Web Services

- Web services are a middleware, like CORBA and RMI.
- What makes web services unique is that the language being used is XML.
- This is good for several reasons:
  - Debugging is possible using a text editor
  - Platform independence, since XML is just text
  - Development of web services middleware is simple, since XML is easy to parse & generate
SOAP

- The XML language used by web services is called SOAP
  - The Simple Object Access Protocol
- Other middleware technologies use different communication languages
  - IIOP is the wire protocol used by CORBA
  - JRMP is the wire protocol used by RMI
  - These languages are binary, however

SOAP

- Simple Object Access Protocol
  - SOAP is the language used (over HTTP) for web services requests and responses
  - This is much like how HTML is used for traditional web responses
- SOAP is usually hidden from developers, but developers may examine the SOAP code for debugging purposes
  - SOAP is easy to understand since it is XML
How SOAP Works

- The client sends a SOAP message (called a SOAP envelope) using HTTP to the service.
- The service is installed into a web server, which receives the HTTP request and passes the message to the service.
- The service processes the requested operation, and returns another SOAP envelope as the HTTP response.
- The client reads and interprets the response SOAP envelope, and reacts to it somehow.

How SOAP Works (Contd.)
How SOAP Works (Contd.)

- One major advantage that SOAP has is that it uses HTTP, which is a protocol already prevalent on the web
  - It is used for web browser/web server communication
- Typically, SOAP clients, however, are not users, but other programs
  - SOAP clients are typically similar in nature to CORBA or RMI clients

Example: SOAP Request

```
POST /InStock HTTP/1.1
Host: www.stock.org
Content-Type: application/soap+xml;charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
  soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Body xmlns:m="http://www.stock.org/stock">
    <m:GetStockPrice>
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```
Web Services

- Many people think that web services will soon become ubiquitous
  - Mostly, these people are betting that Microsoft’s support will be enough to convince everyone to switch their middleware
- Whether or not they become ubiquitous, eventually they will cooperate with CORBA & EJBs
  - This will mean that they can be used in any application
- Many Java gurus/developers HATE XML and Web services.
  - Too slow, too high an overhead, and “back to the future” because RPC is used and real objects can not be passed.
Web Service Middleware

- Most web service middleware prevents you from having to know the SOAP language
  - However, it is a good idea to know something about the language if you intend to use SOAP
- There are several middleware choices if you plan to use web services (and they are roughly compatible):
  - .NET Web Services
  - JAX-RPC
  - Apache Axis
- Each of these technologies are configured differently, but the main idea is the same

Java API for XML-Based RPC (JAX-RPC)

- JAX-RPC is a SOAP middleware that is provided by Sun Microsystems
  - JAX – Java API for XML
  - RPC – Remote Procedure Call
- With JAX-RPC, a Java Interface and its implementation are created (the interface inherits from Remote)
  - An XML configuration file is created to describe the web service
  - The `wscompile` command-line utility generates stubs and ties automatically, given an XML configuration file
In JAX-RPC, a remote procedure call is represented by an XML-based protocol such as SOAP. The SOAP specification defines the envelope structure, encoding rules, and conventions for representing remote procedure calls and responses.

These calls and responses are transmitted as SOAP messages (XML files) over HTTP. JAX-RPC API hides SOAP complexity from the application developer.

On the server side, the developer specifies the remote procedures by defining methods in an interface. The developer also codes one or more classes that implement the methods.

A client creates a proxy (a local object representing the service) and then simply invokes methods on the proxy.

With JAX-RPC, the developer does not generate or parse SOAP messages. JAX-RPC runtime converts the API calls and responses to and from SOAP messages.
JAX-RPC

- Stubs: Used as local classes by the client. For each method call, a SOAP request is generated. The SOAP response is then received and the response data is returned to the client.
- Ties: (also called skeletons) Accepts the SOAP requests from the stubs and calls the appropriate method locally, then returns a SOAP response.

Web Service Architecture

![Diagram of Web Service Architecture](image_url)

Figure 1: The process flow of a Web service.
Developing a JAX-RPC Web Service

- The starting point for developing a JAX-RPC Web service is the service endpoint interface. A service endpoint interface (SEI) is a Java interface that declares the methods that a client can invoke on the service.

- Use the SEI and the wscompile tool to generate the WSDL specification of the Web service and the stubs that connect a Web service client to the JAX-RPC runtime. The Application Server provides the Application Server’s implementation of JAX-RPC.

Summary of Steps

These are the basic steps for creating the Web service and client:
1. Code the SEI and implementation class and interface configuration file.
2. Compile the SEI and implementation class.
3. Use wscompile to generate the WSDL file.
4. Package the files into a WAR file.
5. Deploy the WAR file. The tie classes (which are used to communicate with clients) are generated by the Application Server during deployment.
6. Code the client class.
7. Use wscompile to generate and compile the stub files from the WSDL file.
8. Compile the client class.
9. Run the client.
A service endpoint interface declares the methods that a remote client may invoke on the service. In this example, the interface declares a single method named sayHello.

A service endpoint interface must conform to a few 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.
JAX-RPC Supported Types

- **Primitive types:** boolean byte double float int long short
- **Classes:**
  - `java.lang.Boolean`
  - `java.lang.Byte`
  - `java.lang.Double`
  - `java.lang.Float`
  - `java.lang.Integer`
  - `java.lang.Long`
  - `java.lang.Short`
  - `java.lang.String`
  - `java.math.BigDecimal`
  - `java.math.BigInteger`
  - `java.net.URI`
  - `java.util.Calendar`
  - `java.util.Date`

- **java.util.Collection Classes:** List ArrayList LinkedList Stack Vector Map HashMap Hashtable Properties TreeMap Set HashSet TreeSet

JAX-RPC Value Types

- A **value type** is a class whose state may be passed between a client and remote service as a parameter or return value. For example, in an application for a university library, a client might call a remote procedure with a value type parameter named Book, a class that contains the fields Title, Author, and Publisher.

- A value type must have a public default constructor, not implement the `java.rmi.Remote` interface, and its fields must be supported JAX-RPC types.

- The value type may contain public, private, or protected fields:
  - A public field cannot be final or transient.
  - A non-public field must have corresponding getter and setter methods.
Hello Service Example: Service Interface

- In this example, the service endpoint interface is HelloIF.java:

```java
package helloservice;
import java.rmi.Remote;
import java.rmi.RemoteException;

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

Hello Service Example: Service Implementation

```java
package helloservice;

public class HelloImpl implements HelloIF
{
    public String message ="Hello";

    public String sayHello(String s)
    {
        return message + s;
    }
}
```
Building the Service

- compile HelloIF.java and HelloImpl.java, writing the class files to the build subdirectory.
  
  `Javac .\helloservice\*.java –d build`

- Run wscompile, which creates the WSDL and mapping files (MyHelloService.wsdl and mapping.xml). The WSDL file describes the Web service and is used to generate the client stubs. The mapping file contains information that correlates the mapping between the Java interface and the WSDL definitions.
  
  `wscompile -define -mapping build/mapping.xml -d build -nd build -classpath build config-interface.xml`

Building the Service (Contd.)

- The `-classpath` flag instructs wscompile to read the SEI in the build directory, and the `-define` flag instructs wscompile to create WSDL and mapping files. The `-mapping` flag specifies the mapping file name. The `-d` and `-nd` flags tell the tool to write class and WSDL files to the build subdirectory.

- The `wscompile` tool reads an interface configuration file that specifies information about the SEI. In this example, the configuration file is named config-interface.xml and contains the following:

```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>
```
Building the Service (Contd.)

- This configuration file tells wscompile to create a WSDL file named MyHelloService.wsdl with the following information:
  - The service name is MyHelloService.
  - The WSDL target and type namespace is urn:Foo. The choice for what to use for the namespaces is up to you. The role of the namespaces is similar to the use of Java package names—to distinguish names that might otherwise conflict. For example, a company can decide that all its Java code should be in the package com.wombat.*. Similarly, it can also decide to use the namespace http://wombat.com.
  - The SEI is helloservice.HelloIF.
  - The packageName attribute instructs wscompile to put the service classes into the helloservice package.

Coding the Client

- Before invoking the remote methods on the stub, the client must:
  1. Create a Stub object:
     ```java
     (Stub)(new MyHelloService_Impl().getHelloIFPort())
     ```
     The MyHelloService_Impl stub class will be generated by wscompile.
  2. Set the endpoint address that the stub uses to access the service:
     ```java
     stub._setProperty(javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY, args[0]);
     ```
  3. Cast stub to the service endpoint interface, HelloIF:
     ```java
     HelloIF hello = (HelloIF)stub;
     ```
Coding the Client (contd.)

```java
package helloclient;

import javax.xml.rpc.Stub;

public class HelloClient
{
    private String endpointAddress;

    public static void main(String[] args)
    {
        System.out.println("Endpoint address = " + args[0]);
        try
        {
            Stub stub = createProxy();
            stub._setProperty(javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY, args[0]);
            HelloIF hello = (HelloIF)stub;
            System.out.println(hello.sayHello("World"));
        }
        catch(Exception ex)
        {
            ex.printStackTrace();
        }
    }

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

Building the Client

1. Generate service stub as follows:
   ```bash
   wscompile -gen:client -d build -classpath build config-wsdl.xml
   ```
   The wscompile command generates files based on the information in the WSDL file and on the command-line flags.
   - The `-gen:client` flag instructs `wscompile` to generate the stubs, serializers, and value types.
   - The `wscompile` tool reads a WSDL configuration file that specifies the location of the WSDL file. In this example, the configuration file is named `config-wsdl.xml`, and it contains the following:
     ```xml
     <?xml version="1.0" encoding="UTF-8"?>
     <configuration xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
     <wsdl location="http://localhost:8080/hello-jaxrpc/hello?WSDL" packageName="helloclient"/>
     </configuration>
     ```

2. Compile client:
   ```bash
   javac .\helloclient\*.java -classpath C:\Sun\AppServer\lib\j2ee.jar;C:\Sun\AppServer\lib\jaxrpc-api.jar;C:\Sun\AppServer\lib\jaxrpc-impl.jar;build -d build
   ```

3. Run client:
   ```bash
   java -cp .;C:\Sun\AppServer\lib\j2ee.jar;C:\Sun\AppServer\lib\saaj-impl.jar;C:\Sun\AppServer\lib\mail.jar helloclient.HelloClient http://localhost:8080/hello-jaxrpc/hello
   ```
Java API for XML-Based Registries (JAXR)

- JAXR enables Java software programmers to use a single API to access a variety of XML registries. A unified JAXR information model describes content and metadata within XML registries.
- Developers can write registry client programs that are portable across different target registries.
- The JAXR specification includes detailed bindings between the JAXR information model and both the electronic business XML Registry (ebXML) and the Universal Description, Discovery, and Integration (UDDI version 2) specifications for Business to Business (B2B) application development.
Demo the process of developing and consuming WebServices in NetBeans
Reference