Java API for XML-based Remote Procedure Call (JAX-RPC)

Dr. Kanda Runapongsa  
(krunapon@kku.ac.th)  
Department of Computer Engineering  
Khon Kaen University

Agenda

- Background on remote communication
- What is and Why JAX-RPC?
- Development steps of a JAX-RPC Service
- Type Mapping
- Client Programming
- Service Endpoint Model
Remote Procedure Call (RPC)

- RPC, COM, CORBA, RMI
  - **Synchronous** communication: calling process blocks until there is a response
  - More tightly coupled (than non-RPC model): client must find recipients and know method and its arguments
  - Non persistent

Remote Procedure Calls

- **Common Interface** between client and server
- Stub for client, Tie/skeleton for server
- On-the-wire protocol needs to be agreed upon
Java API for XML-based Remote Procedure Call (JAX-RPC)

**RPC Example - JAX-RPC**

![Diagram](https://via.placeholder.com/150)

- **HelloClient Program**
- **Stubs**
- **JAX-RPC Runtime**
- **SOAP Message**
- **HelloService**
- **Ties**
- **JAX-RPC Runtime**

**Common Interfaces**

- Service is described in IDL (Interface Description Language)
  - WSDL for Web service
  - Java RMI interface in RMI (Language specific)
- Used by tools to statically generate or dynamically configure interfaces, proxies, and ties in a specific environment
Concept of XML-based RPC

- Uses Standards based on XML
  - SOAP is the “protocol”
  - WSDL is the IDL
- Any text based protocol can be used as transport
  - HTTP, SMTP, FTP, etc.

Does “XML-based RPC” make sense?

- Text is not an efficient way to encode data?
- XML just makes it worse
  - Verbose in nature
  - Slower
- I thought HTTP was for web pages...
- Messaging is more robust than RPC
Java API for XML-based Remote Procedure Call (JAX-RPC)

Agenda
- Background on remote communication
- What is and Why JAX-RPC?
- Development steps of a JAX-RPC Service
- Type Mapping
- Client Programming
- Service Endpoint Model

Why XML based RPC on the Internet?
- Everyone is already connected and using HTTP
- XML is an acceptable standard
- SOAP will go through firewalls
  - Can be filtered when it becomes a problem
- RPC is an easy programming model
  - Message (document-driven) model is gaining momentum, however
- JAX-RPC supports document-driven model as well
### What is JAX-RPC?

- **Java API for XML-based RPC**
  - Web services operations are performed by exchanging SOAP 1.1 messages

- **Services are described using WSDL**
  - WSDL is the contract between service provider and client

- **Web service endpoints and clients use JAX-RPC programming model**

- **Key technology for Web Services in the upcoming J2EE 1.4 platform**

### JAX-RPC Design Goals

- **Easy to use programming model**
  - For both defining & using a service

- **Hides all the plumbing**
  - You don’t have to create SOAP messages yourself

- **SOAP and WSDL-based interoperability**
  - Interoperate with any SOAP 1.1 compliant peers

- **Extensibility and Modularity**
  - Support future versions of XML specification
**JAX-RPC Architecture**

- **JAX-RPC Client**
  - Generated Code
  - Container
  - Client-side JAX-RPC Runtime System

- **JAX-RPC Service Endpoint**

- **WSDL Document**

- **Container**

- **SOAP**

- **HTTP**

**JAX-RPC Runtime System**

- **Core of a JAX-RPC implementation**
  - Library that provides runtime services for JAX-RPC mechanisms
  - Implements some of the JAX-RPC APIs

- **Client side**
  - Can be implemented over J2SE, J2EE, or J2ME platforms
  - J2EE 1.3 or 1.4 Containers
Java API for XML-based Remote Procedure Call (JAX-RPC)

JAX-RPC Web Service Endpoint

Agenda

- Background on remote communication
- What is and Why JAX-RPC?
- Development steps of a JAX-RPC Service
- Type Mapping
- Client Programming
- Service Endpoint Model
Developing a Web Service

- Interface (java.rmi.Remote type)
  - Must follow JAX-RPC conventions
- Implementation classes
  - Servlet-based endpoint model
  - Optional handler and serializer classes
- WSDL
  - Service contract
- Packaged application (war file)

You can start from WSDL as well

- Contract for the service you want to implement
- Generated interfaces and value types
- Implementation classes
1. Code the Service Endpoint Interface (SEI) and implementation class and interface configuration file
2. Compile the SEI and implementation class
3. Use wscompile utility program to generate the WSDL and other files required to deploy the service
4. Package the files into a WAR file
5. Deploy the WAR file

1.a. Code Service Endpoint Interface
- Declares the methods that a remote client may invoke on the service
- Rules
  - It extends the java.rmi.Remote interface
  - It must not have constant declarations, such as public final static
  - The methods must throw the java.rmi.RemoteException or one of its subclasses
  - Method parameters and return types must be supported JAX-RPC types
Example: Service Definition Interface (HelloIF.java from “helloservice”)

```
package helloservice;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface HelloIF extends Remote {
    public String sayHello(String s) throws RemoteException;
}
```

1. B Code Service Implementation

- Service implementation class is an ordinary Java class (for servlet-based Web service endpoint) - helloservice example
- Service implementation class is a stateless session bean (for Stateless session bean based Web service endpoint)
Example: Service Implementation (HelloImp.java from "helloservice")

```java
package helloservice;

public class HelloImpl implements HelloIF {
    public String sayHello(String s) {
        return message + s;
    }
}
```

1.C Interface Configuration File

- Specifies information about the SEI
- Used by `wscompile` to generate WSDL

```xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration
    xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
    <service
        name="MyHelloService"
        targetNamespace="urn:Foo"
        typeNamespace="urn:Foo"
        packageName="helloservice">
        <interface name="helloservice.HelloIF"/>
    </service>
</configuration>
```
2. Compile the SEI and Implementation Class

- Go to directory
  `<INSTALL>/j2eetutorial14/examples/jaxrpc/helloservice`

- Then type command
  `asant compile-service`

- Compile Service definition interface and implementation classes
  - `HelloIF.java`
  - `HelloImpl.java`

- Writing the class files to the `build` subdirectory

3. Use wscompile to Generate WSDL and Other Files

- Type command `asant generate-wsdl`
  - Runs command “wscompile -define -mapping build/mapping.xml -d build -nd build -classpath build config-interface.xml”

- This command generates these files in `build` directory
  - WSDL document (MyHelloService.wsdl)
  - Mapping file (mapping.xml)
    - Contains information that correlates the mapping between the Java interfaces and the WSDL definition
    - Portable
4. Package the Files into WAR File

- To package the files into WAR file, run command `asant create-war`
- This command generates `hello-jaxrpc.war` in directory `<INSTALL>/j2eetutorial14/examples/jaxrpc/helloservice`

5. Deploy the WAR File

- Make sure the Application Server is started
- Run `asant deploy-war`
  - `hello-jaxrpc.war` is deployed
- The tie classes (which are used to communicate with clients) are generated by the Application server during deployment
Verify Service

- Deploy as a Web application
- Verify the service from a browser


WSDL of the helloservice Service
Agenda

- Background on remote communication
- What is and Why JAX-RPC?
- Development steps of a JAX-RPC Service
- Type Mapping
- Client Programming
- Service Endpoint Model

Why Type Mapping?

- SOAP, WSDL do not define the mapping between XML and Programming language
  - SOAP and WSDL are designed to be programming language independent
  - Difference from CORBA world
- Yet, we need a standard way of mapping between the two
  - Otherwise, we will have interoperability problem
XML Data Types to Java Mapping

- Simple built-in type
  - `xsd:string` to `java.lang.String`
- Array
  - Mapped into a Java array
- Enumeration into a simple built-in type
  - Mapped into an enumeration Java class
- XML Struct and Complex type
  - Mapped into JavaBeans with getter and setter methods

Example: XML Struct to Java Mapping

```xml
<element name="Book"/>
<complexType>
  <all>
    <element name="author" type="xsd:string"/>
    <element name="preface" type="xsd:string"/>
    <element name="price" type="xsd:float"/>
  </all>
</complexType>
```

```java
public class Book implements java.io.Serializable {

  // ...
  public String getAuthor() { ... }
  public void setAuthor(String author) { ... }
  public String getPreface() { ... }
  public void setPreface(String preface) { ... }
  public float getPrice() { ... }
  public void setPrice(float price) { ... }
}
```
Java to XML Type Mapping

- Mapping from the Java types to the XML data types
- Performed by the JAX-RPC runtime system
- Only JAX-RPC supported Java types can be passed as parameters and return values

Supported Types

- Subset of J2SE classes
- Collections
- Primitives
- Arrays
- Value types
- JavaBeans
Subset of J2SE classes

- java.math.BigDecimal, java.math.BigInteger
- java.net.URI
- java.util.Calendar, java.util.Date

Collections

- List
  - ArrayList, LinkedList, Stack, Vector
- Map
  - HashMap, Hashtable, Properties, TreeMap
- Set
  - HashSet, TreeSet
Primitives & Wrapper Classes

- boolean
- byte
- double
- float
- int
- long
- short

Arrays

- Arrays with members of supported JAX-RPC types
- Examples
  - int[]
  - String[]
  - BigDecimal[]
Value Types

- A value type is a class whose state may be passed between a client and remote service as a method parameter or return value

Example

- Book class which contains the fields Title, Author, and Publisher

Rules for Value Type

- It must have a public default constructor
- It must not implement the java.rmi.Remote interface
  - Because SOAP does not support “value by reference parameters”
- It fields must be supported by JAX-RPC types
- A public field cannot be final or transient
- A non-public field must have corresponding getter and setter methods
Example: Value Types

```java
public class MeetingInfo {
    // private fields
    private String id;

    // public fields - does not need getter
    // setter methods
    public String address;

    // has to have getter and setter for
    // non-public fields
    public String getID() { ...}
    public void setID(String id) {...}
}
```

JavaBeans

- Must follow the same rules for Value types
- Must have a getter and setter method for each bean property
- The type of the bean property must be a supported JAX-RPC type
Example: JavaBean Type

```java
public class AddressBean implements java.io.Serializable {
    private String street; private String city; private String state; private String zip;

    public AddressBean() {
    }

    public AddressBean(String street, String city) {
        this.street = street;
        this.city = city;
    }

    public String getStreet() { return street; }
    public void setStreet(String street) { this.street = street; }

    public String getCity() { return city; }
    public void setCity(String city) { this.city = city; }

    ...}
```

WSDL & JAX-RPC

- Services are described using WSDL
- WSDL is the only contract needed between service provider and client
- JAX-RPC tools in J2EE 1.4 SDK
  - `wscompile` tool creates “abstract part of WSDL” from Service definition interface (server side)
  - `wsdeploy` tool creates “complete WSDL” including port address and tie classes (server side)
  - `wscompile` tool creates stubs (client side)
JAX-RPC Relationship to WSDL

JAX-RPC describes a Web Service as a collection of remote interfaces and methods.

Tools are used to convert between WSDL documents and sets of Java remote interfaces (example: wscompile).

WSDL describes a Web Service as a collection of ports and operations.

From the JAX-RPC API to WSDL and Back

- The JAX-RPC API specifies all the details of the mapping to/from WSDL.

<table>
<thead>
<tr>
<th>JAX-RPC</th>
<th>WSDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>service interface</td>
<td>&lt;service&gt;</td>
</tr>
<tr>
<td>remote interfaces</td>
<td>&lt;port&gt;'s/&lt;portType&gt;'s</td>
</tr>
<tr>
<td>methods</td>
<td>&lt;operation&gt;'s</td>
</tr>
<tr>
<td>value types</td>
<td>complex types (schema)</td>
</tr>
<tr>
<td>exceptions</td>
<td>faults</td>
</tr>
</tbody>
</table>
WSDL to Java Mapping Rules

- A WSDL document into a Java Package
- Abstract part of WSDL into Java interfaces and classes
  - `wsdl:portType`, `wsdl:operation`, `wsdl:message`
- Concrete binding part of WSDL into Java representation
  - `wsdl:binding`, `wsdl:port`, `wsdl:service`

WSDL portType/operation/message

- A `wsdl:portType` maps into a Java interface (Service Definition Interface) that extends `java.rmi.Remote`
- A `wsdl:operation` is mapped into a method of the Service definition interface
- `wsdl:message's` are mapped into parameters of the method
- `wsdl:type's` of `wsdl:message's` are mapped into the types of the parameters
Example: Mapping of WSDL portType to Java Service Definition Interface

```xml
<!------------------- WSDL Document ------------------->
<message name="GetLastTradePriceInput">
  <part name="tickerSymbol" type="xsd:string"/>
</message>
<message name="GetLastTradePriceOutput">
  <part name="result" type="xsd:float"/>
</message>
<portType name="StockQuoteProvider">
  <operation name="GetLastTradePrice" parameterOrder="tickerSymbol">
    <input message="tns:GetLastTradePriceInput"/>
    <output message="tns:GetLastTradePriceOutput"/>
  </operation>
</portType>

///////////////////////////////////////////////////////////////////// Java Interface 
public interface StockQuoteProvider extends java.rmi.Remote {
  float getLastTradePrice(String tickerSymbol)
    throws java.rmi.RemoteException;
}
```

WSDL binding/port/service (1/2)

- `wsdl:service` is mapped into an implementation of `javax.xml.rpc.Service` interface
- JAX-RPC runtime provides the implementation
WSDL binding/port/service (2/2)

- A `javax.xml.rpc.Service` class acts as a factory of:
  - Instance of a generated stub class
  - Dynamic proxy for a service port
  - Instance of the type `javax.xml.rpc.Call` for the dynamic invocation of a remote operation on a service port

Example: WSDL to binding, port, service

```xml
<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>
  </soap:binding>
</binding>

<service name="StockQuoteService">
  <documentation>My first service</documentation>
  <port name="StockQuotePort" binding="tns:StockQuoteBinding">
    <soap:address location="http://example.com/stockquote"/>
  </port>
</service>
```
**javax.xml.rpc.Service**

```java
package javax.xml.rpc;
public interface Service {
    public java.rmi.Remote getPort(QName portName,
                                     Class proxyInterface)
        throws JAXRPCException;
    public Call createCall(QName portName)
        throws JAXRPCException;
    public Call createCall(QName portName, String
                           operationName)
        throws JAXRPCException;
    public Call createCall() throws JAXRPCException;
    public java.net.URL getWSDLDocumentLocation();
    public QName getServiceName();
    public java.util.Iterator getPorts();
}
```

---

**Typical WSDL to Java Mapping Tool**

- Read WSDL document and then generates:
  - Service interface (javax.xml.rpc.Service) and its implementation
  - Service definition interface (Extension of java.rmi.Remote)
  - Stub and tie classes
  - Additional classes
    - Serializers, deserializers
SOAP Binding in WSDL

- JAX-RPC supports SOAP binding specific in WSDL 1.1
  - rpc and document style operations
  - literal and encoded representations
- Mapping of literal message part (either a parameter or return value)
  - Using Java data binding API: JAXB API
  - Mapping to SOAPElement as a document fragment
- Faults are mapped to Java exceptions

SOAP Message with Attachments

- RPC request or response can include MIME encoded content. Examples:
  - XML document or image
- JAX-RPC specifies mapping between MIME types and Java types:
  - image/gif, image/jpeg, text/plain, multipart/*, text/xml and application/xml
- Use of Java Activation Framework's `DataHandler` API
Java API for XML-based Remote Procedure Call (JAX-RPC)

Agenda

- Background on remote communication
- What is and Why JAX-RPC?
- Development steps of a JAX-RPC Service
- Type Mapping
- Client Programming
- Service Endpoint Model

JAX-RPC Client Environment

- Independent of how an XML based RPC service (service endpoint) is implemented on the server side
- Generates a Java based client side representation for a service from WSDL document
- Must not be exposed or tied to a specific XML based protocol, transport or any JAX-RPC implementation specific mechanism
- Can use either J2SE or J2EE programming model
Client Service Invocation Programming Models

- **Stub-based**
  - Both Interface and implementation are created at compile time

- **Dynamic proxy**
  - Interface is created at compile time while implementation created at runtime

- **Dynamic invocation interface (DII)**
  - Both interface and implementation are created at runtime

---

Stub-based Invocation Model

- Stub class gets **generated** from WSDL at compile time
- All needed value classes are also generated
- Instantiated using generated Service class
- Stub class is bound to a specific XML protocol (i.e. SOAP) and transport (i.e. HTTP)
- Static compilation gives maximum performance
- Stub class implements
  - `javax.xml.rpc.Stub` interface
  - **Service Definition Interface**
Steps of Coding Static Stub Client

- Creates a Stub object
  - (Stub)(new MyHelloService_Impl().getHelloIFPort())
- Sets the endpoint address that the stub uses to access the service
  - stub._setProperty(javax.xml.rpc.Stub.ENDPOINT_ADDRESS, SPROPERTY, args[0]);
- Casts stub to the service endpoint interface, HelloIF
  - HelloIF hello = (HelloIF)stub

Stub Configuration

- Stub instance must be configured
  - XML protocol binding
  - endpoint address
- Can be configured in two ways
  - Static configuration based on the WSDL description of a target service endpoint
    - wsdl:binding, soap:binding, wsdl:port
  - Runtime configuration using the javax.xml.rpc.Stub API
Standard Properties for Stub Configuration

- Username for authentication (required)
- Password for authentication (required)
- Target service endpoint address (optional)
- Flag for "session enabled" (required)

Static Stub Based Client

```java
package staticstub;
import javax.xml.rpc.Stub;
public class HelloClient {
    private String endpointAddress;
    public static void main(String[] args) {
        try {
            Stub stub = createProxy();
            stub._setProperty
                (javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY,
                 args[0]);
            HelloIF hello = (HelloIF)stub;
            System.out.println(hello.sayHello("Duke!"));
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    private static Stub createProxy() {
        // Note: MyHelloService_Impl is implementation-specific.
        return (Stub) (new MyHelloService_Impl().getHelloIFPort());
    }
}
```
Compile and Run Static Client

- Type commands
  - asant build
  - asant run-client
- You then will get

```
Command Prompt
C:\sun\j2etutorial14\examples\jaxrpc\staticstub>asant run-client
buildfile: build.xml
run-client:
[java] Endpoint address = http://localhost:8080/hello-jaxrpc/hello
[java] Hello Duke!
BUILD SUCCESSFUL
Total time: 3 seconds
C:\Sun\j2etutorial14\examples\jaxrpc\staticstub>
```

Dynamic Proxy-based Invocation Model

- Stubs are generated on the fly by JAXRPC client runtime
- Application provides the service definition interface the dynamic proxy conforms to
- Does not depend on implementation specific class
**Steps of Coding Dynamic Proxy Client**

1. Creates a Service object
2. Create a proxy with a type of the service endpoint interface

---

**Step1: Create a Service object**

```java
Service helloService = 
    serviceFactory.createService(helloWsdlUrl, 
    new QName(nameSpaceUri, serviceName));
```

- Service object is a factory for proxies
- Service object itself is created from ServiceFactory object
- Parameters of createService()
  - URL of the WSDL file
  - QName object
Step 2: Create a Dynamic Proxy object

dynamicproxy.HelloIF myProxy =
    (dynamicproxy.HelloIF)helloService.getPort(
        new QName(nameSpaceUri, portName),
        dynamicproxy.HelloIF.class);

- HelloIF class is generated by *wscompile*
- The port name (HelloIFPort) is specified by the WSDL file

Dynamic Proxy Client (1/2)

```java
package dynamicproxy;
import java.net.URL;
import javax.xml.rpc.Service;
import javax.xml.rpc.JAXRPCException;
import javax.xml.namespace.QName;
import javax.xml.rpc.ServiceFactory;
import dynamicproxy.HelloIF;
public class HelloClient {
    public static void main(String[] args) {
        try {
            String UrlString = args[0] + "?WSDL";
            String nameSpaceUri = "urn:Foo";
            String serviceName = "MyHelloService";
            String portName = "HelloIFPort";
            System.out.println("UrlString = " + UrlString);
            URL helloWsdlUrl = new URL(UrlString
```
Dynamic Proxy Client (2/2)

```
ServiceFactory serviceFactory =
    ServiceFactory.newInstance();
Service helloService =
    serviceFactory.createService(helloWsdlUrl,
        new QName(nameSpaceUri, serviceName));
dynamicproxy.HelloIF myProxy =
    (dynamicproxy.HelloIF)
    helloService.getPort(
        new QName(nameSpaceUri, portName),
        dynamicproxy.HelloIF.class);
System.out.println(myProxy.sayHello("Buzz"));
} catch (Exception ex) {
    ex.printStackTrace()
}
```
DII Invocation Model (1/2)

- Gives complete control to client programmer
- A client can call a remote procedure even if the signature of the remote procedure or the name of the service are unknown until runtime
- Does not require wscompile to create runtime classes
- Most complex programming among the three

DII Invocation Model (2/2)

- Enables broker model
  - Client finds (through some search criteria) and invokes a service during runtime through a broker
  - Used when service definition interface is not known until runtime
  - You set operation and parameters during runtime
Steps of Coding DII Client

1. Create a Service object
2. From the Service object, create a Call object
3. Set the service endpoint address on the Call object
4. Set properties on the Call object
5. Specify the method's return type, name, and parameter
6. Invoke the remote method on the Call object

Step1: Create a Service object

- Invoke createService() method of a ServiceFactory object
  
  ```java
  Service service = 
  factory.createService(new QName(qnameService));
  ```

- qnameService parameter is the name of the service specified in WSDL
  
  ```xml
  <service name="MyHelloService">
  ```
Step 2: From the Service object, create a Call object

- A Call object supports the dynamic invocation of the remote procedures of a service
  
  ```java
  QName port = new QName(qnamePort);
  Call call = service.createCall(port);
  ```

- The parameter of createCall is a QName object that represents the service endpoint interface, which is specified in WSDL
  
  ```xml
  <portType name="HelloIF">
  ```

Step 3: Set the service endpoint address on the Call object

- In the WSDL file, this address is specified by the `<soap:address>` element
  
  ```java
  call.setTargetEndpointAddress(endpoint);
  ```
Step 4: Specify the method's return type, name, and parameter

- **Properties to set**
  - SOAPACTION_USE_PROPERTY
  - SOAPACTION_URI_PROPERTY
  - ENCODING_STYLE_PROPERTY

Step 5: Specify the method's return type, name, and parameter

- **Return type, method name, parameter**
  
  ```java
  QName QName_TYPE_STRING = new QName(NS_XSD, "string");
  call.setReturnType(QNAME_TYPE_STRING);
  
  call.setOperationName(new QName(BODY_NAMESPACE_VALUE, "sayHello"));
  
  call.addParameter("String_1", QName_TYPE_STRING, ParameterMode.IN);
  ```
Step 6: Invoke the remote method on the Call object

- Assign the parameter value (Murphy) to a String array (params) and then executes the invoke method with the String array as an argument

```java
String[] params = { "Murphy" };
String result = (String)call.invoke(params);
```

Example: DII Client (1/3)

```java
package dii;
import javax.xml.rpc.Call;
import javax.xml.rpc.Service;
import javax.xml.rpc.JAXRPCException;
import javax.xml.namespace.QName;
import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.ParameterMode;
public class HelloClient {
    private static String qnameService = "MyHelloService";
    private static String qnamePort = "HelloIF";
    private static String BODY_NAMESPACE_VALUE = "urn:Foo";
    private static String ENCODING_STYLE_PROPERTY = "javax.xml.rpc.encodingstyle.namespace.uri";
    private static String NS_XSD = "http://www.w3.org/2001/XMLSchema";
    private static String URI_ENCODING = "http://schemas.xmlsoap.org/soap/encoding/";
}
```
Example: DII Client (2/3)

```java
public static void main(String[] args) {
    System.out.println("Endpoint address = " + args[0]);
    try {
        ServiceFactory factory = ServiceFactory.newInstance();
        Service service = factory.createService(new QName(qnameService));
        QName port = new QName(qnamePort);
        Call call = service.createCall(port);
        call.setTargetEndpointAddress(args[0]);
        call.setProperty(Call.SOAPACTION_USE_PROPERTY, new Boolean(true));
        call.setProperty(Call.SOAPACTION_URI_PROPERTY, "");
        call.setProperty(ENCODING_STYLE_PROPERTY, URL_ENCODING);
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

Example: DII Client (3/3)

```java
QName QName_TYPE_STRING = new QName(NS_XSD, "string");
call.setReturnType(QNAME_TYPE_STRING);
call.setOperationName(new QName(BODY_NAMESPACE_VALUE,"sayHello"));
call.addParameter("String_1", QName_TYPE_STRING, ParameterMode.IN);
String[] params = { "Murph!" };
String result = (String)call.invoke(params);
System.out.println(result);
}
```
Compile and Run DII Client

- Type commands
  - asant build
  - asant run-client
- You then will get

```
c:j2eetutorial\examples\jaxrpc\dii asant run-client
Buildfile: build.xml

run-client:
  [java] Endpoint address = http://localhost:8080/hello-jaxrpc/hello
  [java] Hello Murph!

BUILD SUCCESSFUL
Total time: 1 second

c:j2eetutorial\examples\jaxrpc\dii
```

Agenda

- Background on remote communication
- What is and Why JAX-RPC?
- Development steps of a JAX-RPC Service
- Type Mapping
- Client Programming
- Service Endpoint Model
Service Endpoint Model

- Service endpoint can be either
  - Servlet based endpoint or
  - Stateless session bean
- JAX-RPC 1.0 specifies Servlet based endpoint model
- EJB 2.1 specifies Stateless session bean based endpoint model

Web Services for the J2EE 1.4 Platform

- Client View
  - JAX-RPC
- Server View
  - Servlet based endpoint
    - JAX-RPC
    - Runtime is provided by Web container
  - Stateless Session Bean based endpoint
    - EJB 2.1
    - Runtime is provided by EJB container
### Service Developer
- Generates service definition interface
- Implements service definition interface (service endpoint class)
- Service endpoint class
  - May implement `ServiceLifecycle` interface
  - Can access servlet context via `javax.servlet.ServletContext` interface
- Creates `*.war` package

### Service Deployer
- Handles
  - Protocol binding
  - Port assignment
- Multiple protocol binds for a single service endpoint definition
- Creates `web.xml`
- Creates complete WSDL document which contains concrete binding information
  - This WSDL document can be published
- Creates and deploys assembled `*.war` file
References

- JAX-RPC Home

- Java Web Services Developer Pack Download

- Java Web Services Developer Pack Tutorial

- J2EE 1.4 SDK
  - http://java.sun.com/j2ee/1.4/download-dr.html

- Web Services Course Programming Page
  - http://www.javapassion.com/webservices