Contents

**Chapter 1: Introduction**

[1.1 INTRODUCTION: 4](#_Toc251633160)

[1.2 Objectives: 5](#_Toc251633161)

[1.3 Project outline : 5](#_Toc251633162)

[1.5 General Requirements 5](#_Toc251633163)

**Chapter 2: Servlet And JSP**

[2.1 Introduction: 8](#_Toc251633164)

[2.2 Servlet Application Architecture 8](#_Toc251633165)

[2.3 How a Servlet Works: 8](#_Toc251633166)

[2.4 A Servlet's Life Cycle: 9](#_Toc251633168)

[2.5 Demonstrating the Life Cycle of a Servlet: 11](#_Toc251633169)

[2.6 Requests and Responses: 12](#_Toc251633170)

[2.7 The ServletRequest Interface 12](#_Toc251633171)

[2.8 The ServletResponse Interface 13](#_Toc251633173)

[2.9 The GenericServlet Wrapper Class: 14](#_Toc251633175)

[2.10 JSP Basics: 15](#_Toc251633176)

[2.11 What's Wrong with Servlets? 15](#_Toc251633177)

[2.12 Writing a JSP File: 16](#_Toc251633178)

[2.13 How JSP Works: 17](#_Toc251633179)

[2.13.1 The JSP Servlet Generated Code: 18](#_Toc251633180)

[2.14 The JSP API 18](#_Toc251633181)

[2.14.1The JspPage Interface 18](#_Toc251633182)

[2.14.2 The HttpJspPage Interface 19](#_Toc251633183)

[2.14.3The JspFactory Class 19](#_Toc251633184)

[2.14.4 The JspEngineInfo Class 20](#_Toc251633185)

[2.14.5 The PageContext Class 20](#_Toc251633186)

[2.14.6 The JspWriter Class 21](#_Toc251633187)

[2.15 JSP Syntax: 21](#_Toc251633188)

[2.15.1 Directives 22](#_Toc251633189)

[2.15.2 The Page Directive 22](#_Toc251633190)

**Chapter 3: Web Service And JAX-RPC**

[3.1 Introduction 26](#_Toc251633191)

[3.2 Web Services Technology 26](#_Toc251633192)

[3.2.1 Service Transport 27](#_Toc251633193)

[3.2.2 Service Messaging 27](#_Toc251633194)

[3.2.3 Service Description 28](#_Toc251633195)

[3.2.4 Service Registry 29](#_Toc251633196)

[ Java and Web Services 29](#_Toc251633197)

[3.3 JAX-RPC Service Model 30](#_Toc251633198)

[Fig3.2 JAX-RPC Service Model 31](#_Toc251633199)

[3.4 Data Types and Serialization 33](#_Toc251633200)

[3.4.1 Java-to-XML Marshalling 35](#_Toc251633201)

[3.5 JAX-RPC Development 38](#_Toc251633202)

[3.5.1 Service Definition 39](#_Toc251633203)

[3.5.2 Service Implementation 40](#_Toc251633204)

[3.5.3 Service Deployment 41](#_Toc251633205)

[3.5.4 Service Description 43](#_Toc251633206)

[3.5.5 Service Consumption 46](#_Toc251633207)

Chapter 4: Methology And System analysis and design

[4.1 INTRODUCTION 49](#_Toc251633208)

[4.2 How to Build a Successful News System: 50](#_Toc251633209)

[4.3 Design Overview: 52](#_Toc251633211)

[4.4 Object orieneted model: 52](#_Toc251633212)

[4.4.1 Data Flow Diagram: 52](#_Toc251633213)

[44.2 Use Case Diagram: 55](#_Toc251633214)

[References 59](#_Toc251633215)

Chapter One

Introduction

# 1.1 INTRODUCTION:

Driven by the explosive growth of information available on the Internet, intelligent information access has become a central research area in computer science. The 20th century is commonly characterized as “The Information Age”, and the sheer amount of information readily available today has created novel challenges. Numerous intelligent information agents software tools that provide personalized assistance for users navigating large information spaces -- have been described in the literature and deployed on the World-Wide-Web However, the need for intelligent information agents is not limited to web-based applications, as we are currently witnessing an increasing trend towards “ubiquitous information access”. Different types of wireless information devices, designed to tap into the Internet’s vast information resources without physical constraints, are currently being released into the marketplace. For example, cell phones can access Internet-based information services, and pagers can alert users of late-breaking news. While these devices undoubtedly enhance the utility of online information and are likely to open up opportunities for revolutionary information centric applications, they are cramped by several technical constraints. First, the small size of wireless information devices leads to inherently limited user interfaces. Second, bandwidth constraints impose limits on the amount of information to be transferred. Third .The World-Wide-Web is currently witnessing an ongoing trend towards personalized information access. As part of this trend, numerous personalized news services are emerging. For example, Internet portals such as Yahoo, Lycos and Excite offer personalized access to daily news stories from a large range of categories. These services are based on static questionnaires that users fill out in order to make use of news filtering capabilities. We believe that this level of personalization is not fine-grained enough for price- and bandwidth-sensitive information access. Here, we suggest the use of an intelligent news agent that unobtrusively learns about a user’s interests in daily news stories by observing the user’s browsing behavior. Using a content-based machine learning algorithm originally developed for a web-based client ,the agent learns to rank-order news stories with respect to the user’s individual interests.

# 1.2 Objectives:

* Provide a brief overlook of the importance and growth of News System in the latest couple of years.
* Provide a clear categorization of the target system according to Customer.
* Provide a fully functional system analysis of the target system
* Provide an adequate explanation of the implemented techniques used to achieve the final system.
* Provide a brief overlook of web service using JAX-RPC.
* Provide a brief overlook of web client using JSP .
* Provide a brief overlook of client using J2SE.
* Provide a brief overlook of mobile client using j2me.

# 1.3 Project outline :

The following three chapters will be divided as follows:

**Chapter 2:Servlet And JSP.**

In this chapter we will define Servlet and JSP and how its work.

**Chapter 3:WebService.**

In this chapter we well define web service and define JAX-RPC

**Chapter 4:Methology And System analysis and design .**

In this chapter we well describe the Methology of news system and we will design this system.

# 1.5 General Requirements

The system consists of a server that handles a database containing newsgroups and articles, and a client that accepts commands from the user and communicates with the server. Several clients may be connected to the server simultaneously. The user can perform the following tasks:

\_ List all newsgroups.

\_ Create and delete newsgroups.

\_ List articles in a newsgroup.

\_ Read, write and delete articles in a newsgroup.

The system keeps track of the title and the author of each article. It cannot handle subject.threads and follow-up articles.

**Requirements on the client:**

\_ The clients reads commands from the terminal, communicates with the server and presents the replies from the server on the terminal.

\_ The client is easy to use and contains informative help texts. No manual is necessary to use the client program.

\_ The client tries to handle all errors. If it cannot recover from an error, it terminates with an informative error message.

# 

Chapter Two

Servlet And JSP

# 2.1 Introduction:

The servlet technology is the foundation of web application development using the Java programming language. It is one of the most important Java technologies, and it is the underlying technology for another popular Java technology for web application development: JavaServer Pages (JSP). Therefore, understanding the servlet technology and its architecture is important if you want to be a servlet developer.Even if you plan to develop your Java web application using JSP pages alone,understanding the servlet technology helps you build a more efficient and effective JSP application.

# 2.2 Servlet Application Architecture

A servlet is a Java class that can be loaded dynamically into and run by a special web server. This servlet-aware web server is called a servlet container, which also was called a servlet engine in the early days of the servlet technology .Servlets interact with clients via a request-response model based on HTTP. Because servlet technology works on top of HTTP, a servlet container must support HTTP as the protocol for client requests and server responses. However, a servlet container also can support similar protocols, such as HTTPS (HTTP over SSL) for secure transactions.





Fig 2.1 Servlet Application Architecture

# 2.3 How a Servlet Works:

A servlet is loaded by the servlet container the first time the servlet is requested. The servlet then is forwarded the user request, processes it, and returns the response to the servlet container, which in turn sends the response back to the user. After that, the servlet stays in memory waiting for other requests—it will not be unloaded from the memory unless the servlet container sees a shortage of memory. Each time the servlet is requested, however, the servlet container compares the timestamp of the loaded servlet with the servlet class file. If the class file timestamp is more recent, the servlet is reloaded into memory. This way, you don't need to restart the servlet container every time you update your servlet.



# Fig 2.2 How a Servlet Works

# 2.4 A Servlet's Life Cycle:

Let there be servlet. This interface in the javax.servlet package is the source of all activities in servlet programming. Servlet is the central abstraction of the Java servlet technology. Every servlet you write must implement this javax.servlet.Servlet interface, either directly or indirectly. The life cycle of a servlet is determined by three of its methods: init, service, and destroy.

#### The init( ) Method

The init method is called by the servlet container after the servlet class has been instantiated. The servlet container calls this method exactly once to indicate to the servlet that the servlet is being placed into service. The init method must complete successfully before the servlet can receive any requests.

You can override this method to write initialization code that needs to run only once, such as loading a database driver, initializing values, and so on. In other cases, you normally leave this method blank.

The signature of this method is as follows:

public void init(ServletConfig config) throws ServletException

#### The service( ) Method

The service method is called by the servlet container after the servlet's init method to allow the servlet to respond to a request.

Servlets typically run inside multithreaded servlet containers that can handle multiple requests concurrently. Therefore, you must be aware to synchronize access to any shared resources, such as files, network connections, and the servlet's class and instance variables. For example, if you open a file and write to that file from a servlet, you need to remember that a different thread of the same servlet also can open the same file" for more on the topic of multithreading and synchronization.

This method has the following signature:

public void service(ServletRequest request, ServletResponse response)

throws ServletException, java.io.IOException

The servlet container passes a ServletRequest object and the ServletResponse object. The ServletRequest object contains the client's request and the ServletResponse contains the servlet's response. These two objects are important because they enable you to write custom code that determines how the servlet services the client request.

The service method throws a ServletException if an exception occurs that interferes with the servlet's normal operation. The service method also can throw a java.io.IOException if an input or output exception occurs during the execution of this method. As the name implies, the service method exists so that you can write code that makes the servlet function the way it is supposed to.

#### The destroy( ) Method

The servlet container calls the destroy method before removing a servlet instance from service. This normally happens when the servlet container is shut down or the servlet container needs some free memory.

This method is called only after all threads within the servlet's service method have exited or after a timeout period has passed. After the servlet container calls this method, it will not call the service method again on this servlet.

The destroy method gives the servlet an opportunity to clean up any resources that are being held (for example, memory, file handles, and threads) and make sure that any persistent state is synchronized with the servlet's current state in memory.

The signature of this method is as follows:

public void destroy()

# 2.5 Demonstrating the Life Cycle of a Servlet:

contains the code for a servlet named PrimitiveServlet, a very simple servlet that exists to demonstrate the life cycle of a servlet. The PrimitiveServlet class implements javax.servlet.Servlet (as all servlets must) and provides implementations for all the five methods of servlet. What it does is very simple. Each time any of the init, service, or destroy methods is called, the servlet writes the method's name to the console.

After you compile the source code into the myApp\WEB-INF\classes directory, add the servlet to the web.xml under the name Primitive

##### The web.xml File for PrimitiveServlet

<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app

PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"

"http://java.sun.com/dtd/web-app\_2\_3.dtd">

<web-app>

<servlet>

<servlet-name>Primitive</servlet-name>

<servlet-class>PrimitiveServlet</servlet-class>

</servlet>

</web-app>

# 2.6 Requests and Responses:

Requests and responses are what a web application is all about. In a servlet application, a user using a web browser sends a request to the servlet container, and the servlet container passes the request to the servlet.

In a servlet paradigm, the user request is represented by the ServletRequest object passed by the servlet container as the first argument to the service method. The service method's second argument is a ServletResponse object, which represents the response to the user.

# 2.7 The ServletRequest Interface

The ServletRequest interface defines an object used to encapsulate information about the user's request, including parameter name/value pairs, attributes, and an input stream.

The ServletRequest interface provides important methods that enable you to access information about the user. For example, the getParameterNames method returns an Enumeration containing the parameter names for the current request. To get the value of each parameter, the ServletRequest interface provides the getParameter method.

The getRemoteAddress and getRemoteHost methods are two methods that you can use to retrieve the user's computer identity. The first returns a string representing the IP address of the computer the client is using, and the second method returns a string representing the qualified host name of the computer.



# Fig 2.3 ServletRequest Interface

# 2.8 The ServletResponse Interface

The ServletResponse interface represents the response to the user. The most important method of this interface is getWriter, from which you can obtain a java.io.PrintWriter object that you can use to write HTML tags and other text to the user.



# Fig 2.4 ServletResponse Interface

# 2.9 The GenericServlet Wrapper Class:

Throughout this chapter, we have been creating servlet classes that implement the javax.servlet.Servlet interface. Everything works fine, but there are two annoying things that you've probably noticed:

1. You have to provide implementations for all five methods of the Servlet interface, even though most of the time you only need one. This makes your code look unnecessarily complicated.
2. The ServletConfig object is passed to the init method. You need to preserve this object to use it from other methods. This is not difficult, but it means extra work.

The javax.servlet package provides a wrapper class called GenericServlet that implements two important interfaces from the javax.servlet package: Servlet and ServletConfig, as well as the java.io.Serializable interface. The GenericServlet class provides implementations for all methods, most of which are blank. You can extend GenericServlet and override only methods that you need to use. Clearly, this looks like a better solution.

# 2.10 JSP Basics:

JavaServer Pages (JSP) is another Java technology for developing web applications. JSP was released during the time servlet technology had gained popularity as one of the best web technologies available. JSP is not meant to replace servlets, however. In fact, JSP is an extension of the servlet technology, and it is common practice to use both servlets and JSP pages in the same web applications.

Authoring JSP pages is so easy that you can write JSP applications without much knowledge of the underlying API. If you want to be a really good Java web programmer, however, you need to know both JSP and servlets. Even if you use only JSP pages in your Java web applications, understanding servlets is still very important. For example, in JSP you work with HTTP requests and HTTP responses, request parameters, request attributes, session management, cookies, URL-rewriting, and so on. This chapter explains the relation between JSP and servlets, introduces the JSP technology, and presents many examples that you can run easily.

# 2.11 What's Wrong with Servlets?

The history of web server-side programming in Java started with servlets. Sun introduced servlets in 1996 as small Java-based applications for adding dynamic content to web applications. Not much later, with the increasing popularity of Java, servlets took off to become one of the most popular technologies for Internet development today.

##### Displays All Parameter/Value Pairs in a Request Using a Servlet

Nearly half of the content sent from the doPost method is static HTML. However, each HTML tag must be embedded in a String and sent using the println method of the PrintWriter object. It is a tedious chore. Worse still, the HTML page may be much longer.

Another disadvantage of using servlets is that every single change will require the intervention of the servlet programmer. Even a slight graphical modification, such as changing the value of the <BODY> tag's BGCOLOR attribute from #DADADA to #FFFFFF, will need to be done by the programmer (who in this case will work under the supervision of the more graphic-savvy web designer).

Sun understood this problem and soon developed a solution. The result was JSP technology. According to Sun's web site, "JSP technology is an extension of the servlet technology created to support authoring of HTML and XML pages." Combining fixed or static template data with dynamic content is easier with JSP .What needs to be highlighted is that "JSP technology is an extension of the servlet technology." This means that JSP did not replace servlets as the technology for writing server-side Internet/intranet applications. In fact, JSP was built on the servlet foundation and needs the servlet technology to work.

JSP solves drawbacks in the servlet technology by allowing the programmer to intersperse code with static content, for example. If the programmer has to work with an HTML page template written by a web designer, the programmer can simply add code into the HTML page and save it as a .jsp file. If at a later stage the web designer needs to change the HTML body background color, he or she can do it without wasting the charging-by-the-hour programmer's time. He or she can just open the .jsp file and edit it accordingly.

You can see that <HTML> tags stay as they are. When you need to add dynamic content, all you need to do is enclose your code in <% … %> tags.

Again, JSP is not a replacement for servlets. Rather, JSP technology and servlets together provide an attractive solution to web scripting/programming by offering platform independence, enhanced performance, separation of logic from display, ease of administration, extensibility into the enterprise, and most importantly, ease of use.

# 2.12 Writing a JSP File:

A JSP page consists of interwoven HTML tags and Java code. The HTML tags represent the presentation part and the code produces the contents. In its most basic form, a JSP page can include only the HTML part.



Fig 2.5 jsp file interface

# 2.13 How JSP Works:

Inside the JSP container is a special servlet called the page compiler. The servlet container is configured to forward to this page compiler all HTTP requests with URLs that match the .jsp file extension. This page compiler turns a servlet container into a JSP container. When a .jsp page is first called, the page compiler parses and compiles the .jsp page into a servlet class. If the compilation is successful, the jsp servlet class is loaded into memory. On subsequent calls, the servlet class for that .jsp page is already in memory; however, it could have been updated. Therefore, the page compiler servlet will always compare the timestamp of the jsp servlet with the jsp page. If the .jsp page is more current, recompilation is necessary. With this process, once deployed, JSP pages only go through the time-consuming compilation process once.

You may be thinking that after the deployment, the first user requests for a .jsp page will experience unusually slow response due to the time spent for compiling the .jsp file into a jsp servlet. To avoid this unpleasant situation, a mechanism in JSP allows the .jsp pages to be pre-compiled before any user request for them is received. Alternatively, you deploy your JSP application as a web archive file in the form of a compiled servlet.

## 2.13.1 The JSP Servlet Generated Code:

When the JSP is invoked, Tomcat creates two files in the C:\%CATALINA\_HOME%\work\localhost\examples\jsp directory. Those two files are SimplePage\_ jsp.java and SimplePage\_ jsp.class. When you open the SimplePage\_ jsp.java file, you will see the following:

# 2.14 The JSP API

The JSP technology is based on the JSP API that consists of two packages: javax.servlet.jsp and javax.servlet.jsp.tagext. Both packages are given in detail in In addition to these two packages, JSP also needs the two servlet packages—javax.servlet and javax.servlet.http. When you study the javax.servlet.jsp package, you will know why we say that JSP is an extension of servlet technology and understand why it is important that a JSP application programmer understands the servlet technology well.The javax.servlet.jsp package has two interfaces and four classes. The interfaces are as follows:

* JspPage
* HttpJspPage

The four classes are as follows:

* JspEngineInfo
* JspFactory
* JspWriter
* PageContext

In addition, there are also two exception classes: JspException and JspError.

## 2.14.1The JspPage Interface

The JspPage is the interface that must be implemented by all JSP servlet classes. This may remind you of the javax.servlet.Servlet interface , "The Servlet Technology," of course. And, not surprisingly, the JspPage interface does extend the javax.servlet.Servlet interface.

The JSPPage interface has two methods, JspInit and JspDestroy, whose signatures are as follows:

public void jspInit()

public void jspDestroy()

jspInit, which is similar to the init method in the javax.servlet.Servlet interface, is called when the JspPage object is created and can be used to run some initialization. This method is called only once during the life cycle of the JSP page: the first time the JSP page is invoked.

The jspDestroy method is analogous with the destroy method of the javax.servlet.Servlet interface. This method is called before the JSP servlet object is destroyed. You can use this method to do some clean-up, if you want.

Most of the time, however, JSP authors rarely make full use of these two methods. The following example illustrates how you can implement these two methods in your JSP page:

## 2.14.2 The HttpJspPage Interface

This interface directly extends the JspPage interface. There is only one method: \_ jspService. This method is called by the JSP container to generate the content of the JSP page. The \_ jspService has the following signature:

public void \_jspService(HttpServletRequest request,

HttpServletResponse response) throws ServletException, IOException.

You can't include this method in a JSP page, such as in the following code:

## 2.14.3The JspFactory Class

The JspFactory class is an abstract class that provides methods for obtaining other objects needed for the JSP page processing. The class has the static method getDefaultFactory that returns a JspFactory object. From the JspFactory object, a PageContext and a JspEngineInfo object can be obtained that are useful for the JSP page processing. These objects are obtained using the JspFactory class's getEngineInfo method and the getPageContext method, whose signatures are given here:

public abstract JspEngineInfo getEngineInfo()

public abstract PageContext getPageContext (

Servlet requestingServlet, ServletRequest request,

ServletResponse response, String errorPageURL,

boolean needsSession, int buffer, boolean autoFlush)

The following code is part of the \_ jspService method that is generated by the JSP container:

JspFactory \_jspxFactory = null;

PageContext pageContext = null;

jspxFactory = JspFactory.getDefaultFactory();

pageContext = \_jspxFactory.getPageContext(this, request,

response, "", true, 8192, true);

## 2.14.4 The JspEngineInfo Class

The JspEngineInfo class is an abstract class that provides information on the JSP container. Only one method, getSpecificationVersion, returns the JSP container's version number. Because this is the only method currently available, this class does not have much use.

You can obtain a JspEngineInfo object using the getEngineInfo method of the JspFactory class.

## 2.14.5 The PageContext Class

PageContext represents a class that provides methods that are implementation-dependent. The PageContext class itself is abstract, so in the \_ jspService method of a JSP servlet class, a PageContext object is obtained by calling the getPageContext method of the JspFactory class.

The PageContext class provides methods that are used to create other objects. For example, its getOut method returns a JspWriter object that is used to send strings to the web browser. Other methods that return servlet-related objects include the following:

* getRequest, returns a ServletRequest object
* getResponse, returns a ServletResponse object
* getServletConfig, returns a ServletConfig object
* getServletContext, returns a ServletContext object
* getSession, returns an HttpSession object

## 2.14.6 The JspWriter Class

The JspWriter class is derived from the java.io.Writer class and represents a Writer that you can use to write to the client browser. Of its many methods, the most important are the print and println methods. Both provide enough overloads that ensure you can write any type of data. The difference between print and println is that println always adds the new line character to the printed data.

Additional methods allow you to manipulate the buffer. For instance, the clear method clears the buffer. It throws an exception if some of the buffer's content has already been flushed. Similar to clear is the clearBuffer method, which clears the buffer but never throws any exception if any of the buffer's contents have been flushed.

# 2.15 JSP Syntax:

"JSP Basics," you learned that a JSP page can have Java code and HTML tags. More formally, you can say that a JSP page has elements and template data. The elements, which also are called JSP tags, make up the syntax and semantics of JSP. Template data is everything else. Template data includes parts that the JSP container does not understand, such as HTML tags.

There are three types of elements:

* Directive elements
* Scripting elements
* Action elements

To write an effective JSP page, you need to understand all these elements well. Elements have two forms: the XML form and the <% … %> alternative form. Template data remains as it is, normally passed through the client uninterrupted.

This chapter discusses the three types of JSP elements and comments. It also presents examples on how to use these elements. You will also learn how incorporating these elements affects the JSP servlets—servlets that result from the translation of JSP pages.

## 2.15.1 Directives

Directives are messages to the JSP container containing information on how the JSP container must translate a JSP page into a corresponding servlet. Directives have the following syntax:

<%@ directive (attribute="value")\* %>

The asterisk (\*) means that what is enclosed in the brackets can be repeated zero or more times. The syntax can be re-written in a more informal way as follows:

<%@ directive attribute1="value1" attribute2="value2" ... %>

White spaces after the opening <%@ and before the closing %> are optional, but are recommended to enhance readability.

## 2.15.2 The Page Directive

The Page directive has the following syntax:

<%@ page (attribute="value")\* %>

Or, if you want to use the more informal syntax:

<%@ page attribute1="value1" attribute2="value2" ... %>

An example of the use of the Page directive is as follows:

<%@ page buffer="16384" session="false" %>

With JSP, you can specify multiple page directives in your JSP page, such as the following:

<%@ page buffer="16384" %>

<%@ page session="false" %>

##### The extends Attribute

The extends attribute defines the parent class that will be inherited by the generated servlet. You should use this attribute with extra care. In most cases, you should not use this attribute at all. In Tomcat, the parent class that will be subclassed by the resulting servlet is HttpJspBase.

##### The import Attribute

The import attribute is similar to the import keyword in a Java class or interface. The attribute is used to import a class or an interface or all members of a package. You will definitely use this attribute often. Whatever you specify in the import attribute of a page directive will be translated into an import statement in the generated servlet class. By default, Tomcat specifies the following import statements in every generated servlet class. You don't need to import what has been imported by default:

import javax.servlet.\*;

import javax.servlet.http.\*;

import javax.servlet.jsp.\*;

import javax.servlet.jsp.tagext.\*;

import org.apache.jasper.runtime.\*;

As an example, consider the following JSP page that imports the java.io package and the java.util.Enumeration interface:

<%@ page import="java.io.\*" %>

<%@ page import="java.util.Enumeration" %>

The two will be added before the default import statements in the generated servlet class, as described in the following code fragment:

import java.io.\*;

import java.util.Enumeration;

import javax.servlet.\*;

import javax.servlet.http.\*;

import javax.servlet.jsp.\*;

import javax.servlet.jsp.tagext.\*;

import org.apache.jasper.runtime.\*;

Chapter Three

Web Service And JAX-RPC

# 3.1 Introduction

The most significant aspect of Web services is that every software and hardware company in the world has positioned itself around these technologies for interoperability. No single technological advancement will have as great an impact on the way systems are developed as Web services.

Web services allow systems to communicate with each other using standard Internet technologies. Systems that have to communicate with other systems use communication protocols and the data formats that both systems understand. Developers use technologies such as CORBA, RMI, or DCOM most often. The principal problem with these communication technologies is that not every platform supports them. Developers must create gateways to convert an unsupported protocol and data format into one that the target platform understands.

The emergence of the Internet has forced vendors to support standards such as HTTP and XML. Over the past few years, vendors and their customers quickly realized that programs that communicate with each other could also use the technologies that run the Internet. Web services use Internet technology for system interoperability. The advantage that Web services have over previous interoperability attempts, such as CORBA, is that they build on the existing infrastructure of the Internet and are supported by virtually every technology vendor in existence. As a result of the ubiquitousness of the technologies they use, Web services are platform-independent. This means that whether the Web service is built using .NET or J2EE, the client uses the service in the exact same way.

# 3.2 Web Services Technology

The Web services stack shown in Figure 3.1, categorizes the technology of Web services into a layered model. The stack starts at the bottom with the basic technologies that allow data to transfer from one machine to another. Each layer builds on the lower layers and adds higher-level abstractions. The upper layers of the stack do not necessarily depend on the lower layers and in some ways are orthogonal concerns. They are shown in this format simply to demonstrate a higher level of abstraction.

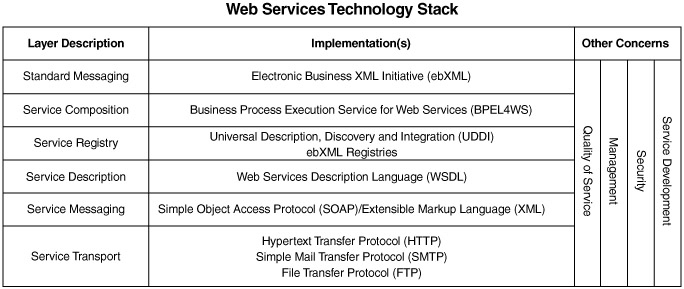


Fig 3.1 Web Service Technology Stack

## 3.2.1 Service Transport

The main function of the service transport layer is to transfer data from one machine to another. This is the protocol for the transmission of data for Web services. Web services use multiple transports to transfer data from service to service—including, but not limited to, HTTP, SMTP, and FTP.

The most popular protocol by far for use in Web services is HTTP. The Internet and the World Wide Web use HTTP to transmit data. HTTP is not blocked by most firewalls and thus is the standard for interoperable systems. HTTP 1.0 is a connectionless and stateless protocol. This means that each request and response a client sends and receives from a server is independent and not in the overall context of a conversation.

## 3.2.2 Service Messaging

The messaging layer of the technology stack describes the data formats used to transmit data from one service to another over the transport. XML is the base format used for Web services. XML is a text-based protocol whose data is represented as characters in a character set. The XML data is structured as a tree with elements, and the entire tree structure is called a document. XML has no data description separate from the data itself, unlike fixed or delimited data formats. The messages are self-describing. The data has specially formatted tags around it that give the data a name as well as a position in the document's tree structure.

Simple Object Access Protocol (SOAP) is a specification that tells a service consumer and a service provider how to format and read a specially formatted XML message for use in a service. A SOAP message has three sections: the envelope, the header, and the body.

The envelope is the top element of the XML message. It indicates that the message is a SOAP message, and it has instructions for processing the message.

The SOAP header contains application context information and directives. The header information is usually read and processed by the server, not the service. For instance, if an application needs entries for authentication or transaction management, the header will contain data for these features. SOAP messages can be passed along through multiple intermediaries between the service consumer and the service provider. Each intermediary reads the header information and uses it for routing, logging, and other system functions.

The SOAP body contains the application data. Web services support multiple message exchange patterns. A SOAP message may be sent in a document-oriented format or an RPC style format. In a document-oriented message exchange, the service consumer and provider exchange XML documents. In the RPC-style message exchange, data is passed as arguments. SOAP messages are by nature one-way transmissions from a sender to a receiver but are usually combined to implement a request/response model.

### 

## 3.2.3 Service Description

The service description specifies three aspects of the service:

* Operations the service has made available
* Messages the service will accept
* The protocol to which the consumer must bind to access the service

Web services uses the Web Services Description Language (WSDL) to specify a service contract. The service contract is a description of a set of endpoints that operate on messages and the specification for how an XML document should be formatted when it is sent to the endpoints. An endpoint is a network address that accepts a message formatted to the specification defined in the WSDL. WSDL uses the term *port* to describe a service endpoint for a message. WSDL describes the contract for a service as a collection of ports the service has made available.

## 3.2.4 Service Registry

Web services support the concept of dynamic discovery of services. A consumer of a service uses a service registry to find the services it is interested in using. Universal Description, Discovery and Integration (UDDI) is a Web service itself that supports a standard set of services that allow a Web service consumer to dynamically discover and locate the description for a Web service. UDDI registries are themselves Web services that expose an API as a set of well-defined SOAP messages. The Web service provider and Web service consumer use SOAP and HTTP to publish and retrieve information about services in the registry. Public UDDI registries contain a Web services contact, business owner, and technical information about the Web service. UDDI supports two types of conversations:

* The service provider uses the UDDI directory to publish information about the Web services it supports.
* The Web service consumer sends SOAP-formatted XML messages over HTTP to the UDDI directory, to retrieve a listing of Web services that match its criteria.
* Java and Web Services
* Web services are platform-independent. This means that a Web service may be developed in a large number of languages to run on many platforms. The Java language and Java 2 Enterprise Edition (J2EE) platform provide features for building and deploying Web services. The benefits of the Java language and the J2EE platform are vendor independence and application portability. Applications built on the J2EE platform may be deployed on J2EE implementations from a large number of vendors. Using Java to develop Web services gives you the benefit of vendor independence in addition to the inherent platform independence of Web services.
* Java has support for Web services through the Java Web Services Developer Pack (JWSDP). JWSDP contains libraries for generating XML and SOAP, processing XML, accessing service registries, and calling RPC-based Web services. JWSDP and the libraries it contains constitute the bulk of this book, so we won't go into detail here.

# 3.3 JAX-RPC Service Model

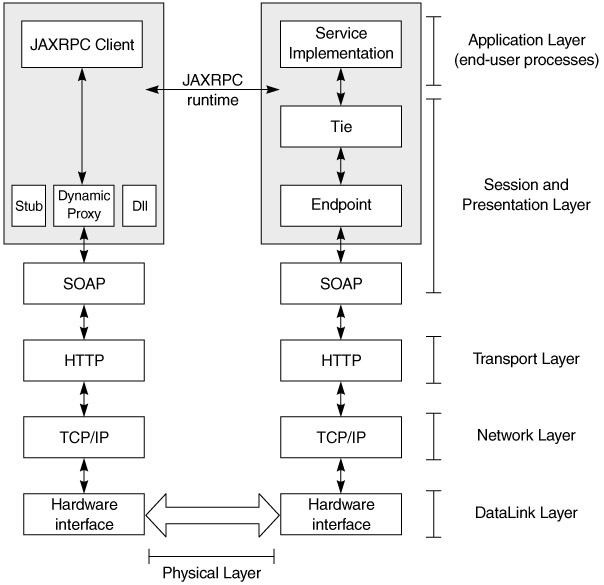
JAX-RPC was designed to provide a simple way for developers to create web services and web service clients using techniques that are not very different from those used in nondistributed Java programming. Programming with JAX-RPC is very similar to using RMI to create a distributed application, in the sense that client code appears to be making ordinary method calls on local objects. In reality, however, the infrastructure handles these calls by converting them to messages that are sent over a network to the server, where they cause a local call to be made on the actual method implementation. The results of this call are used to create a reply message that is sent back the client, where they are extracted and presented as return values from the client application's method call.

Although there are similarities between RMI and JAX-RPC, the major difference arises from the fact that the messages exchanged between JAX-RPC clients and services are encoded using an XML-based protocol and can potentially be carried by a range of transport-level protocols, including HTTP (or its more secure variant HTTPS), SMTP, or even FTP. JAX-RPC allows a client written in the Java programming language to access a service implemented on, for example, the Microsoft .NET platform, whereas RMI clients and servers must both be written in Java (although it is possible to expose an RMI/CORBA hybrid service written in any language that has a binding to CORBA IIOP). In other words, it can communicate with foreign services without needing to be aware of the technology that its peer is actually using.

One of the benefits of using JAX-RPC over a lower-level web services technology such as SAAJ or JAXM (both of which will be covered later in this book) is that it doesn't require you to know much about XML before you can start building a distributed application. This is because, with a few exceptions that fall into the advanced category, the programming interfaces are completely independent of both the underlying messaging infrastructure and the transport protocol that is used to carry the XML messages. The JAX-RPC specification requires every implementation to support at least the use of SOAP over HTTP 1.1, but, as a developer, you can use JAX-RPC without having to be an expert in XML, SOAP, or HTTP. On the other hand, it is possible to use some of the more advanced JAX-RPC features to gain access to the lower levels. Here, you can directly handle SOAP headers or extend the set of data types that the client and server can exchange beyond those supported transparently by JAX-RPC.

As Figure 3.2 shows, the service model for JAX-RPC is similar to other RPC models, such as RMI-IIOP and CORBA. The model has several components

## Fig3.2 JAX-RPC Service Model



The layers shown in Figure 3.2 correspond to the Open System Interconnection (OSI) networking model, which has these characteristics:

* The physical layer conveys the bitstream through the network.
* The data link layer encodes and decodes data packets into bits.
* The network layer provides switching, routing, packet sequencing, addressing, and forwarding between virtual circuits, to transmit data from node to node.
* The transport layer provides transparent transfer of data between hosts and is responsible for end-to-end error recovery and flow control. Clearly, the HTTP binding for SOAP lacks some of this, whereas other bindings, such as POP-SMTP, IMAP, and JMS do not.
* The session layer establishes, coordinates, and terminates connections, exchanges, and dialogs between the applications.
* The presentation layer, also known as the syntax layer, provides independence from differences in data representation by translating from application to network format, and vice versa. The presentation layer works to transform data into the form the application layer can accept.
* The application layer is the actual application and end-user processes, where business functionality is addressed.

Although JAX-RPC relies on complex protocols, the API hides this complexity from the application developer. On the server side, the developer specifies the remotely accessible procedures by defining methods in a Java *service definition* interface and writing one or more Java classes that implement those methods. JAX-RPC exposes these objects as a *service endpoint* and generates the relevant ties. The client never directly communicates with the *service implementation.* The client uses a stub or other mechanisms to communicate with the endpoint (covered later in this chapter), and the endpoint uses the tie. The client then invokes the service, passing in relevant parameters, and the service returns the results to the client.

Before we dive into the internals of this model, we will take a look at the data types and see how the marshalling and unmarshalling occurs. We will then see how to use that in developing JAX-RPC services.

# 3.4 Data Types and Serialization

Let us revisit some object-oriented concepts. An object at any time has state. This state, represented by its member variables at that time, is the object's snapshot. The definition of the object is the class file or compiled representation. An object with no member variables—that is, no state—is essentially just a utility that does something useful every time its methods are invoked. It may create other objects and change their states, but the *scope* of such secondary objects is limited to the method.

To do a remote procedure call, something representing *state* must be sent over the wire, and something representing *state* must be returned. Sending objects over the network is not trivial, since the network is not aware of objects; it supports only bit transmission.

The mechanism used to change the objects into a format that can be transmitted over the network is called *marshalling,* and reconstructing the objects from this format is called *unmarshalling.* Marshalling over the wire requires object state to be extracted and sent in a well-defined format. Unmarshalling requires that the format be known, for reconstruction to take place. To marshal and unmarshal successfully, both sides in the exchange must use the same protocol to *encode* and *decode* object structure and data. For example, RMI Java uses Java serialization to marshal and unmarshal objects over Java Remote Method Protocol (JRMP). CORBA uses IIOP, DCOM uses ORPC, and Gemstone uses SRP.

In summary, four things are required between communication parties in different address spaces:

1. An agreement on the data format
2. An agreement on the mechanism for transforming and reconstructing object state into this format
3. An agreement on the protocol for communication between objects
4. An agreement on the transport protocol

XML helps in achieving item 1, XML schemas and SOAP with 2 and 3, and HTTP (and others in the IP family of protocols) with 4

So how is this relevant to JAX-RPC? JAX-RPC defines

* The data type mapping of Java-XML and XML-Java for making the remote service invocation possible
* Java-WSDL and WSDL-Java for making the service description possible

This is significant, because JAX-RPC provides a *standard* for vendors to implement and makes developer code vendor-neutral, much the way any of the other Java specifications do. Just as developers write a J2EE application and expect it to behave the same across J2EE-compliant application servers from multiple vendors, JAX-RPC applications will behave the same across JAX-RPC runtimes.

This does not mean that a JAX-RPC client can call only a JAX-RPC service and a JAX-RPC service can be used only by a JAX-RPC client. An application could still use a JAX-RPC client to invoke a .NET service and a .NET client to invoke a JAX-RPC service, as we will demonstrate later. As Figure 3.3 shows, because the data format, object communication protocol, and transport protocol are platform- and vendor-implementation independent, the application can be accessed by any client on any platform, as long as it uses these standards. The data type mapping and serialization rules defined by JAX-RPC are useful when the JAX-RPC runtime is being used on the Java platform at the client or server end.

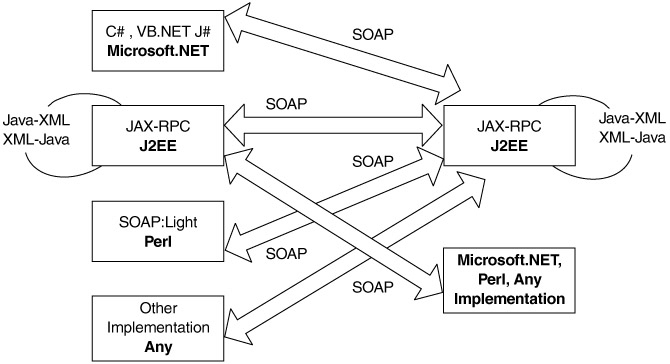


Fig 3.3 JAX-RPC client-server interaction

RPC implementations using SOAP from different vendors:

* Apache SOAP 2.2
* Apache Axis (Alpha-1)
* HP Web Services Platform
* IBM Web Services Toolkit, WSIF
* IONAXML Bus 1.2
* Microsoft SOAP Toolkit 2.0
* Microsoft.NET
* Others: PocketSOAP, SOAP::Lite, Systinet WASP, SOAP-RMI, GLUE, Cape Clear

From an RPC perspective, if the client and service are written in Java, the runtime needs to know the following information:

1. The endpoint of the service—that is, where to invoke the service
2. The method name and signature—that is, what to invoke
3. How to marshal Java data types and objects into an XML format that can be transmitted over the wire to invoke the method
4. How to unmarshal the incoming XML into Java data types and objects to process the results of that operation, if any

## 3.4.1 Java-to-XML Marshalling

While JAX-RPC does not define the actual marshalling mechanism, it does define the input and output types that result from that marshalling. Vendors write the marshalling code as part of their implementations.

In JAX-RPC, marshalling is different from the standard Java serialization mechanism, where all nontransient fields in the class are automatically serialized. JAX-RPC defines a standard set of Java types as method arguments and return types, meaning that a JAX-RPC—compliant system will provide the ready-to-use serializers and deserializers for these types:

1. All Java primitives, with the exception of a char (int, float, long, short, double, byte, Boolean). A char is treated as a String, since XML schemas have no char type primitive .
2. An object that is an instance of
   * java.lang.String
   * java.util.Date
   * java.util.Calendar
   * java.math.BigInteger
   * java.math.BigDecimal
3. An object that is an instance of a class that conforms to the following restrictions:
   * It should conform to the JavaBean specification, so that its variables can be easily accessed.
   * It should not be a Remote object (i.e., should not implement java.rmi.Remote).
   * It should have a default no arguments constructor.

These are important, because any other class is passed between the client and the server. For example, if a java.util.Map of com.flutebank.Account objects must be passed from the client to the server, a pluggable serializer and deserializer pair must be written. This is explored later in this chapter.

1. An array (with the caveat that it must contain bytes or a supported type)
2. A java.lang.Exception class.
3. The above rules differ from the standard Java serialization requirements. Fig3.4 lists the details for the same class to be used across an RMI/RMI-IIOP application as well as a JAX-RPC application. This is relevant where an existing EJB, RMI, or RMI-IIOP object must expose itself directly as a JAX-RPC service and the code must be reused across these interfaces.

| Fig 3.4: Portability across JAX-RPC and RMI | | |
| --- | --- | --- |
| **JAX-RPC** | **Java serialization** | **A portable value object** |
| Should not extend Remote. | Can extend Remote. It is treated as an remote object and passed by reference. | Should not implement Remote. |
| Serializable not required. | Serializable required. | Should implement Serializable. |
| transient fields are not serialized. | transient fields are not serialized. | transient fields are not serialized. |
| static fields are serialized. | static fields are not serialized. | Should not contain static fields. |
| All public variables are serialized. | public transient variables are not serialized. | Should not contain any public transient variables. |
| Only private, protected, package-level fields that have get/set methods are serialized. | Get/set methods are not required. Private, protected, package-level fields are still serialized. | Should have get/set methods for all private, protected, package-level fields. |
| Bean properties are serialized. | Bean properties are serialized. | Can have bean properties with get/set methods. |

Once the parameter types have been defined, rules and a standard mechanism to map these data types from Java to XML must also be defined. JAX-RPC does this, as shows. Fig 3.5

| **Java type** | **XML type** |
| --- | --- |
| Boolean | xsd:oolean |
| Byte | xsd:byte |
| Short | xsd:short |
| Int | xsd:int |
| Long | xsd:long |
| Float | xsd:float |
| Double | xsd:double |
| byte[] | xsd:base64Binary |
| Byte[] | xsd:base64Binary |
| java.lang.String | xsd:string |
| java.math.BigInteger | xsd:integer |
| java.math.BigDecimal | xsd:decimal |
| java.util.Calendar | xsd:dateTime |
| java.util.Date | xsd:dateTime |
| javax.xml.namespace.Qname | xsd:Qname |
| JavaBean class whose properties are any supported  Java data type or another valid JavaBean | XML schema sequence of elements |
| Array of any of above | SOAP array |

Fig 3.5 Java-to-XML Data Type Mapping

# 3.5 JAX-RPC Development

We have just covered how data can be transferred over the wire, along with the rules and associated mechanics governing that. In this section, we will look at how services can be developed and realized using JAX-RPC and the steps involved in doing so.

Developing and consuming a JAX-RPC service can be categorized into five steps:

1. Service definition
2. Service implementation
3. Service deployment
4. Service description
5. Service consumption

## 3.5.1 Service Definition

The term *service definition* is used to refer to the abstraction that defines the publicly surfaced view of the service. The service definition is represented as a Java interface that exposes the service's operations. The service definition is also called a remote interface, because it must extend the java.rmi.Remote interface, and because all methods in it must throw a java.rmi.RemoteException. The code below shows the BillPay Web service:

package com.flutebank.billpayservice;

import java.util.Date;

import java.rmi.Remote;

import java.rmi.RemoteException;

public interface BillPay extends Remote {

public PaymentConfirmation schedulePayment(Date date, String nickName, double

amount) throws ScheduleFailedException, RemoteException;

public PaymentDetail[] listScheduledPayments() throws RemoteException;

public double getLastPayment(String nickname) throws RemoteException;

}

The methods in the interface must have valid JAX-RPC data types (disussed earlier) as arguments and return types. If they are not a supported data type (e.g. java.util.Map), then appropriate *serializers* and *deserializers* must be available, so that these types can be marshaled and unmarshalled to and from their corresponding XML representations. The data type can also be a *holder* class. Holders and pluggable serializers are covered later in this chapter.

An implemenatation will usually verify this type information at compile time and warn the developer if it is not correct. A request sent with incorrect type information at runtime will generate a SOAP fault, because it will not be able to unmarshall the XML.

## 3.5.2 Service Implementation

The *service implementation,* also known as a *servant,* is the concrete representation of the abstract service definition; it is a class that provides the implementation or the service definition. The Java class must have a default constructor and must implement the remote interface that defines the service.

Services are deployed in a JAX-RPC *runtime,* which is a container that implements the JAX-RPC specifications. By default, the runtime will just invoke the methods corresponding to the RPC request in the Java implemenatation. The service implementation can choose to provide hooks to allow the runtime to manage the service's lifecycle and allow the container to invoke callbacks on the service when major lifecycle events occur. The "hook" is defined as a javax.xml.rpc.server.ServiceLifeCycle interface that the service can implement. The container will then invoke methods on this service appropriately, via this interface. The interface defines an init(Object context) and a destroy() method:

public interface ServiceLifecycle{

public void init(Object obj) throws ServiceException;

public void destroy();

}

The behavior of these methods is similar to the init() and destroy() methods in a servlet. When the implementation is first instantiated, the init() method is invoked, and a context object passed to it, the destroy() method is called before the implementation needs to be removed (e.g., at shutdown or during a resource crunch). These methods are good places to initialize and release expensive resources, such as database connections and remote references. The context is defined as an Object, to allow for different endpoint types to be used, as we will see later (e.g., the context will be different for an HTTP endpoint and a JMS endpoint).

As with a servlet, an implementation should not hold a client-specific state in instance variables, because the runtime can invoke methods from multiple threads. Architects should also avoid synchronizing the methods themselves. There are other ways to maintain client state.

## 3.5.3 Service Deployment

We mentioned earlier that a service is deployed in a JAX-RPC runtime. A *service* *endpoint* is the perimeter where the SOAP message is received and the response dispatched. It is the physical entity exposed to service consumers that essentially services client requests. An endpoint is provided by the runtime and is not written by developers. An endpoint is bound to the transport protocol. Because a runtime is required to support an HTTP transport, JAX-RPC also defines the behavior of an endpoint for this protocol as a Java servlet, as Figure 3.6 shows.

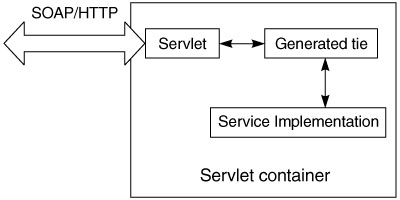


Fig 3.6 Service deployment

The servlet receives the SOAP message as the HTTP request, determines the servant to use for servicing that request, and delegates to it or its proxy representation (the tie). Once the service has done its work, the servlet is responsible for packaging the SOAP message and sending it back over HTTP.

The exact implementation of the servlet endpoint is left up to the runtime. The reference implementation contains a single servlet (com.sun.xml.rpc.server .http.JAXRPCServlet) that delegates to a tie, based on the xrpcc-generated properties file (we will see this later in the chapter). Because the endpoint is a servlet, it requires a Servlet 2.2—compliant container. Also, the packaging and deployment to the endpoint of the service has to be the standard J2EE WAR file, with its defined structure (WEB-INF/classes and the web.xml file, etc.)

If a service implementation implements the ServiceLifeCycle interface, the context object passed in the init() is of type javax.xml.rpc.server.ServletEndpointContext:

public interface ServletEndpointContext{

public MessageContext getMessageContext();

public Principal getUserPrincipal();

public HttpSession getHttpSession();

public ServletContext getServletContext();

}

This context provides methods to access the MessageContext, Principal, HttpSession and ServletContext objects associated with the user. The listing below shows an example of how this can be used. These objects are good places for maintaining different kinds of state information:

* The HttpSession is a good place to maintain *client*-specific state, using the getAttribute() and setAttribute() methods.
* The ServletContext is a good place to access *application*-specific state, such as configuration parameters, Java Naming and Directory Interface (JNDI) names, and JNDI contexts, using the getAttribute() and setAttribute() methods.
* The MessageContext is a good place to obtain state set by message handlers during preprocessing of the message. Handlers are covered in detail later in the chapter.

public class BillPayImpl implements BillPay, ServiceLifecycle {

private ServletEndpointContext ctx;

public void init(java.lang.Object context){

ctx=(ServletEndpointContext)context;

}

public PaymentDetail[] listScheduledPayments() {

SOAPMessageContext msgctx= (SOAPMessageContext) (ctx.getMessageContext());

HttpSession session = ctx.getHttpSession();

ServletContext servletctx= ctx.getServletContext()

// other code

}

}

The usage of the ServletEndpointContext is analogous to the SessionContext and EntityContext in EJBs.

## 3.5.4 Service Description

Once the service is defined, implemented, and ready for deployment as an endpoint, it also must be *described* clearly for service consumers. This is where WDSL comes in. Based on the service definition, the WSDL document describes the service, its operations, arguments, return types, and the schema for the data types used in them.

#### xrpcc Internals

The JAX-RPC reference implementation comes with the xrpcc (XML-based RPC Compiler) tool, which reads a tool-specific XML configuration file and generates the client- or server-side bindings shown in Figure 3.7 . A developer can start with

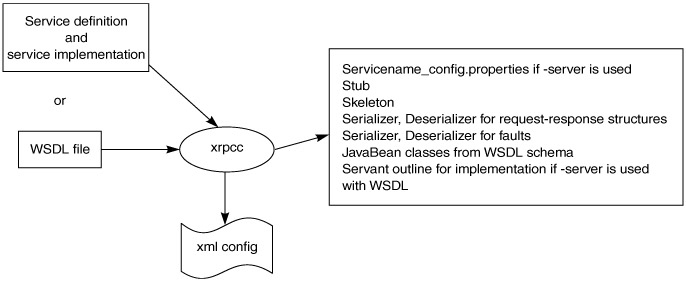


Fig 3.7 xrpcc artifacts

* A remote interface and use xrpcc to generate the stubs, ties, and WSDL
* A WSDL document and generate the stubs to consume the service
* A WSDL document and generate the stubs, ties, and remote interface and implement the service

**Service Element.** This describes the overall service. Only one service can be defined in the XML descriptor, to prevent potential name clashes in the generated code for the different services and the types they use.

* name. The name of the service. This is also used as the value for the service element in the generated WSDL.
* package. The package name for the generated service classes. xrpcc generates the stubs with the same package name as the service interface.
* targetnamespace. The target namespace for the generated WSDL document.
* typenamespace. The namespace for the schema portion of the generated WSDL document.

**Interface element.** This defines details about the interface the service supports. A service can have multiple interfaces.

* name. Fully qualified name of an interface, such as com.flutebank.billpay.Billpay.
* servant. Fully qualified name of the service interface implementation.
* soapAction. Value to be used as the SOAPAction for all operations in the corresponding port (optional).
* soapaActionBase. Value used as a prefix for the SOAPAction strings for the operations in the corresponding port (optional).

**Handlerchain element.** Defines information about handlers for this service. The handler element can be defined inside a service. If so, it is available to all interfaces inside the interface element, in which case it is specific only to that interface.

* runAt. Defines where the handler is to be executed. Possible values are client or server.
* roles. Lists or defines the roles that the handler will run as. This is the whitespace-separated List (xsd:anyURI)value returned by HandlerChain .getRoles().
* className. Fully qualified name of the handler class.
* headers. The header blocks processed by the handler. This is the whitespace-separated List(xsd:QName)-qualified name of a header block's outermost element.
* property. Multiple and arbitrary name-value pairs the handler can use internally, such as configuration and initialization parameters. These properties are passed as input to Handler.init(HandlerInfo config) through the "config" argument. The HandlerInfo.getHandlerConfig() method returns a Map containing all property name-value pairs specified in the <property/> elements.

#### Java-WSDL Mappings

we will discussed the WSDL structure, the role of vendor tools, and the significance of a standard specification to map WSDL elements to Java (and vice versa). To understand this mapping, let us revisit the role of WSDL elements from that chapter (Figure 3.8).

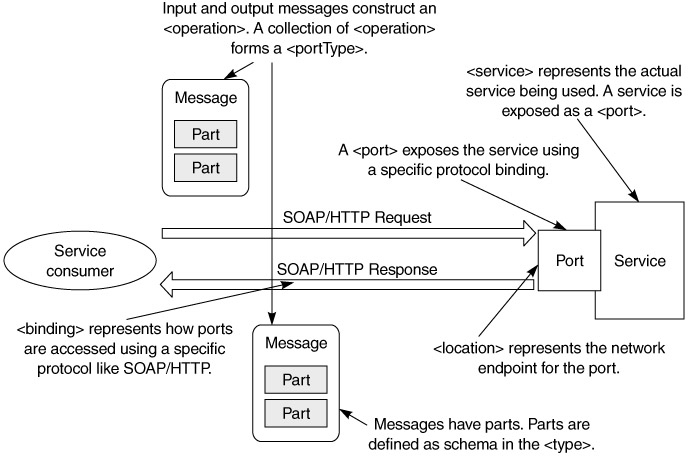


Fig 3.8 WSDL elements and dynamic interaction of a service and its consumer.

A Web service exposes groups of business operations for service consumers to use. Operations are grouped together to form portTypes. To invoke an operation, the consumer sends an input message containing the input data. It gets an output message containing the data that results from the business processing, or a fault if a problem occurs. The input and output messages may have multiple data items in them; each is called a part.

The wire protocol used for the invocation and the format of the input and output messages on the wire for that protocol are specified in a binding element. The service exposes itself to consumers through one or more ports, each of which specifies a network address where the service is located and the binding to use with that port. A service may render itself though several ports, where each port has a different binding (e.g., the same service may expose itself via SOAP/HTTP and SOAP/SMTP).

## 3.5.5 Service Consumption

Until now, we have seen how to define, implement, and deploy a JAX-RPC service. Let us now look at how such a service can be consumed. A *service consumer* represents the abstraction of the entity invoking the facilities of an existing service. Invocation modes for doing so fall into three broad categories:

* **Synchronous request-response.** The client invokes a remote procedure and blocks until a response or an exception is received from the service. The client cannot do any other work while awaiting the response. This is analogous to making a phone call. Either someone responds by picking up the handset on the other end, or a busy tone is received.
* **One-way RPC.** The client invokes a remote procedure but does not block or wait to receive a return and is free to do other work. In fact the client does not receive any return parameters. This is analogous to sending a fax (fire and forget!). When a fax is sent, a person does not need to pick up the phone on the receiving end for the fax to go through.
* **Nonblocking RPC invocation.** The client invokes a remote procedure and continues processing without waiting for a return. The client may process the return later by polling some service or by using some other notification mechanism. This is analogous to making a phone call and getting an answering machine. The caller leaves a message and continues. The person on the other end gets the message and returns the call by dialing the number left on the machine or a number he or she already knows.

The significant difference between one-way and nonblocking invocation is that in the former, the client will not receive a return value.

As a bare minimum, JAX-RPC implementations must support the first two modes for client invocation and HTTP 1.1 as the transport binding for SOAP. The sematics of nonblocking RPC are quite complicated. For example, the client must inform the service of an endpoint to which the service can repond, and both parties must deal with issues of reliability and availability. If your application requires asynchronous communication, messaging is probably more appropriate.

Let us now look at the mechanisms an RPC client can use to consume the service in these invocation modes. The client can be written to invoke the service using one of the following three mechanisms:

* Stub
* Dynamic invocation interface
* Dynamic proxies

|  |
| --- |
|  |

Chapter Four

Methology And System analysis and design

# 4.1 INTRODUCTION

Driven by the explosive growth of information available on the Internet, intelligent information access has become a central research area in computer science. The 20th century is commonly characterized as “The Information Age”, and the sheer amount of information readily available today has created novel challenges. Numerous intelligent information agents software tools that provide personalized assistance for users navigating large information spaces -- have been described in the literature and deployed on the World-Wide-Web However, the need for intelligent information agents is not limited to web-based applications, as we are currently witnessing an increasing trend towards “ubiquitous information access”. Different types of wireless information devices, designed to tap into the Internet’s vast information resources without physical constraints, are currently being released into the marketplace. For example, cell phones can access Internet-based information services, and pagers

can alert users of late-breaking news. While these devices undoubtedly enhance the utility of online information and are likely to open up opportunities for revolutionary information centric applications, they are cramped by several technical constraints. First, the small size of wireless information devices leads to inherently limited user interfaces. Second, bandwidth constraints impose limits on the amount of information to be transferred. Third .The World-Wide-Web is currently witnessing an ongoing trend towards personalized information access. As part of this trend, numerous personalized news services are emerging. For example, Internet portals such as Yahoo, Lycos and Excite offer personalized access to daily news stories from a large range of categories. These services are based on static questionnaires that users fill out in order to make use of news filtering capabilities. We believe that this level of personalization is not fine-grained enough for price- and bandwidth-sensitive information access. Here, we suggest the use of an intelligent news agent that unobtrusively learns about a user’s interests in daily news stories by observing the user’s browsing behavior. Using a content-based machine learning algorithm originally developed for a web-based client ,the agent learns to rank-order news stories with respect to the user’s individual interests.

In this Project, we focus on a System designed to help users access interesting news stories. System organizer as an example of a wireless information device, noting that the learning approach and interface design reported here generalize to other devices such as cell phones or two-way pagers.

# 4.2 How to Build a Successful News System:

News System are fast-paced System which track, aggregate and disseminate current news on a set range of topics. Their main selling points are **comprehensiveness** and **timeliness**: a good news System must cover the field throughly and speedily by pointing readers to new information or developments. Content for news System is published regularly on a daily basis and sometimes multiple times throughout the day. Apart from the value of the actual opinions offered by news bloggers: people subscribe to or follow these news blog largely because they help them to stay on top of current issues, innovations or ideas concerning a industry. News System are also profitable if they are monetized because they publish a very high volume of content, which continuously attracts both search and referral traffic. If you’re an entrepreneur or webmaster looking to generate some online income, a news system is a good foundation to start with because you’ll never run out of content: the **consistent news flow drives you to publish regularly**. If you’re interested in a specific field, publishing a news system forces you to keep up with what other people are talking about. It’s different from just reading other blogs because you’ll participate by contributing your own ideas or opinions. This is a good way for you to build a [strong online reputation](http://www.doshdosh.com/how-to-use-the-web-to-build-a-powerful-reputation-in-any-industry/).

### Determining Your Information Sources: An Essential First Step

One of the best ways to find interesting news is to subscribe to other systems in your niche. This includes both other news system and non-news system and that you will pay extra attention to blogs which specialize in a sub-niche because they are usually the ones who publish unique content worth mentioning.

**There are several ways to find other blogs in your niche:**

**Search Engines** :This increases the relevance of the search results.

**Blog rolls** : Some of the blogs in your niche will have blog rolls or a links page with links to other relevant blogs or news sources. This is an excellent to surf around and find other relevant blogs which cover the same topic.

**Personal Recommendations:** Sometimes the easiest way is to ask your friends or members of a forum for recommendations on sites you should check out. The benefit of personal recommendations is that you’ll almost always come across good weblogs. It’s a good way to weed out most of the junk.

**Online Newspapers and Magazines**: Online newspapers and magazines will provide you with a lot of news on your chosen topic and its a good complement to blogs and social news websites, which may sometimes overlook a particular piece of news.

Online newspapers also has the advantage of offering localized news on either a national or city-basis. This is useful if you run a news blog that is divided in country or region-based coverage. Start first by subscribing to the major news outlets like the BBC, CNN and New York Times.

**Academic Journals and Trade Publications:** Academic journals and trade publications often publish reports which you can highlight on your blog in the form of news. This usually includes polls, statistical analysis of a specific phenomenon or reports on the performance of an industry. Not all of these publications are free and some do require a paid subscription.

It’s possible to obtain complimentary copies of these academic journals regularly, if you’re a blogger with a certain degree of clout. In some scenarios, publishing houses will be willing to send you new books or journals in return for a mention on your website. It is also possible to obtain these journals by being a contributing member of a scholarly society or educational institute.

**Press Releases and Media Contacts:** When we are starting out as a relatively unknown system, we need to take the initiative to **accumulate media contacts**. Email companies and web services which are relevant to our system topical focus and inform them that you are open to receiving email press releases from their marketing/PR department.

We’ll usually be put on an email list and you can take this chance to initiate better relationships with the specific PR manager/executive. Ideally, you want to be invited to launch parties and networking events as a member of the press.

# 4.3 Design Overview:

The News Service System was designed for implementation using the JSP platform, running on Apache Tom Cat The implementation tool used to develop the system is NetBeans v6.7.1, and the language that was used is Java. However, for the purposes of testing and evaluation, the system will use a MySql database, And we using JAX-RPC for web service with a schema developed specifically for this project.

The choice of this technology was made primarily due to our desire to learn this powerful programming language and technology. Further, the initial assessment of the development environment was that it was not only suitable for large-scale web development, but that it provides excellent support for such development. It is developed and supported by one of the largest players in the software industry, and thus is not likely to disappear from the market place in the foreseeable future.

# 4.4 Object orieneted model:

We will describe our system by using object oriented model Data Flow Diagram and use case diagram.

## 4.4.1 Data Flow Diagram:

The overall system DFDs in figure 4.1 and 4.2 shows several main interfaces and processes that will govern the flow of data between the different system components, which is clearly shown down. The administrator here could represent the person responsible of running the expert system that is represented by the "Administration Order" data flow, where he enters certain predefined parameters and tables provided by the system In the implementation of the system the expert system was replaced with a primitive sub-system called Organizer .

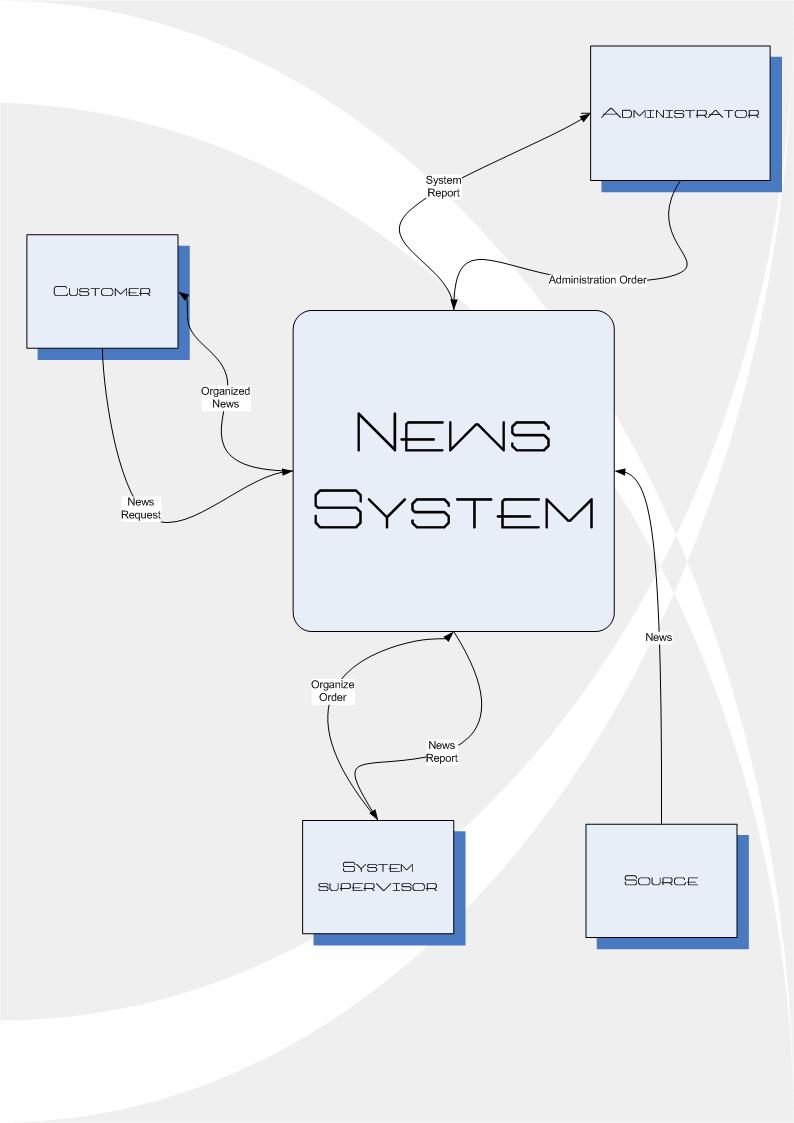


Fig 4.1 Context Diagram Of News System



Fig 4.2 Diagram 0 of News System

## 44.2 Use Case Diagram:

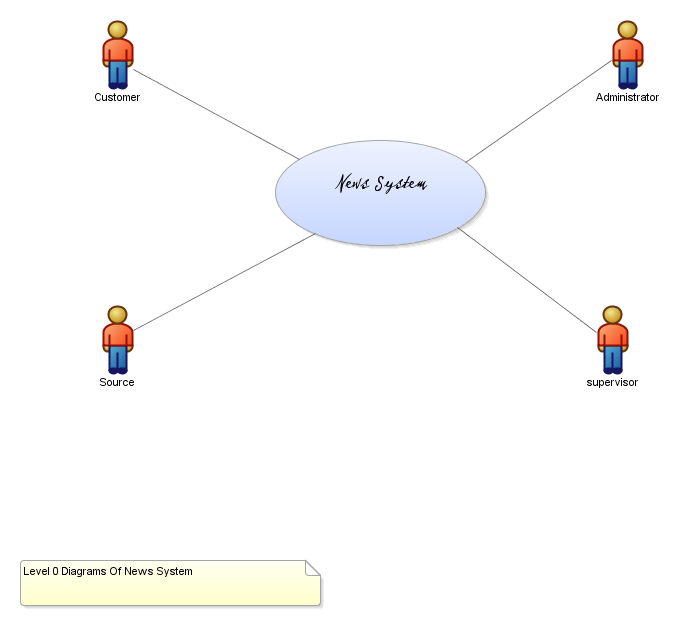


Fig 4.3 Use Case Level 0

**Brief Description**

This use case is news system its provide information and data to supervisor and to customer and get this data from source and it get administration order from administrator

**Primary Actor**

Customer

**Other Actors**

Source

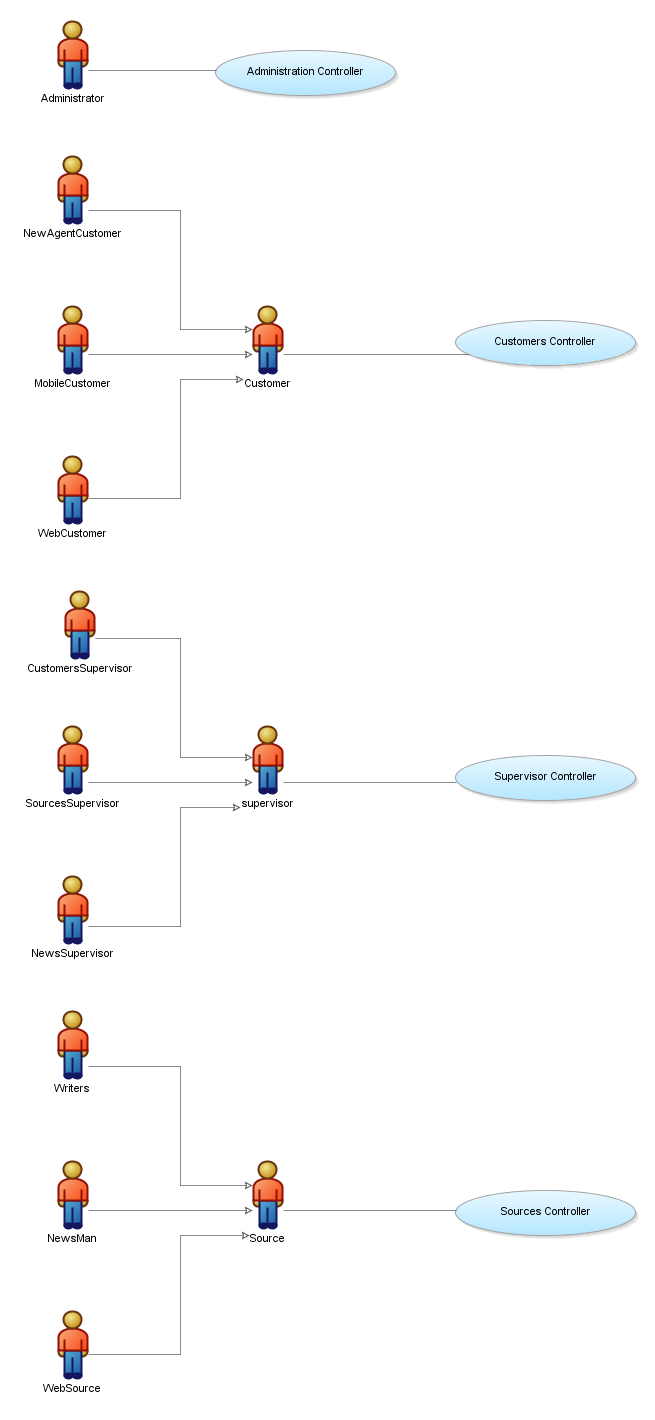
Supervisor

Administrator

**Uses**

None

**Extends:** None

Fig 3.3 Use Case Level 1

**Brief Description**

We have three use case customer controller, source controller And supervisor controller this use cases explain how the customer can get new from system and how the source provide news to the system and how the supervisor can supervising in this system.

**Primary Actor**

Customer

**Other Actors**

Source

Supervisor

Administrator

Web source

News man

Writers

Source supervisor

Customer supervisor

New super visor

Mobile customer

News agent customer

Web customer

**Uses**

None

**Extends:** None

# References

Daniel Billsus, M. J. (2008). A Learning Agent for Wireless News Access. *University of California,* .

James McGovern, S. T. (2005). *Java Web Services Architecture.* Morgan Kaufmann Publishers.

Perry, B. W. (January 2004). *Java Servlet & JSP Cookbook.* O'Reilly.

Peter den Haan, L. L. (2004). *Beginning JSP 2: From Novice to Professional.* Apress .

Topley, K. (2003). *Java Web Services in Nutshell.* O'Reilly.