Two strategies for the persistence of entity beans can be used:
1. Bean managed persistence (BMP)—in this case the bean provider needs to manage and code the persistence to the datastore. For instance, the
getFirstName():String method would result in an SQL query to the CUSTOMER table to retrieve the FIRST_NAME field as in:

```sql
SELECT CUSTOMER.FIRST_NAME FROM CUSTOMER WHERE CUSTOMER.EMAIL = <email-value>
```

2. Container managed persistence (CMP)—in this case the container provides all persistence management.

The lifecycle for an entity bean is more complex, since as per the EJB specification an entity bean state must persist client crashes, transaction roll-back etc … This is shown by the calls to ejbStore() and ejbLoad() methods in the Ready state. Further, entity beans can also be pooled or passivated—shown with transition from Ready to Pooled state. This is done when the application server is running out of resources e.g. database connections or memory.

Figure 7 Entity bean lifecycle state diagram [8]
Other J2EE APIs Distilled

We have only scratched the surface of the server-side component support in J2EE. The platform also provides various other services required for the creation of complete distributed OO enterprise applications—please see [1][3] for references:

1. Java Database Connectivity (JDBC)—used to give transparent access to database systems in Java. This is a widely supported standard with drivers for almost all commercial relational database systems.
2. Java Messaging Service (JMS)—gives transparent access to enterprise messaging systems. Used for asynchronous message delivery and publish/subscribe type of messaging.

Java Authorization and Authentication Service (JAAS)—trust and security framework; implements the Kerberos

3. [18] authentication framework
4. Java Transaction Service (JTS)—provides a means to create distributed transactions in J2EE

The J2EE platform is currently being extended to also support Web services; primarily two specifications are used for that support:

1. J2EE Web services support or JSR109 [12]—this specification describe how to add support for Web services into the J2EE platform. Specifically, it introduces an endpoint interface that stateless session bean can implement and expose as a WSDL portType.
2. Java API for XML Remote Procedure Call (JAX-RPC) or JSR101 [11]—defines WSDL <- -> Java mapping as well as a client and server side programming model for consuming and exposing Web services in Java.

Both specifications are currently in final review stages but should be included in the J2EE 1.4 specification final release.
Web Services

Web services allow exposing business methods on the Web using a series of XML based standards. The typical protocol for the communication is SOAP [19] and the description of the services is done via WSDL [20].

Figure 8 shows the general architecture of applications using Web services. The application typically query a registry that contains a listing of service description categorized in some manner e.g. UDDI registries [7]. Once the client makes the choice of the service to use, it then binds with the service provider (found in the WSDL description) and then makes request to the service. This of course assumes that the provider published the service in the registry prior to queering the registry.

Web Services Description Language (WSDL)

The WSDL description of a service includes the following parts (cf. Figure 9):

1. wsdl:types—defines any custom type used by the service. Typically XML Schema is used for this definition
2. wsdl:messages—messages are defined as request with specific parts or parameters. Messages are combined into operations.
3. `wsdl:portType`—a collection of operations. Essentially the service interface
   a. `wsdl:operation`—defines operations with input and output messages
4. `wsdl:service`—a collection of `portTypes`
5. `wsdl:binding`—concrete address and encoding for a `portType`. The most common binding is SOAP, though others can be used
Figure 9 Web service abstract XML description – StockQuoteService WSDL
JAX-RPC (JSR101)\(^6\)

JAX-RPC is the standard worked in the Java Community Process (JCP) as Java Specification Request 101 (JSR101).

The standard defines programming models for clients of Web services in Java as well as mappings from WSDL to Java and vice versa.

In general, a client will use some tooling to generate the client side proxy objects that match the WSDL document for the service selected. These proxy objects give the client access to the service.

JAX-RPC also defines a dynamic invocation interface (DII) that allows clients to generate requests to a service without the need for the proxy classes—the DDI is similar to the reflection API in Java.

Table 1 shows the mapping of XML schema types to Java, specified by JAX-RPC. This is used mostly for the \texttt{wsdl:types} section of the WSDL document as well as when mapping Java operation parameters and return types to XML and of course \texttt{wsdl:part} types to Java.

<table>
<thead>
<tr>
<th>XML Type</th>
<th>Java Type</th>
</tr>
</thead>
</table>
| \texttt{xsd:string, xsd:integer, xsd:int, xsd:long,}  
  \texttt{xsd:float, xsd:double, xsd:boolean, xsd:byte} | \texttt{String,}  
  \texttt{java.math.BigInteger, int,}  
  \texttt{long, float, double,}  
  \texttt{boolean, byte} |
| \texttt{xsd:dateTime} | \texttt{java.util.Calendar} |
| \texttt{<xsd:complexType name=\texttt{ArrayOfInt}>}  
  \texttt{<xsd:complexContent>}  
  \texttt{<xsd:restriction base=\texttt{soapenc:Array}>}  
  \texttt{<xsd:attribute ref=\texttt{soapenc:arrayType}}  
  \texttt{wsdl:arrayType=\texttt{xsd:int[]}"/>}  
  \texttt{</xsd:restriction>}  
  \texttt{</xsd:complexContent>}  
  \texttt{</xsd:complexType>} | \texttt{int[]} |
| \texttt{<xsd:complexType name=\texttt{StockPrice}>}  
  \texttt{<xsd:sequence>}  
  \texttt{<xsd:element name=\texttt{price" type=\texttt{xsd:float}}/>}  
  \texttt{<xsd:element name=\texttt{volume" type=\texttt{xsd:int}}/>}  
  \texttt{</xsd:sequence>}  
  \texttt{</xsd:complexType>} | \texttt{class StockPrice}  
  \texttt{...}  
  \texttt{public float getPrice()}  
  \texttt{...}  
  \texttt{public void setPrice( int i ) {...}}  
  \texttt{public int getVolume()}  
  \texttt{...}  
  \texttt{public void setVolume( int i ) {...} }  
  \texttt{...} |

\(^6\) As of this writing there are no complete released implementations of JAX-RPC – though that should change quickly. The Apache Axis [21] SOAP server is the closest implementation available publicly and was used to generate some of the code in this section; however, I manually modified the generated code to make it match the JAX-RPC specification.
Table 1 JAX-RPC type mapping – Java ↔ XML [11]

Table 2 shows the WSDL to Java mapping. Essentially, the `wsdl:types` and `wsdl:part` section of the WSDL document maps to either Java classes (JavaBeans) or Holder classes; `wsdl:portType` and `wsdl:service` to Java interfaces. The `wsdl:portType` is the key client interface since it exposes all the `wsdl:operation` that each map to a Java method.

When a Java method throws an exception, that exception is defined in the `wsdl:types` and it is used in the definition of the appropriate `wsdl:operation` as a `wsdl:message`.

The `wsdl:service` maps to a Java interface that gives access to all of the endpoint interfaces.

<table>
<thead>
<tr>
<th>WSDL</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wsdl:definitions</code></td>
<td>Java package</td>
</tr>
<tr>
<td><code>wsdl:types</code></td>
<td>appropriate Java type and JavaBeans class as defined in Table 1</td>
</tr>
<tr>
<td><code>wsdl:part</code></td>
<td>method parameter: <code>wsdl:input</code> are passed by value, <code>wsdl:output</code> are passed wrapped in a Holder class</td>
</tr>
<tr>
<td><code>wsdl:operation</code></td>
<td>a method in the service endpoint</td>
</tr>
<tr>
<td><code>wsdl:portType</code></td>
<td>service endpoint interface with methods as per <code>wsdl:operation</code> definitions</td>
</tr>
<tr>
<td><code>wsdl:service</code></td>
<td><code>javax.xml.rpc.Service</code> interface that groups a set of ports and thus gives factory methods to create <code>javax.xml.rpc.Call</code> objects for each port</td>
</tr>
</tbody>
</table>

Table 2 JAX-RPC WSDL mapping – WSDL structure ↔ Java [11]

Using the JAX-RPC specification we can anticipate the following set of classes for the `StockQuoteService` of Figure 9.

```java
package edu.ncsu.csc517.services;

public class StockPriceType implements java.io.Serializable {
    private float price;
    private int volume;

    public StockPriceType() {}
}
```

Guest lecture, "Enterprise Java™ -- EJBs and Web Services" © 2002, E. Michael Maximilien <maxim@us.ibm.com>
public float getPrice() { return price; }

public void setPrice( float price ) { this.price = price; }

public int getVolume() { return volume; }

public void setVolume( int volume ) { this.volume = volume; }

//Various methods for serializing and deserializing type
//depends on tooling

package edu.ncsu.csc517.services;

public interface StockQuoteServiceService
  extends javax.xml.rpc.Service
{
  public String getStockQuoteServiceAddress();

  public StockQuoteServicePT getStockQuoteService()
    throws javax.xml.rpc.ServiceException;

  public StockQuoteServicePT getStockQuoteService( java.net.URL port )
    throws javax.xml.rpc.ServiceException;
}

package edu.ncsu.csc517.services;

public interface StockQuoteServicePT
  extends java.rmi.Remote
{
  public StockPriceType getLastTradePrice( String symbol )
    throws java.rmi.RemoteException;
}

Listing 4 Example Java code generated from StockQuoteService WSDL

JAX-RPC also has a dynamic invocation interface (DDI) to allow clients to
dynamically invoke services without the need for the proxy classes—Listing 5
shows an example client that just makes a simple request to the
StockPriceService getLastTradePrice operation using the DDI.

import javax.xml.rpc.Call;
import javax.xml.rpc.Service;

public class Client
{
  public static void main( String [] args )
  {
    try
    {
      String endpoint =
        "http://localhost:8081/axis/services/StockPriceService";

      Service service = new Service();
      Call call = (Call)service.createCall();

      call.setTargetEndpointAddress( new java.net.URL( endpoint ) );
      call.setOperationName( new QName("StockPriceService",

"getLastTradePrice") );

Object price = call.invoke( new Object[] { "IBM" } );

System.out.println( "StockPriceService.getLastTradePrice( " +
"IBM" + " ) == " +
price.toString() +
" -- [Response class == " +
price.getClass() + "]" );
}
catch( Exception e ) { System.err.println( e.toString() ); }
}
"

Listing 5 JAX-RPC Dynamic Invocation Interface (DII) client example

Alternatively, client can use generated proxy classes of Listing 4 to access the service and make calls to the interface mapping the wsdl:portType.
References


