Developing Java Web Services

Developing Java Web Services

APTECH LIMITED
Contact E-mail: ov-support@onlinevarsity.com

Edition 1 - 2014

Dear Learner,

We congratulate you on your decision to pursue an Aptech course.

Aptech Ltd. designs its courses using a sound instructional design model – from conceptualization to execution, incorporating the following key aspects:

- Scanning the user system and needs assessment

Needs assessment is carried out to find the educational and training needs of the learner

Technology trends are regularly scanned and tracked by core teams at Aptech Ltd. TAG* analyzes these on a monthly basis to understand the emerging technology training needs for the Industry.

An annual Industry Recruitment Profile Survey# is conducted during August - October to understand the technologies that Industries would be adapting in the next 2 to 3 years. An analysis of these trends & recruitment needs is then carried out to understand the skill requirements for different roles & career opportunities.

The skill requirements are then mapped with the learner profile (user system) to derive the Learning objectives for the different roles.

- Needs analysis and design of curriculum

The Learning objectives are then analyzed and translated into learning tasks. Each learning task or activity is analyzed in terms of knowledge, skills and attitudes that are required to perform that task. Teachers and domain experts do this jointly. These are then grouped in clusters to form the subjects to be covered by the curriculum.

In addition, the society, the teachers, and the industry expect certain knowledge and skills that are related to abilities such as *learning-to-learn, thinking, adaptability, problem solving, positive attitude etc*. These competencies would cover both cognitive and affective domains.

A precedence diagram for the subjects is drawn where the prerequisites for each subject are graphically illustrated. The number of levels in this diagram is determined by the duration of the course in terms of number of semesters etc. Using the precedence diagram and the time duration for each subject, the curriculum is organized.

- Design & development of instructional materials

The content outlines are developed by including additional topics that are required for the completion of the domain and for the logical development of the competencies identified. Evaluation strategy and scheme is developed for the subject. The topics are arranged/organized in a meaningful sequence.

The detailed instructional material – Training aids, Learner material, reference material, project guidelines, etc.- are then developed. Rigorous quality checks are conducted at every stage.

- Strategies for delivery of instruction

Careful consideration is given for the integral development of abilities like thinking, problem solving, learning-to-learn etc. by selecting appropriate instructional strategies (training methodology), instructional activities and instructional materials.

The area of IT is fast changing and nebulous. Hence considerable flexibility is provided in the instructional process by specially including creative activities with group interaction between the students and the trainer. The positive aspects of web based learning –acquiring information, organizing information and acting on the basis of insufficient information are some of the aspects, which are incorporated, in the instructional process.

- Assessment of learning

The learning is assessed through different modes – tests, assignments & projects. The assessment system is designed to evaluate the level of knowledge & skills as defined by the learning objectives.

- Evaluation of instructional process and instructional materials

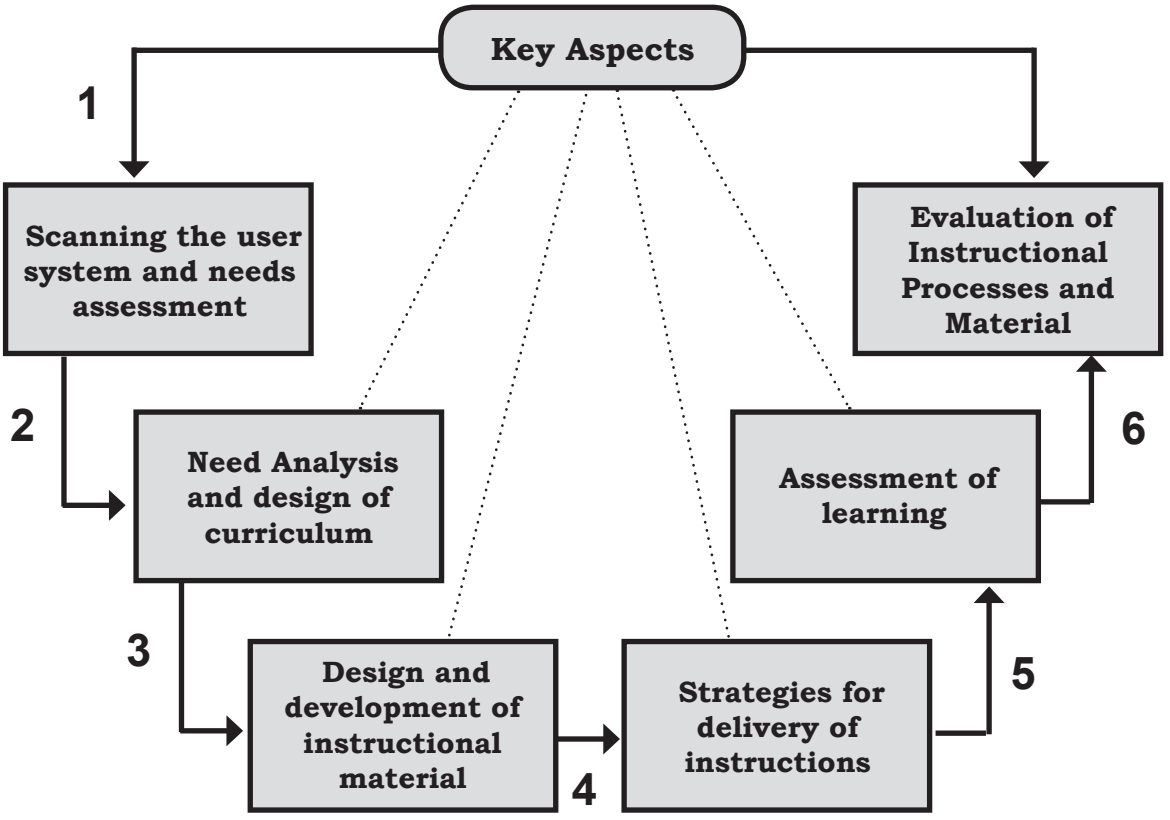
The instructional process is backed by an elaborate monitoring system to evaluate - on-time delivery, understanding of a subject module, ability of the instructor to impart learning. As an integral part of this process, we request you to kindly send us your feedback in the reply pre-paid form appended at the end of each module.

*TAG – Technology & Academics Group comprises of members from Aptech Ltd., professors from reputed Academic Institutions, Senior Managers from Industry, Technical gurus from Software Majors & representatives from regulatory organizations/forums.

Technology heads of Aptech Ltd. meet on a monthly basis to share and evaluate the technology trends. The group interfaces with the representatives of the TAG thrice a year to review and validate the technology and academic directions and endeavors of Aptech Ltd.

Industry Recruitment Profile Survey - The Industry Recruitment Profile Survey was conducted across 1581 companies in August/September 2000, representing the Software, Manufacturing, Process Industry, Insurance, Finance & Service Sectors.

Aptech New Products Design Model



Onlinevarsity
your e-way to learning

WRITE-UPS BY

EXPERTS AND **LEARNERS**

TO PROMOTE NEW AVENUES AND
ENHANCE THE LEARNING EXPERIENCE

FOR FURTHER READING, LOGIN TO

www.onlinevarsity.com

Preface

In a network, various types of data exchange occur between software systems. Different software use different programming languages. Hence, there arises a need for a method of communication between two different electronic devices on the network. A Web service is such a software system that is designed to support interoperable machine-to-machine interaction over the network.

The book 'Developing Java Web Services' provides a foundation to develop Web services using Java EE 7. The book begins with an introduction to Web services. It explains about Service Oriented Architecture (SOA) and Java APIs for XML Web Services (JAX-WS). It also explains Java APIs for XML Processing (JAXP), Java APIs for XML Registry (JAXR), SOAP with Attachment API for Java (SAAJ), and Java Architecture for XML Binding (JAXB). Further, the book explains about Simple Object Access Protocol (SOAP), Web Service Description Language (WSDL), and Universal Description, Discovery, and Integration (UDDI). Finally, the book introduces the concept of JAX-WS and RESTful Web services and different Web service clients.

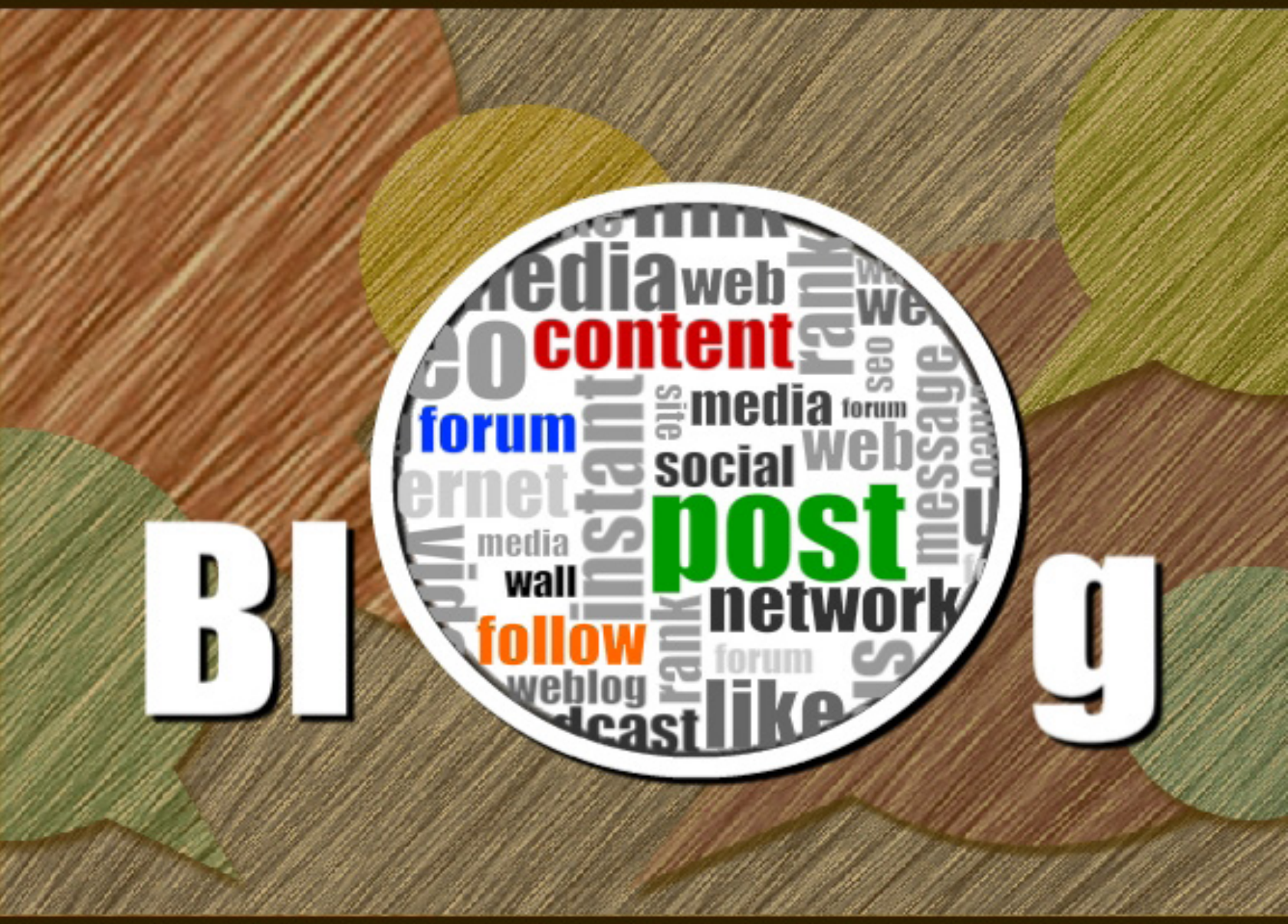
The knowledge and information in this book is the result of the concentrated effort of the Design Team, which is continuously striving to bring to you the latest, the best and the most relevant subject matter in Information Technology. As a part of Aptech's quality drive, this team does intensive research and curriculum enrichment to keep it in line with industry trends and learner requirements.

We will be glad to receive your suggestions. Please send us your feedback, addressed to the Design Centre at Aptech's corporate office.

Design Team

Onlinevarsity

your e-way to learning



Balanced Learner-Oriented Guide

for enriched learning available @

www.onlinevarsity.com

Table of Contents

Sessions

ASK to LEARN

Onlinevarsity
your e-way to learning

Questions *in your* mind?

are here to HELP

Post your questions in the ASK to LEARN section for solutions.

01 Introduction to Web Services

Welcome to the Session, **Introduction to Web Services.**

This session introduces Web services. It explains Service Oriented Architecture (SOA). It discusses the support provided by Java Enterprise Edition 7 (Java EE7) and its APIs. It explains Java APIs for XML Web Services (JAX-WS) and how Java Classes and Enterprise JavaBeans can be used for Web service implementation. Further, it explains how XML documents can be processed for Web services using Java APIs for XML Processing (JAXP). Finally, it describes Java APIs for XML Registry (JAXR), SOAP with Attachment API for Java (SAAJ), and Java Architecture for XML Binding (JAXB).

In this Session, you will learn to:

- Define Web services and describe their purpose
- Describe Service Oriented Architecture (SOA)
- Describe role of XML, SOAP, Registry standards, and WSDL in Web services
- Describe purpose of using JAXP in processing XML documents
- Explain parsing of XML document using SAX and DOM
- Describe JAX-WS
- Explain JAXR architecture, its components, and its interfaces
- Describe the purpose and features of SAAJ
- Describe purpose and limitations of JAXB and its components
- Explain the marshalling and unmarshalling processes

1.1 Introduction to Web Services

A Web service is a service available on the Web. In programming terms, Web service is a method made available on the Web.

Today a large variety of Web services are available on the Web. For example, many Websites provide services such as credit card validation, email-id verification, and so on. All you have to do is understand how to use the service, pass the required parameters to it, and invoke it. The service returns the result as a response. For example, for credit card validation service, you pass a credit card number as the parameter. The service may then return a true value to indicate that the credit card number was valid.

Web services are an integral part of the Web today. Moreover, their language and platform independence have made them widely successful. Figure 1.1 shows the Client-Server architecture.

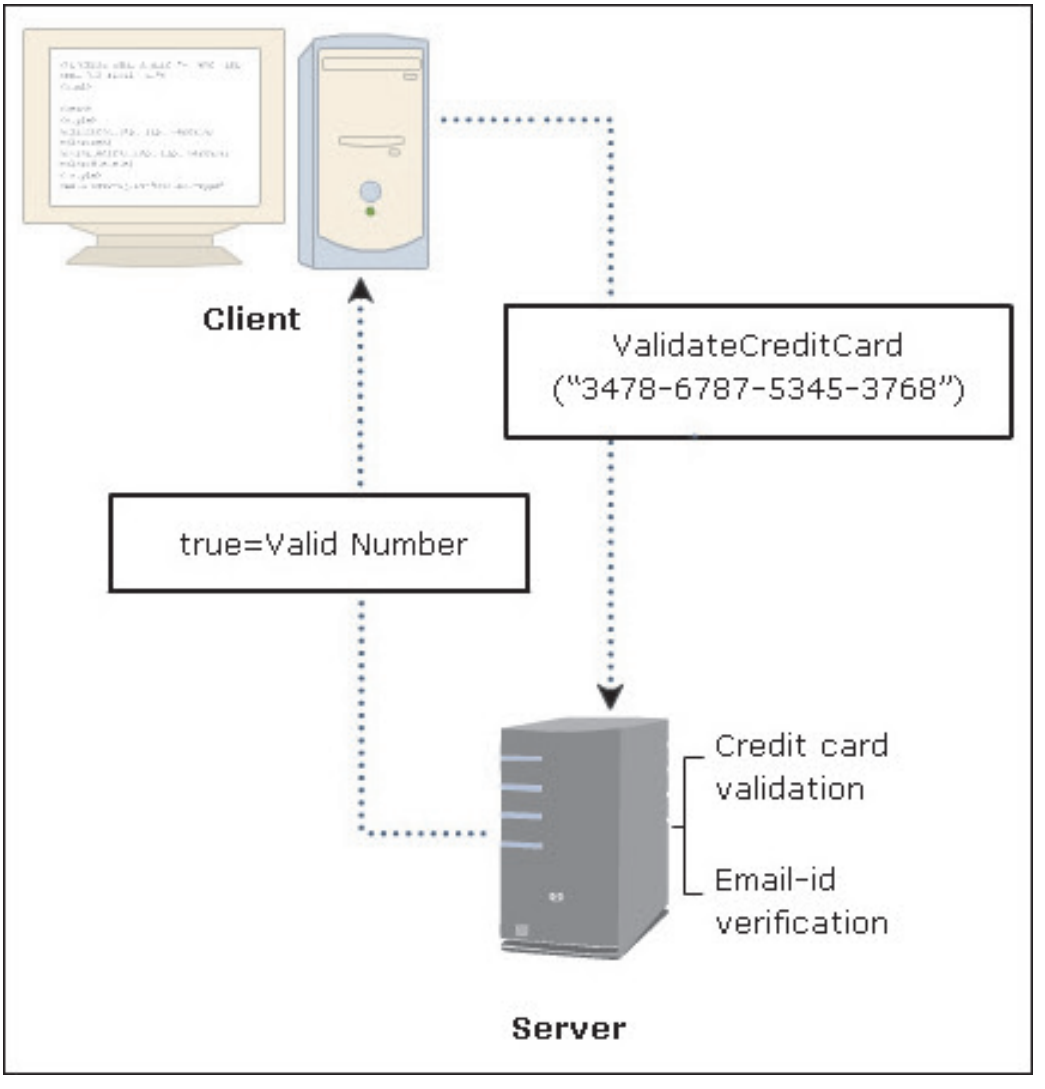


Figure 1.1: Client-Server Architecture

1.1.1 Characteristics of Web Services

Some of the characteristics of Web services are described in table 1.1.

Characteristic	Description
Accessibility	A Web service is accessible over the Web.
Communication Standards	Web services use standard Web protocols to interact with other Web services or application programs. The Web services and application programs use XML to exchange information. Hence, any node in the network that supports HTTP and XML can host and access Web services.
Integration	Web services are loosely coupled and are integrated only when required. The services are searched for in the service registry and are dynamically integrated at the required instance.

Table 1.1: Characteristics of Web Services

1.1.2 Uses of Web Services

Web services are widely used in:

- **Application-to-Application (A2A) Integration** – Generally, an organization uses different applications to process and maintain its data over a network. Web services are widely used to communicate and exchange data between various applications within an organization. A2A integration is also known as Enterprise Application Integration (EAI).
- **Business-to-Business (B2B) Communication** – When different applications belonging to multiple organizations, typically business partners, exchange data using Web services, it is called B2B communication. When any two organizations are business partners, the varied applications deployed in these organizations should be able to exchange information with each other. This is made possible by Web services.

The uses of Web services are illustrated in figure 1.2.

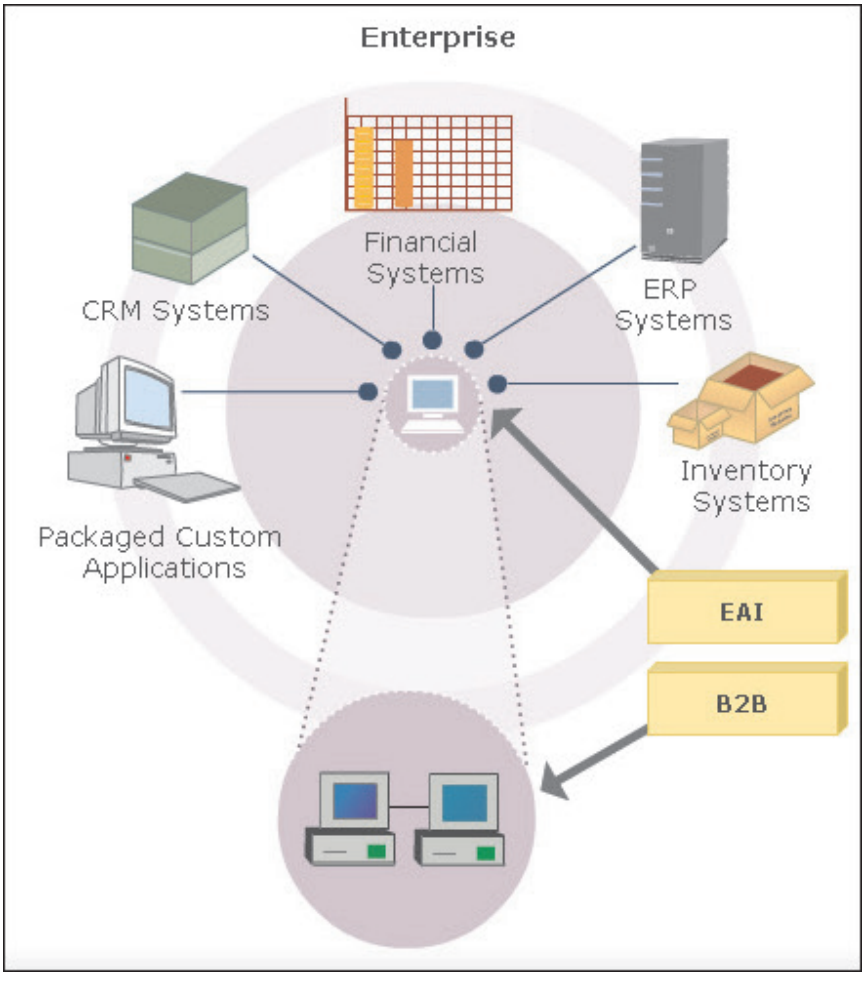


Figure 1.2: Uses of Web Services

1.1.3 Advantages of Using Web Services

Web services have come a long way since their advent. The advantages they provide have resulted in a steep rise in their popularity. Some of these benefits are as follows:

- **Freedom to use any platform and language for code development** – Since Web services are interoperable, developers are free to use any software platform, architecture, and programming language for development.
- **Flexibility to reuse existing software code** – Extending software code as Web services allow enterprise application developers to use and reuse them. Reusing existing code saves time and effort.
- **Availability of wide variety of tools** – A large variety of tools is available for developing Web services. Organizations now have a wide variety of options to choose from.
- **Wider market for services** – Extending applications and services as Web services avail organizations a vast range of clientele.

1.1.4 Types of Web Services

There are two types of Web services, namely Big Web services and Representational State Transfer (RESTful) Web services.

- **Big Web Services**

  Big Web services are offered using the SOAP standard, message architecture, and message formats of the XML language. In addition, the operations of this service are described in Web Services Description Language (WSDL) – an XML language – which is machine-readable. Java has developed the Java API for XML Web Services (JAX-WS) to support big Web services for Websites developed using the JAVA EE.

  The SOAP message format and the WSDL interface definition language have become popular and are used by many development tools such as NetBeans IDE. These tools simplify the development of Web service applications.

- **RESTful Web Services**

  RESTful Web services employ the Java API for RESTful Web services (JAX-RS) functionality. JAX-RS is a Java programming language API, which supports building of Web services as per the REST architecture.

  RESTful Web services are better incorporated with HTTP than SOAP-based services. They do not require XML messages or WSDL service-API definitions. RESTful Web services are economical. They can be easily adopted. You can use development tools like NetBeans IDE to simplify the development of Web services.

  RESTful design can be used when Web services are stateless. The performance can be enhanced by leveraging the caching infrastructure. However, for most servers, caches are limited to the HTTP GET method. There is no formal description of the Web services interface. Hence, many commercial applications provide value-added toolkits that describe the interfaces to developers in popular programming languages. REST is widely used in limited-profile devices where the headers and additional layers of SOAP elements are restricted.

1.2 Service Oriented Architecture (SOA)

Service Oriented Architecture (SOA) involves building an infrastructure that provides location and implementation transparency. Location transparency allows a service to be located without any modification in the source code.

Similarly, implementation transparency allows an end user to use the service in an application regardless of the platform and programming language the application is using.

1.2.1 Components of SOA

The components of a typical SOA are described in table 1.2.

Characteristic	Description
Service Broker	This component publishes information related organizations and their services. It also provides information on the accessibility and usage of the services provided.
Service Provider	This component allows service consumers to use the services depending on the service cost, availability of the service, and security.
Service Consumer	This component is another application or service used in an organization internally/externally. It makes use of a service broker to locate one or more services. Then, it connects to the service provider to make use of services.

Table 1.2: Components of SOA

For example, assume that an employee wants to know the number of leaves he/she can avail in a year. A service consumer can use the service published on the service broker to provide the number of leaves an employee can avail in a year. Figure 1.3 illustrates the SOA.

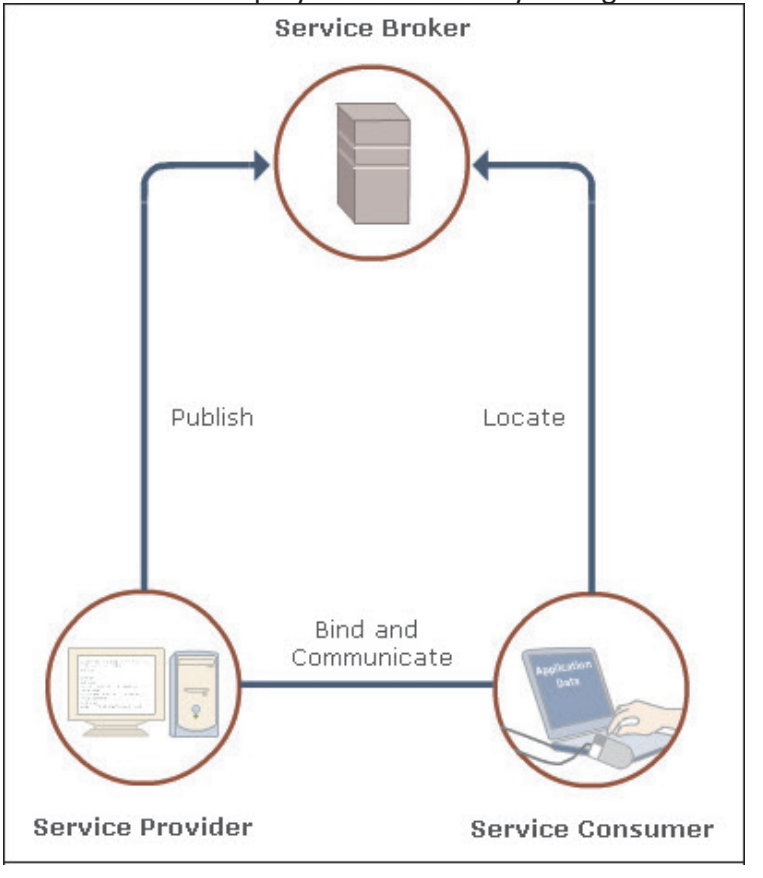


Figure 1.3: SOA

1.2.2 SOA Implementation by Web Services

Web service is created by service provider and hosted on the server. Similar to SOA, it uses XML to locate and implement transparency. The information related to accessibility and usage is published in registries such as UDDI or ebXML, which serve as service brokers. The service consumer identifies the required service and gets its information. Based on this information, the service consumer invokes the Web service that is placed on the service provider's server. XML is used for communication and Hypertext Transfer Protocol (HTTP) is used as the communication protocol. Hence, Web service can be used between different platforms. Figure 1.4 illustrates the registry.

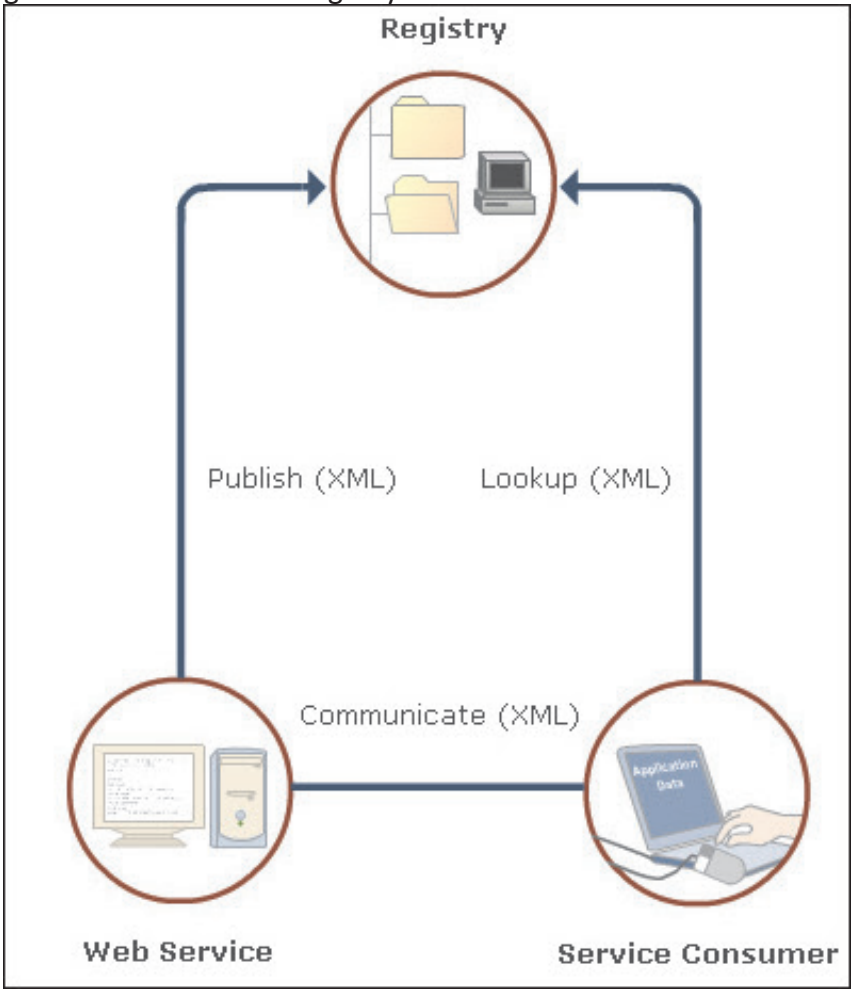


Figure 1.4: Registry

1.3 Web Service Standards

There are certain standards to:

- Exchange information over the Web

- Transfer data
- Maintain registries
- Ensure interoperability between different platforms

1.3.1 Extensible Markup Language (XML)

The most common markup language to communicate over the Web is XML. It is the most flexible language that allows applications to communicate irrespective of their geographic locations or software platforms. This is possible because other standards such as SOAP, WSDL, and UDDI are developed using XML. Thus, you can implement and execute Web services by exclusively using XML.

1.3.2 Simple Object Access Protocol

Simple Object Access Protocol (SOAP) is a standard protocol that facilitates transfer of XML data among various applications. With the help of SOAP, applications can send and receive data in a common format. There is no need to add any technology either by the service provider or by service consumer to make SOAP functional.

The SOAP message, also called as SOAP envelope, is an XML document. It has a header and a body that has message data enclosed in an envelope. The structure of a SOAP message is depicted in figure 1.5.

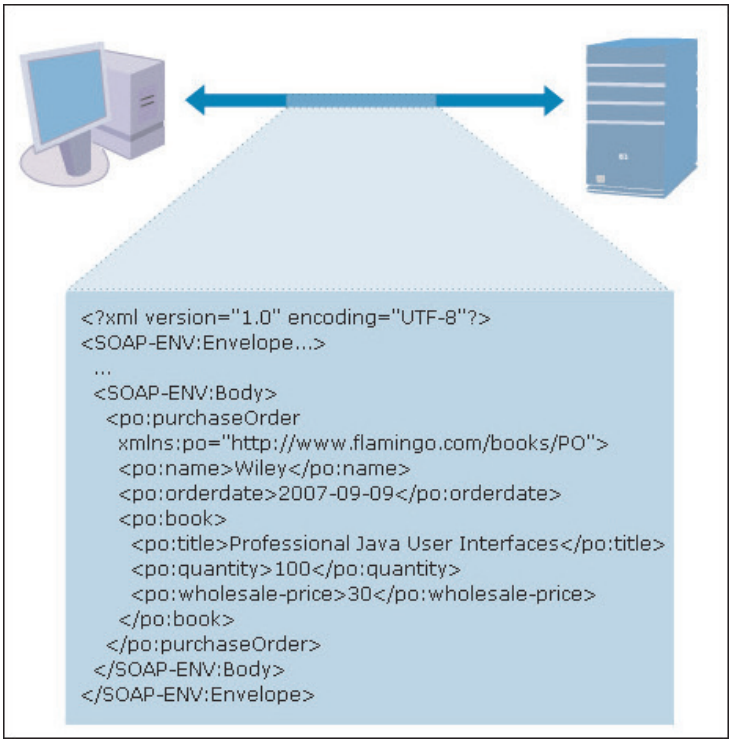


Figure 1.5: Structure of a SOAP Message

1.3.3 Web Service Registries

Web service registry is a service that allows service providers to publish services on the Web. To use this information, you should first create a registry account and then add information including name and service of service provider, Web service name, type of service, and contact information of the service provider.

Every service provider is listed in the registry. If a consumer needs a service, the entire registry is explored to locate the required service. Figure 1.6 illustrates the Web service registries.

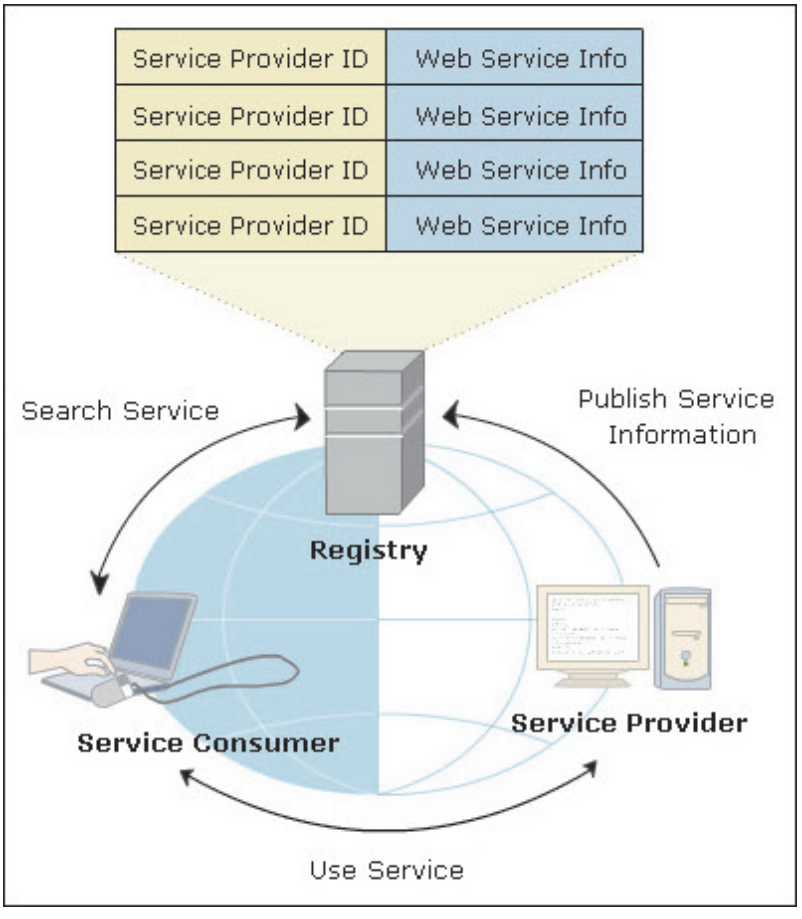


Figure 1.6: Web Service Registries

1.3.4 Web Services Description Language

Web services can be used by first retrieving a document (written in WSDL) from the registry. The document includes the following information:

- Web service location that refers to the WSDL file.
- Parameters that have to be passed to the methods (Method is a set of code. It is referred by a name.)

- Methods provided by the Web service.
- Syntax to invoke method (Using a method name, you can invoke the code it represents.).
- Return type of methods.

It is not required to code WSDL manually as many of the Integrated Development Environments (IDEs) automatically generate the document. The location of WSDL file is illustrated in figure 1.7.

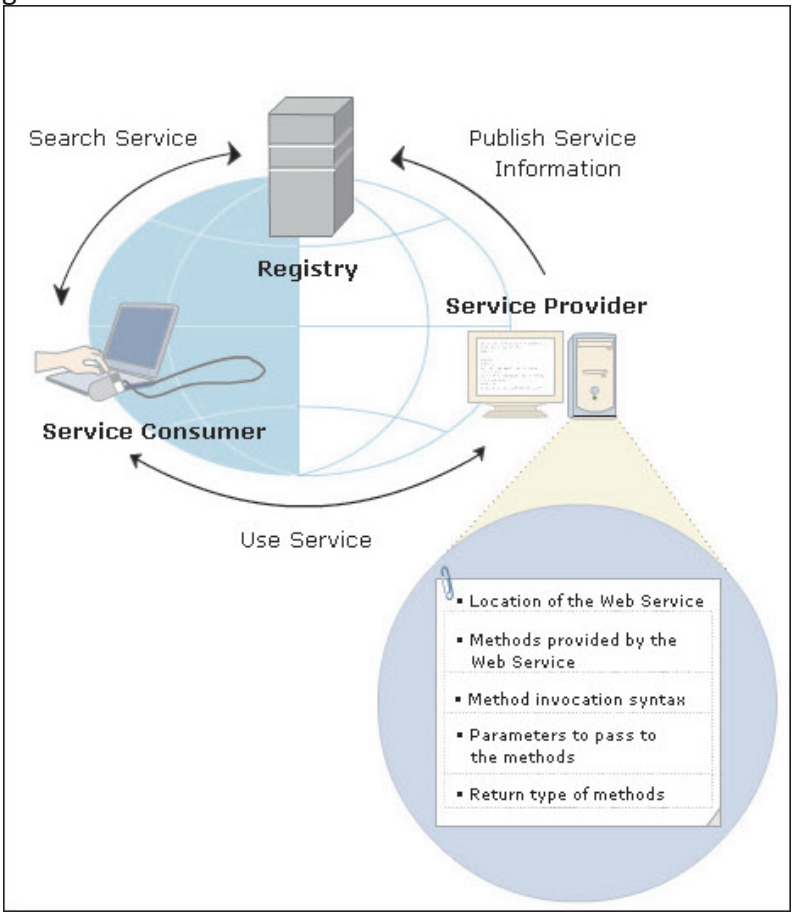


Figure 1.7: Location of WSDL File

1.3.5 Future Trends in Web Services

The key concerns in Web services are security, complicated business processes, and time-consuming transactions. Organizations such as the Web Services Interoperability (WS-I) try to enhance the current standards and trends of Web services. WS-I has made the Basic Profile 2.0 specification to define and publish a set of cross-platform standards to ensure interoperability.

1.4 Java EE 7 Web Services Framework

The J2EE platform consists of various Web service APIs to support and implement Web services.

1.4.1 Java EE Web Services APIs

There are four Web service APIs in Java EE that implement Web service applications. They are described in table 1.3.

Web Service API	Description
Java API for XML-based Web services (JAX-WS)	This API enables communication with Java and non-Java Web services.
SOAP with Attachments API for Java (SAAJ)	This API helps read, manipulate, create and transmit SOAP messages, and process SOAP header blocks in JAX-RPC. It complies with SOAP 1.1 and the SOAP Messages with Attachments specification.
Java API for XML Registries (JAXR)	This API allows an application to access registries such as UDDI or ebXML. It simplifies publishing and querying of Web services.
Java API for XML Processing (JAXP)	This API enables users to use DOM2 and SAX2 interfaces. Apart from this API, standard Java APIs can also read, write, and modify XML documents.

Table 1.3: Web Service APIs

1.4.2 Benefits of Using the Java EE Platform

The Java EE platform helps to:

- improve development of applications using component-based model
- take the support of Web service standards and WS-I basic profile
- make portable applications and interoperable services
- expand distributed applications with little effort
- declare component security requirements

1.5 Java EE 7 Web Services Framework

Java EE 7 Web Services Technologies use Java API for XML Web Services (JAX-WS). Web services are implemented using stateless session bean. The Web Services Technologies of Java EE 7 are listed in table 1.4.

Web Services Technology	Java Specification Request (JSR)	Description
Java API for RESTful Web Services (JAX-RS) 2.0	339	It assists in the creation of an API that supports RESTful Web services in the Java Platform.
Implementing Enterprise Web Services 1.3	109	It defines the programming model and runtime architecture to implement Web services in Java.
Java API for XML-Based Web Services (JAX-WS) 2.2	224	It is the next generation Web services API replacing JAX-RPC 1.0.
Web Services Metadata for the Java Platform	181	It defines an annotated Java format that uses Java Language Metadata (JSR 175). It simplifies the definition of Java Web services in a J2EE container.
Java API for XML-Based RPC (JAX-RPC) 1.1 (Optional)	101	It supports emerging industry XML-based RPC standards.
Java APIs for XML Messaging 1.3	67	It helps to package and transport business transactions. It uses on-the-wire protocols defined by ebXML.org, Oasis, W3C, and IETF.
Java API for XML Registries (JAXR) 1.0	93	It assists a set of distributed Registry Services to enable business-to-business integration between enterprises. It uses the protocols defined by ebXML.org, Oasis, and ISO 11179.

Table 1.4: Web Services Technologies

1.5.1 Implementation of Web Service Using Java Class

There can be two types of endpoints – namely, JAX-WS endpoint and EJB endpoint in J2EE Web services. The steps to develop a JAX-WS-based endpoint are as follows:

1. **Defining remote interface:** In this step, a remote interface is defined. This interface has all methods as a part of the service that would be exposed in the service. These methods throw java.rmi.RemoteException. This interface also extends java.rmi. Remote interface.

2. **Implementing remote interface:** In this step, the class that implements the interface is defined. This class has all the methods declared in the remote interface.

3. **Building service:** In this step, the .java files of remote interface and implementation class are compiled. The WSDL file and mapping file are created using the wscompile tool. WSDL file has Web service description and mapping file has information that correlates the mapping between the remote interface and the definitions in WSDL file. This step can be easily done in many Integrated Development Environments (IDEs).

4. **Packaging and deployment:** In this step, the class files, WSDL file, and mapping file are packaged into a WAR file and deployed on a server. This step is performed by using the Graphical User Interface (GUI), which is found in many IDEs.

1.5.2 Enterprise JavaBeans Endpoint

A J2EE Web service can also have an EJB endpoint. The steps to implement an EJB endpoint are similar to the steps to implement JAX-WS endpoint but for an exception. Here, the implementation class is a stateless session bean, which can implement all the methods defined in a remote interface. This approach can be used when it is possible to expose a stateless session bean as a Web service.

1.6 JAXP

The JAXP uses applications written in Java to validate and parse the XML document. It uses either Simple API for XML (SAX) or Document Object Model (DOM) Java standards to parse XML documents. These APIs can read, write, and modify XML documents by using XML parser.

1.6.1 Processing XML Documents Using JAXP

JAXP uses Extensible Stylesheet Language Transformation (XSLT) engines to transform XML documents from one format to another. Figure 1.8 shows how JAXP processes XML documents.

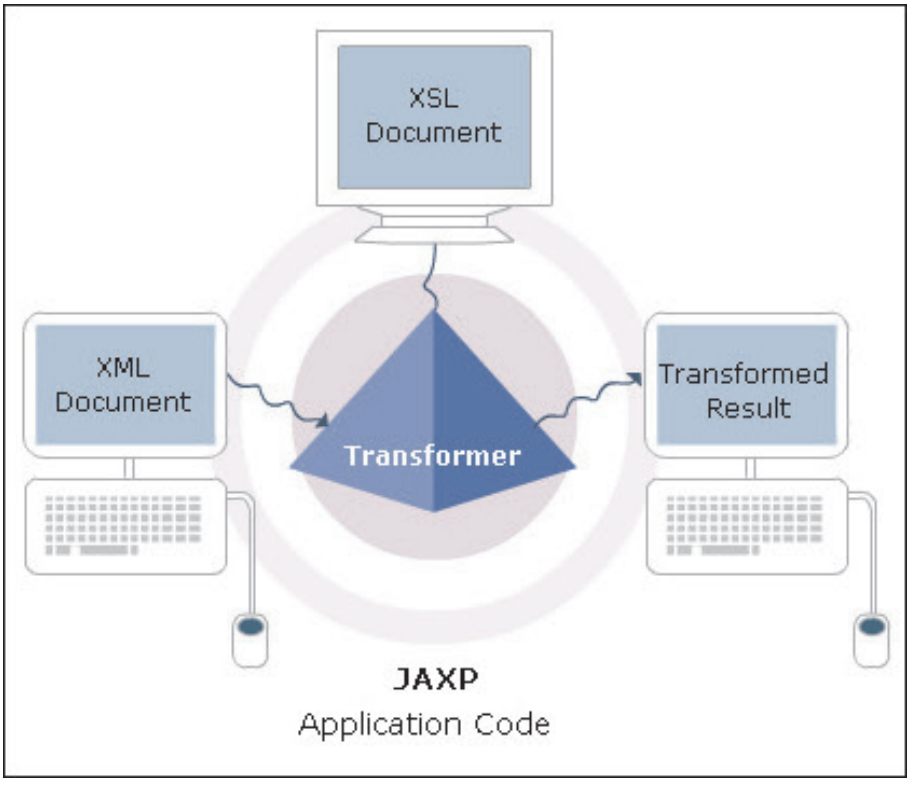


Figure 1.8: Processing XML Document Using JAXP

1.6.2 Classes and Packages of JAXP

A package is a namespace attributed to the classes based on their category or functionality. You use these packages to organize the classes. The packages that define the JAX API are described in table 1.5.

Package	Description
javax.xml.parsers	It defines SAXParserFactory and DocumentBuilderFactory classes. These classes provide a common interface for SAX and DOM parsers to process XML documents.
javax.xml.transform	It defines the generic APIs. These APIs process instructions to transform source to result.
javax.xml.transform.dom	It implements DOM-specific transformation APIs. Using the DOMSource class, the client that implements this API can specify a DOM Node as the source of the input tree.
javax.xml.transform.sax	It implements SAX2-specific transformation APIs.

Table 1.5: Packages of JAX API

1.6.3 SAX Parser

SAX processes the XML documents sequentially in which the data elements are interpreted and converted into a series of events. SAX is very fast as it does not load the document into memory. The SAX parser is depicted in figure 1.9.

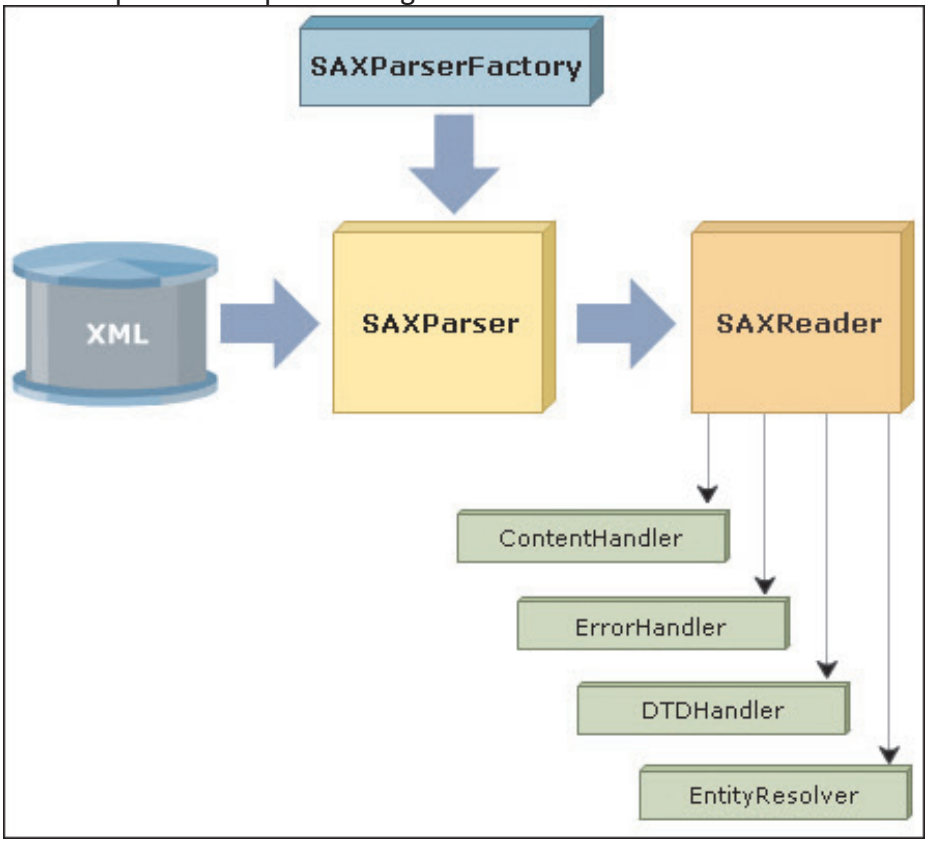


Figure 1.9: SAX Parser

The key classes/interfaces that are used to parse XML documents are described in table 1.6.

Class/Interface	Description
DefaultHandler	This class is present in the org.xml.sax.helpers package. It acts as a default base class for SAX2 event handlers.
SAXParserFactory	This class is present in the javax.xml.parsers package. It returns the SAXParser and the exception classes to report errors.
SAXParser	This class is present in the javax.xml.parsers. It defines the API that wraps an XMLReader implementation class.
XMLReader	This class is present in the java.io package. It is an interface to read an XML document by using callbacks.

Table 1.6: Classes/Interfaces to Parse XML Documents

1.6.4 SAX Parser Implementation

The process of parsing an XML document is done with the help of JAXP APIs and SAX. Code Snippet 1 depicts the step-by-step code to parse an XML document using a SAX-based parser.

Code Snippet 1:

```
//Step 1
public class SAXParsing extends DefaultHandler{
     public void readDocument() {
          //Step 2
          SAXParserFactory spFactory = SAXParserFactory.
newInstance();
          spFactory.setValidating(true);
          //Step 3
          SAXParser saxp = spFactory.newSAXParser();
          //Step 4
          XMLReader xReader = saxp.getXMLReader();
          //Step 5
          xReader.setContentHandler(this);
          //Step 6
          xReader.parse(XMLDocument);
     }
}
```

This code performs the following steps:

Step 1: A class named SAXParsing is created. This class extends the DefaultHandler class.

Step 2: An instance of the SAXParserFactory class, named spFactory, is created, and the validation property of this instance is set to true.

Step 3: An instance of the SAXParser class, named saxp, is obtained from the SAXParserFactory instance.

Step 4: The encapsulated SAX XMLReader is retrieved to read character streams.

Step 5: The ContentHandler property of XMLReader is set to allow the application to register a content event handler.

Step 6: The XML document is parsed using the instance of the XMLReader interface.

1.6.5 DOM Parser

A DOM parser accesses XML documents randomly and creates a tree from its elements. This process splits the documents into fragments and requires more memory. It is easy to create and modify XML documents by using the in-memory content tree.

The packages in DOM parser are described in table 1.7.

Package	Description
org.w3c.dom	This package defines DOM programming interfaces for XML documents. Its primary interfaces are Document and Node. Document is a HTML/XML document and Node is the primary data type for DOM.
javax.xml.parsers	This package defines the DocumentBuilder class and DocumentBuilderFactory class to process XML documents.

Table 1.7: Packages in DOM Parser

DOM Parser Implementation

The process of parsing an XML document can also be done with the help of JAXP APIs and DOM. Code Snippet 2 depicts the step-by-step code to parse an XML document using a DOM-based parser.

Code Snippet 2:

```
public void readDocument(){


// step 1
DocumentBuilderFactory dbFactory = DocumentBuilderFactory.
newInstance();
dbFactory.setvalidating(true);


// step 2
DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();


// step 3
Document dcm = dBuilder.parse(XMLDocument);
  // parse the tree created - node by node
}
}
```

This code performs the following steps:

Step 1: A class named DOMParsing is created for parsing an XML document. An instance of the DocumentBuilderFactory class named dbFactory is created and the validation property of this instance is set to true.

Step 2: A DocumentBuilder object named dBuilder is created using an instance of DocumentBuilderFactory class.

Step 3: The instance of the DocumentBuilder class parses the input file by invoking the parse method and passing the document to be parsed named XMLDocument.

1.7 JAX-WS

In Java EE 7, the JAX-WS technology is used to develop Web services and clients that use XML for communications. Using JAX-WS, you can create message-oriented or remote procedure call-oriented (RPC-oriented) Web services. In JAX-WS, an XML-based protocol, such as SOAP, defines the format used for Web service invocation and responses. JAX-WS uses SOAP messages or XML files to transmit the invocation and responses over HTTP. This feature allows a JAX-WS client to access Web services that are running on non-Java platforms and allows clients running on non-Java platform to access JAX-WS Web services.

JAX-WS makes programming easier by allowing to write Java code for specifying Web service methods and one or two classes that implement the methods. In addition, the JAX-WS client programs create a proxy or a local object for the Web service they want to utilize and invoke the Web service methods on the proxy. This makes client programming also very easy and less complex. When using JAX-WS, the JAX-WS runtime generates SOAP messages for the API calls and parses the SOAP messages from responses. As a result, you do not need to write code for generating or parsing SOAP messages.

1.7.1 Role of JAX-WS in Web Services

JAX-WS performs the following tasks in a Web service:

- Interoperates with SOAP-based Web services using WSDL
- Maps the data types between XML and Java
- Invokes methods on the generated stubs
- Supports standard Internet protocols such as HTTP
- Enables portability of service endpoints and service clients across JAX-WS implementations

The role of JAX-WS in Web services is illustrated in figure 1.10.

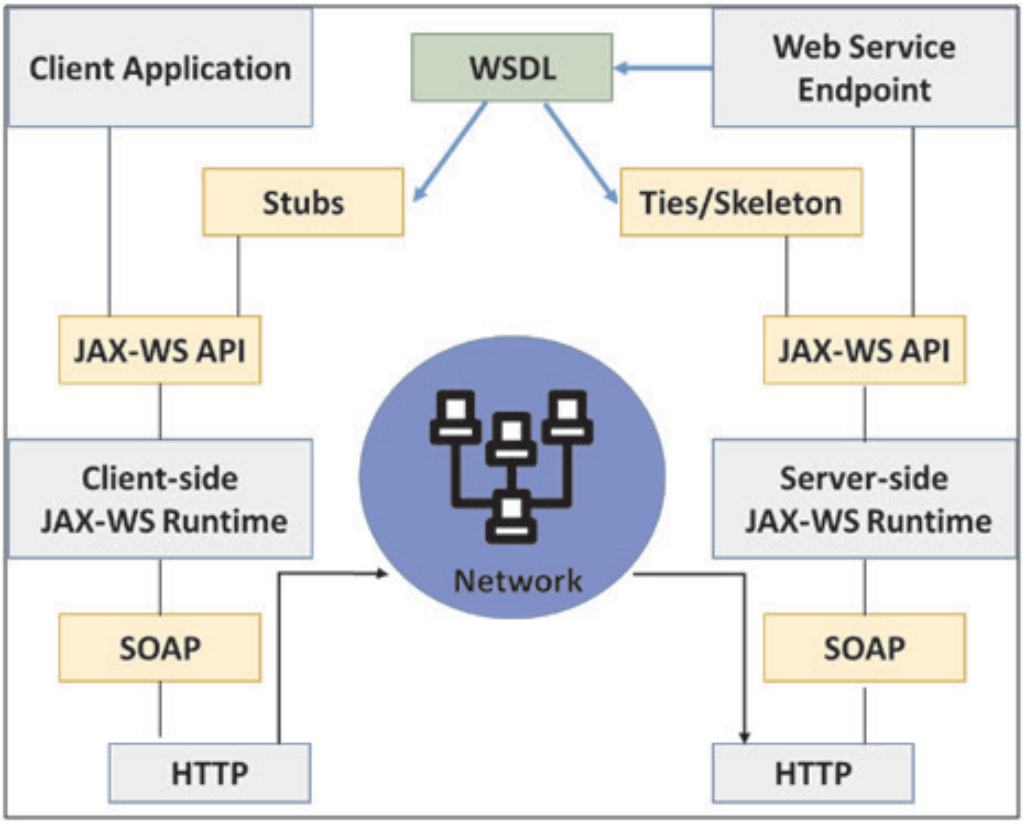


Figure 1.10: JAX-WS with Web Services

1.7.2 Features of JAX-WS

JAX-WS functions by using WS Basic Profile (BP) conformant Web service technologies and by providing object-oriented API that can be used to communicate even with non-Java platforms. Some of the features of JAX-WS framework are as follows:

- It enables interoperability with any SOAP-based Web services using WSDL.
- It provides mapping of data types between XML and Java.
- It invokes methods on the generated stubs.
- It supports standard Internet protocols such as HTTP.
- It provides portability of Service endpoints and service clients across JAX-WS implementations.
- It defines a common programming model for Java EE endpoints and Web service clients.
- It allows hosting of Web service endpoints that can be accessed by non-Java client applications.
- It is designed as a Java API to facilitate interoperation of Java EE applications.

1.7.3 Benefits of Using JAX-WS

Using JAX-WS reduces the complexity for developers in several ways. The key benefits are as follows:

- Standardization of SOAP requests and responses.
- Standardization of parameter marshalling and unmarshalling.
- Simplification of developers' role as SOAP creation and marshalling/unmarshalling tasks are offered through a library or a tool.
- Support for different mapping scenarios, such as XML to Java, Java to XML, WSDL to Java, Java to WSDL, and WSDL/XML and Java.

1.7.4 Client Requests to JAX-WS Service

To use a Web service, the client application should perform the following steps:

1. First, the request from client is passed through client-side JAX-WS runtime.
2. The runtime maps the request to XML and converts it to a SOAP form.
3. The message is then sent to a server on the network.
4. At the server, the SOAP message is received by a JAX-WS runtime.
5. Finally, the runtime applies the XML to Java mapping and maps the request with its parameters to the corresponding Java method.

To make a request, the client application can use the pre-created static classes or classes/interfaces generated at runtime. Figure 1.11 depicts the client requests to JAX-WS service.

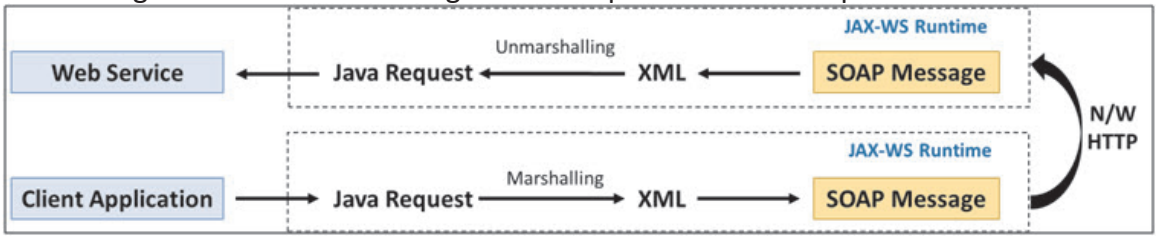


Figure 1.11: Client Request to JAX-WS Service

Code Snippet 3 demonstrates a simple example of implementing JAX-WS.

Code Snippet 3:

```
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public class Welcome {
```

```
    private final String message1 = "Welcome, ";
    private final String message2 = "to WebService Course.";

    public void Welcome()
{
    }

    @WebMethod
    public String sayWelcome(String name)
{
    return message1 + name + message2;
    }
}
```

The code displays a welcome message to the users who call the Web service. The name of the user can be passed as a String while invoking the Web service method.

In this code, the following two annotations have been used:

- **@WebService**: This annotation is used to specify that this Java class implements a Web service. The attributes of this annotation are:
  - **name:** Specifies the name of the Web service. The default value of this attribute is the unqualified name of the Java class.
  - **targetNamespace:** Specifies the XML namespace to be used for WSDL and XML elements that the Web service generates.
  - **serviceName:** Specifies the service name for the Web service. The default value of this attribute is the unqualified name of the Java class appended with the string 'Service'.
  - **wsdlLocation:** Specifies the URL of the predefined WSDL document to be used for the Web service. If a WSDL file is specifies, the Web service does not generate a WSDL file and throws an error if the JWS file does not map to the elements specified in the WSDL file.
  - **endpointInterface:** Specifies the fully qualified name of an existing service endpoint interface file to be used for the Web service.

  All these attributes are optional.
- **@WebMethod**: This annotation is used to indicate an operation provided by the Web service. The attributes of this annotation are:
  - **operationName:** Specifies the name of the operation provided by the Web service. The default value of this attribute is the name of the method.

- **action:** Specifies the action for the operation.

These attributes are optional.

1.8 Registry Standards

A registry is a place where interfaces are published by the service to make the clients know about the service. An XML-based registry requires an expanded information model and query capabilities that are suitable for e-business and Web services standards.

XML-based registries support more complex queries and enable more accurate searches for Web services. They facilitate application-to-application communication and interoperability. They have standards for registering, deregistering, and looking up Web services across different platforms, systems, and languages. The most popular registries are the Universal Description, Discovery, and Integration (UDDI) and the electronic business XML (ebXML) registries.

1.8.1 UDDI Registry Standard

UDDI is an XML registry which can access remote clients and search the Web services. It has a standard mechanism to store information about organizations and their Web services. Figure 1.12 depicts the UDDI registry.

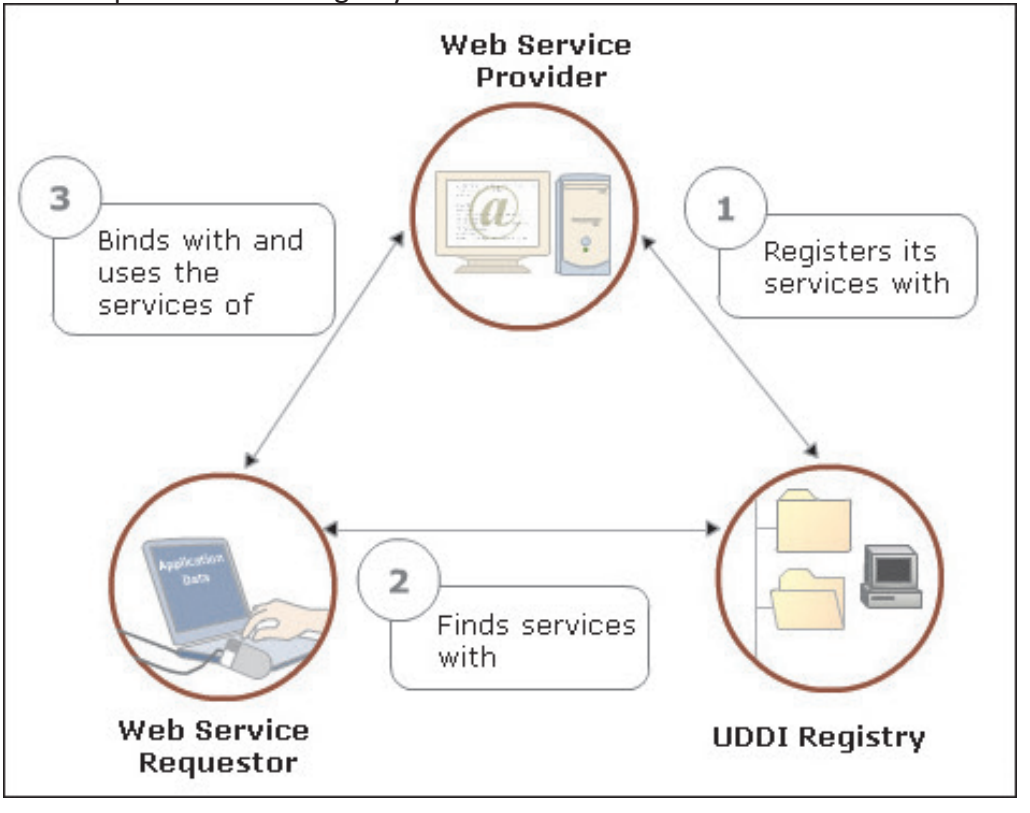


Figure 1.12: UDDI Registry

There are four core elements in the UDDI information model. These are described in table 1.8.

Element	Description
businessEntity	This element describes all information including the name, description, and contact details of a business.
businessService	This element groups related services of an organization.
bindingTemplate	This element provides instructions to invoke a remote Web service.
tModel	This element contains information such as name, publishing organization, and URL pointers to the actual specifications themselves.

Table 1.8: Core Elements in the UDDI Information Model

1.8.2 ebXML Registry Standard

The Electronic Business XML (ebXML) is a business-to-business XML framework. It facilitates electronic business over the Internet. You can advertise and also find out information about businesses. You can also publish Web service descriptions in this registry.

The ebXML registry is a metadata registry and a repository that can hold arbitrary content. Its repository can have metadata, technical specifications, and related artifacts. It stores information about Collaboration-Protocol Profile (CPP) and Collaboration-Protocol Agreement (CPA).

ebXML information model enables data validation for improved integrity of registry data. It also facilitates packaging (or grouping) of related registry objects. It allows both synchronous and asynchronous communication. It supports digital-signature-based authentication, validation, and authorization.

1.9 JAXR API

The Java API for XML Registries (JAXR) API is an abstract uniform Java API. It has a single set of APIs that can access many XML registries, including UDDI and ebXML registries.

1.9.1 JAXR Architecture

There are two parts in JAXR architecture, namely JAXR client and JAXR provider. First, the client uses JAXR interfaces and classes to request access to a registry. A connection is created by using the createConnection() method through the ConnectionFactory interface. After receiving the requests, JAXR provider transforms these methods into registry-specific methods and transfers them to the target registry providers. The registry receives the requests and processes them. After this step, the process is reversed.

The registry returns the response to JAXR provider where the response is transformed to JAXR response. The provider sends this response back to the JAXR client.

The layered architecture of JAXR accommodates different functionalities from different registry providers. There can be two capability profiles, namely level 0 and level 1. Level 0's basic features support business-focused APIs, whereas, level 1's advanced features support generic APIs. Figure 1.13 illustrates the JAXR architecture.

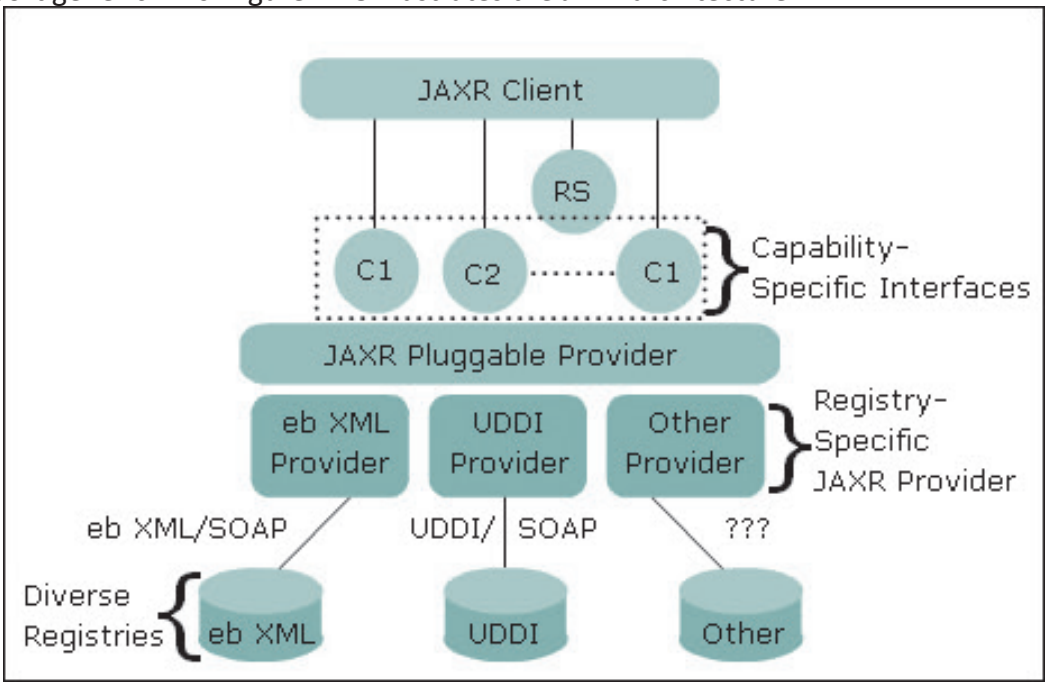


Figure 1.13: JAXR Architecture

1.9.2 JAXR Information Model

The JAXR information model is based on the ebXML Registry Information Model (RIM). It also supports UDDI and has all the data types defined in the UDDI Data Structure Specification. It creates JAXR information model objects when the JAXR client invokes life-cycle and query management methods on the JAXR provider.

JAXR information model provides a more intuitive and natural interface to developers. Some of the interfaces that define the JAXR information model are described in table 1.9.

Interface	Description
Organization	Organization instances are RegistryObjects that provide information on an organization that has been published to the underlying registry.
Service	Service instances are RegistryObjects that provide information on services offered by an organization.

Interface	Description
ServiceBinding	ServiceBinding instances are RegistryObjects that represent technical information on how to access a specific interface offered by a Service instance.
Concept	Concept instances represent an arbitrary notion or concept. It can be virtually anything.
Classification Scheme	ClassificationScheme instances represent a taxonomy that can be used to classify or categorize RegistryObject instances.
ExternalLink	ExternalLink instances provide a link to content managed outside the registry using a URI.
ExternalIdentifier	ExternalIdentifier instances provide identification information to a RegistryObject.
Classification	Classification instances classify a RegistryObject instance using a classification scheme
PostalAddress	PostalAddress instances provide address information for a user and an Organization.

Table 1.9: Interfaces of JAXR API

The UDDI provider exposes the registry though a Web-based HTML interface. Then, the registry may be queried for different fields by using the relevant methods from the BusinessQuery Manager interface.

1.9.3 Connecting to Registry

A JAXR client establishes a connection with a registry by:

- Creating a connection factory
- Configuring connection factory
- Creating a connection
- Obtaining a service object

Creating Connection Factory

Many of the JAXR providers supply one or many preconfigured connection factories. To create an instance of connection factory, first connect to the JNDI service by creating an instance of initial context. Next, use the instance to lookup the connection factory using its JNDI name.

Configuring Connection Factory

To connect to the registry, creates a set of properties that will be used to specify the URL or URLs of the registries that are being accessed by the application. For example, you can use the queryManagerURL and lifeCycleManagerURL properties of the Properties object to specify the URLs of the query service and the publishing service for the registry.

Creating a Connection

The next step is to create a connection instance. This is done by invoking the createConnection() method on the connection factory instance.

Obtaining Service Object

The JAXR client uses the getRegistryService() method to retrieve a RegistryService object. The registry service instance is used to create instances of BusinessQueryManager and BusinessLifeCycleManager. The service information from the registry is retrieved by using BusinessQueryManager instance. The BusinessLifeCycleManager instance is used to add, edit, or remove service information from the registry.

1.9.4 Publishing Data on Registry

After getting registry object, service and service provider's information can be added to it. This can be done by:

- Creating an organization instance
- Associating one or more services with it
- Adding this information to the registry

Creating an Organization Instance

As the first step, an organization is created by using createOrganization() method on the BusinessLifeCycleManager instance. Then, an instance of User class is created and setter methods are used to store contact information about the organization. This information is associated with the organization by invoking the setPrimaryContact() method and passing the user instance to it.

Adding Services and Service Binding

An instance to store various services of an organization is created. Then, a service instance is created by using the createService() method. The setDescription() method describes the service provided. The name, description, and a unique key for each service object are generated when it is registered. When the user searches for a service, the registry sets the service's name. Service bindings are stored using a new collection instance.

The registry is informed to ignore Uniform Resource Identifier (URI) validation. A fictitious URI is provided for the service and service binding is added to the service binding collection instance. Finally, the service collection instance is passed to the addServices() method, which binds the organization instance with the service instance.

Publishing an Organization

A collection instance is created to store organization information. This instance contains contact information, service, and service binding information. These details are saved to the registry by using saveOrganizations() method. Then, the getExceptions() method is used to check whether the organization information was saved successfully. The method returns null if the details are saved successfully.

Then, the collection of keys is retrieved using the getCollection() method. A key is created for every organization instance stored in the registry. An iterator called keyIter is created using the iterator() method. The elements in the keyIter instance can be retrieved using the getID() method.

Code Snippet 4 demonstrates the steps to publish a Web service to UDDI.

Code Snippet 4:

```
public static void main(String[] args) throws JAXRException{
try{
//Setting the properties for the ConnectionFactory
Properties enviro = new Properties();
enviro.setProperty("javax.xml.registry.queryManagerURL",
QUERY_URL);
enviro.setProperty("javax.xml.registry.lifeCycleManagerURL"
, PUBLISH_URL);
enviro.setProperty("javax.xml.registry.factoryClass",
"com.sun.xml.registry.uddi.ConnectionFactoryImpl");

//creating a connection
ConnectionFactory confac = ConnectionFactory.newInstance();
confac.setProperties(enviro);
Connection conn = confac.createConnection();

//Authenticating the UDDI userName and Password
PasswordAuthentication passwdAuth = new
PasswordAuthentication(uddiUserName,
uddiPassword.toCharArray());
Set credentials = new HashSet();
credentials.add(passwdAuth);
conn.setCredentials(credentials);
```

```
//Obtaining a reference to the RegistryService,
//BusinessLifeCycleManager, and the BusinessQueryManager

RegistryService registryservice = conn.getRegistryService();
BusinessLifeCycleManager lifecyclemgr =
registryservice.getBusinessLifeCycleManager();
BusinessQueryManager querymgr =
registryservice.getBusinessQueryManager();

//Creating an organization object
Organization company =
lifecyclemgr.createOrganization("Aptech");
InternationalString description =
lifecyclemgr.createInternationalString("Leading IT
Trainers");

//Creating a user object
User contact = lifecyclemgr.createUser();
PersonName name = lifecyclemgr.createPersonName("John
Miller");
contact.setPersonName(name);

//creating and assigning the users telephone number
TelephoneNumber telnum =
lifecyclemgr.createTelephoneNumber();
telnum.setNumber("1800-420-8000");
Collection phonenumbers = new ArrayList();
phonenumbers.add(telnum);
contact.setTelephoneNumbers(phonenumbers);
```

```
//creating and assigning the users email address
EmailAddress email =
lifecyclemgr.createEmailAddress("enquiry@abcinc.com");
Collection emaillist = new ArrayList();
emaillist.add(email);
contact.setEmailAddresses(emaillist);

//Setting the user as the primary contact for the
//organization
company.setPrimaryContact(contact);

//Creating the Web service object
Collection servicelist = new ArrayList();
Service service = lifecyclemgr.createService("Courses");
InternationalString serviceDescription =
lifecyclemgr.createInternationalString("Available
courses..");
service.setDescription(serviceDescription);
//Creating the Web service bindings
Collection serviceBindings = new ArrayList();
ServiceBinding binding =
lifecyclemgr.createServiceBinding();
InternationalString bindingDescription =
lifecyclemgr.createInternationalString("Course Name");
binding.setDescription(bindingDescription);
binding.setAccessURI("http://www.aptech.org");
boolean b = serviceBindings.add(binding);
service.addServiceBindings(serviceBindings);
servicelist.add(service);
company.addServices(servicelist);
```

```
//Classifying the organization
ClassificationScheme scheme = querymgr.
findClassificationSchemeByName(servicelist,"Courses");
Classification classification = lifecyclemgr.
createClassification(scheme, "Course name", "Course
number");
Collection classificationlist = new ArrayList();
boolean c = classificationlist.add(classification);
company.addClassifications(classificationlist);
Collection organizationlist = new ArrayList();
organizationlist.add(company);

//making the final call to the registry to get a response
BulkResponse response =
lifecyclemgr.saveOrganizations(organizationlist);
Collection exceptions = response.getExceptions();
if (exceptions == null) {
Collection keys = response.getCollection();
Iterator iterator = keys.iterator();
Key key = (Key) iterator.next();
String uid = key.getId();
company.setKey(key);
}
else { // there is an exception
Iterator iterator = exceptions.iterator();
while (iterator.hasNext()) {
Exception exception = (Exception) iterator.next();
System.out.println("Exception...." + exception);
}
```

```
}
conn.close();
}
catch(Exception e) {
e.printStackTrace();
}
}
```

The code publishes data to the UDDI. It first configures the properties, such as queryManagerURL, lifeCycleManagerURL, and factoryClass, for the ConnectionFactory. Then, the code creates a connection and authenticates the UDDI credentials.

Next, the code obtains a reference to the RegistryService, BusinessLifeCycleManager and the BusinessQueryManager classes. The code then defines the data that needs to be published to the UDDI. It also defines the Web services to be offered and the Web service bindings for the same. The code also classifies the Web services offered into the appropriate category.

Finally, the code publishes all the data into the registry and awaits response from the registry. The registry will throw exceptions in case any errors are encountered. If there are any exceptions, the details of the exceptions will be displayed in the console. Else, the data will be published and added to the Web service.

1.9.5 Querying the Registry

A registry can be queried by using organization and service names related to the organization that needs information.

- **Finding Organizations by Name**

  First, a find qualifier collection is created to store find qualifiers. Then, a constant is added to show that the search results have to be sorted in descending order. Many such find qualifier constants are defined by the FindQualifier interface. Then, a name pattern collection namePat is created and the string of organization name is added to it. Using find qualifier and name pattern instances, the findOrganizations() method is passed. This method returns the search results in the form of an instance of BulkResponse. It has all the organization objects whose name begins with the organization name. To search organizations containing the string, the % symbol should be added on both sides of the string.

- **Finding Services and Service Bindings**

  First, an iterator instance is created from the collection returned by getCollection() method. A while loop is defined to process all the organization objects in the iterator. Inside the loop, the organization object is retrieved using the next() method. Then, an iterator is created from the collection returned by getServices() method. Next, the while loop is defined to process each service associated with the object.

Inside the nested while loop, a service object is retrieved using next() method. Then, the collection returned by getServiceBindings() method is used. A while loop is defined to process each service binding.

1.9.6 Removing Data from Registry

A registry allows you to remove any data that you have submitted to it. You use the key returned by the registry as an argument to one of the BusinessLifeCycleManager delete methods. The different delete methods include deleteOrganizations(), deleteServices(), deleteServiceBindings(), and deleteConcepts().

1.10 SAAJ

SOAP with Attachments API for Java (SAAJ) is an API that allows users to create and send SOAP messages with attachments by means of the javax.xml.soap package. Simple Object Access Protocol (SOAP) provides a common message format for Web services. It enables developers to produce and consume messages conforming to the SOAP 1.2 specification and SOAP with Attachments note. Attachments may be in the form of complete XML documents, XML fragments, or MIME-type attachments. In addition, developers can also use it to write SOAP messaging applications directly instead of using JAX-WS.

SAAJ messages follow SOAP standards, which prescribe the format for messages and also specify some things that are required. Using SAAJ API, you can create XML messages that adapt to the SOAP 1.2 and WebService-I Basic Profile 2.0 specifications.

1.10.1 Removing Data from Registry

There are two perspectives of SAAJ, namely Messages and Connections. The two types of SOAP messages are those that have attachments and those without attachments.

- **SOAP Message without Attachments**

  The SAAJ API uses the SOAPMessage class for the SOAP message, the SOAPPart class for the SOAP part and the SOAPEnvelope interface for the SOAP envelope. When a new SOAPMessage object is created, it has a SOAPPart object that contains a SOAPEnvelope object. The SOAPEnvelope object in turn automatically contains an empty SOAPHeader object followed by an empty SOAPBody object. The SOAPHeader object can include one or more headers that contain metadata about the message, such as information about the sending and receiving parties. The SOAPBody object contains the message content. Figure 1.14 shows a SOAP message without attachments.

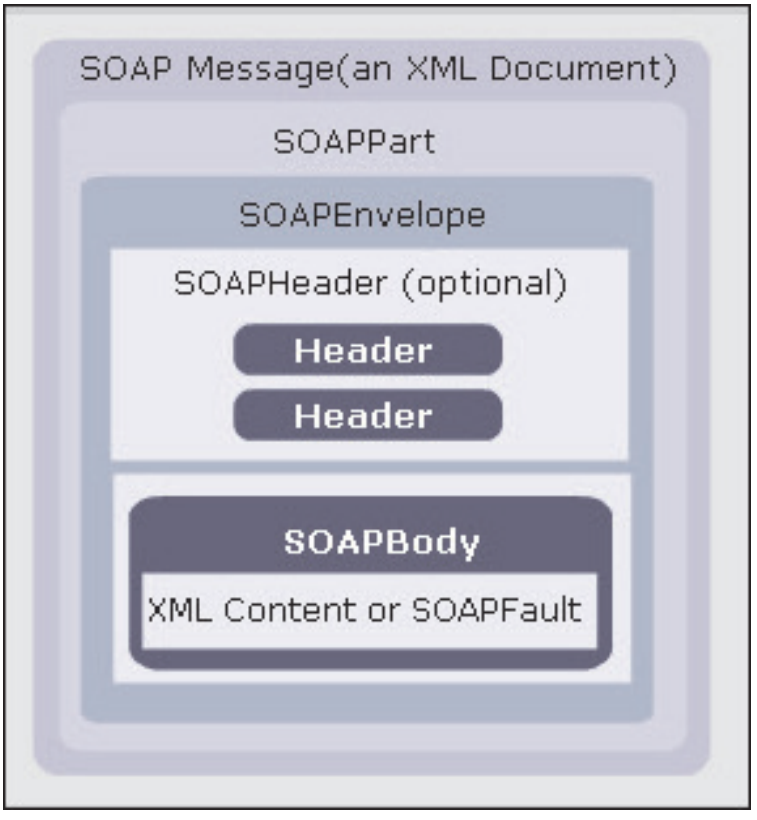


Figure 1.14: SOAP Message without Attachment

- **SOAP Message with Attachments**

A SOAP message can have one or more attachment parts along with the SOAP part. The SOAP part must contain only XML content. The SAAJ API provides the AttachmentPart class to represent an attachment part of a SOAP message. A SOAPMessage object automatically contains a SOAPPart object and its required sub elements. The AttachmentPart objects are optional, therefore, these objects need to be created and added manually.

If a SOAPMessage object has one or more attachments, the MIME header of each AttachmentPart object indicates the type of data contained in it. It may also have additional MIME headers to identify it or to give its location. When a SOAPMessage object has one or more AttachmentPart objects, its SOAPPart object may or may not contain message content. Figure 1.15 shows a soap message with attachment.

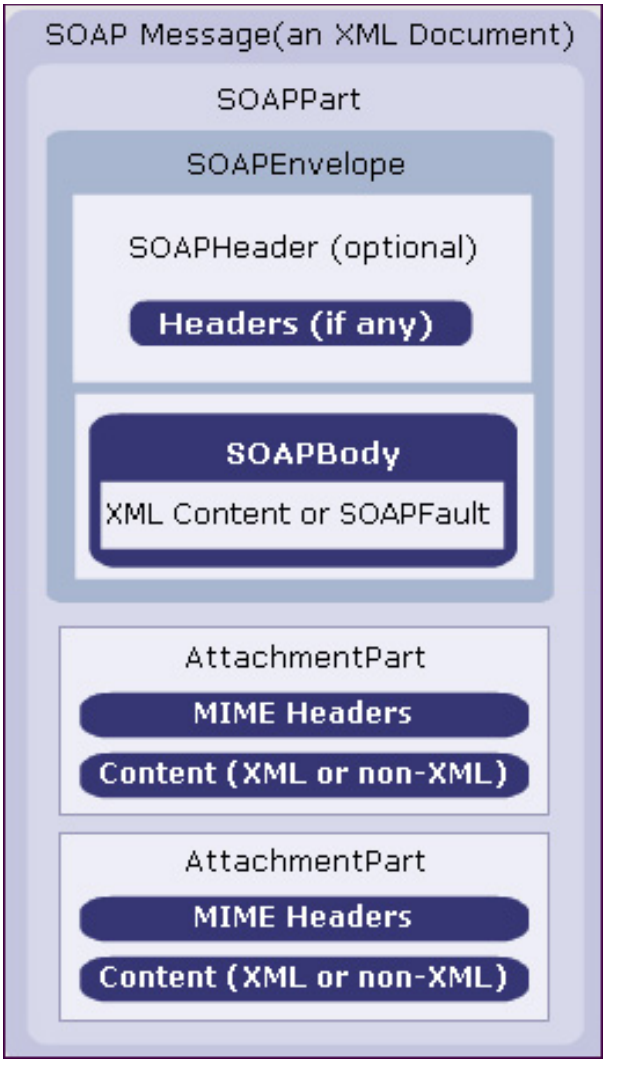


Figure 1.15: SOAP Message with Attachment

1.10.2 SAAJ API

The package javax.xml.soap provides the API for creating and building SOAP messages. Table 1.10 describes some of the classes that constitute SAAJ API.

Class	Description
AttachmentPart	A single attachment to a SOAPMessage object.
MimeHeader	An object that stores a MIME header name and its value.
SOAPConnection	A point-to-point connection that a client can use for sending messages directly to a remote party.
SOAPConnectionFactory	A factory for creating SOAPConnection objects.
SOAPMessage	The root class for all SOAP messages.

Table 1.10: SAAJ API Classes

1.10.3 SOAP Connection

In SAAJ API, the connection is represented by a SOAPConnection object, which goes from the one endpoint to another endpoint. Hence, this type of connection is called a point-to-point connection. The messages that are sent using the SAAJ API are called request-response messages. They are sent over a SOAPConnection object using the call() method by sending a request message and then blocking it, until it receives the response reply. The request parameter represents the message being sent and the endpoint represents the location to which it is sent.

The response received by the Web service is a SOAPMessage object. The response is an acknowledgment that the update was received successfully.

Code Snippet 5 creates the SOAPConnection object con and then uses it to send the message. All the messages sent over a SOAPConnection object are sent with the call() method, which sends the message and blocks until it receives the response. Thus, the return value for the call() method is the SOAPMessage object that is the response to the message that was sent.

Code Snippet 5:

```
SOAPConnectionFactory fact = SOAPConnectionFactory.newInstance();
SOAPConnection con = fact.createConnection();

// create a request message and give it content
java.net.URL endpoint = new
URL("http://java.sun.com/javaee/7/docs");
SOAPMessage response = con.call(request, endpoint);
```

SAAJ 1.3 provides some new features. They include support for the WS-I Basic Profile 1.1, Document Object Model integration where the SAAJ APIs now extend Document Object Model API. Further, SOAPMessage properties are used for setting character set encoding and turning on the writing of an XML declaration at the start of the SOAP part of the message.

1.11 JAXB

The Java Architecture for XML Binding (JAXB) is a set of interfaces which helps client applications to communicate with code generated from a schema. JAXB allows easy access to XML documents from applications written in Java programming language.

It binds the XML schemas and Java representations very fast and makes work easy for Java developers. It enables them to incorporate XML data and process functions in Java applications. It converts XML instance documents into Java content trees and vice versa.

JAXB comprises three components, which are described in table 1.11.

Interface	Description
Binding compiler	It creates Java classes from a given schema.
Binding framework	It provides runtime services such as marshalling, unmarshalling, and validation that have to be performed on the contents classes.
Binding language	It describes schema binding of Java classes using which you can override the default binding rules.

Table 1.11: Components of JAXB

1.11.1 JAXB

There is a standard structure to be followed by any two parties who communicate by passing XML documents between them. This standard is defined by the standard schema facility for XML documents. This standardization enables the communicating parties to understand the contents of the documents easily.

JAXB has a good quality XML data-binding facility for the J2EE platform. Figure 1.16 shows the JAXB architecture.

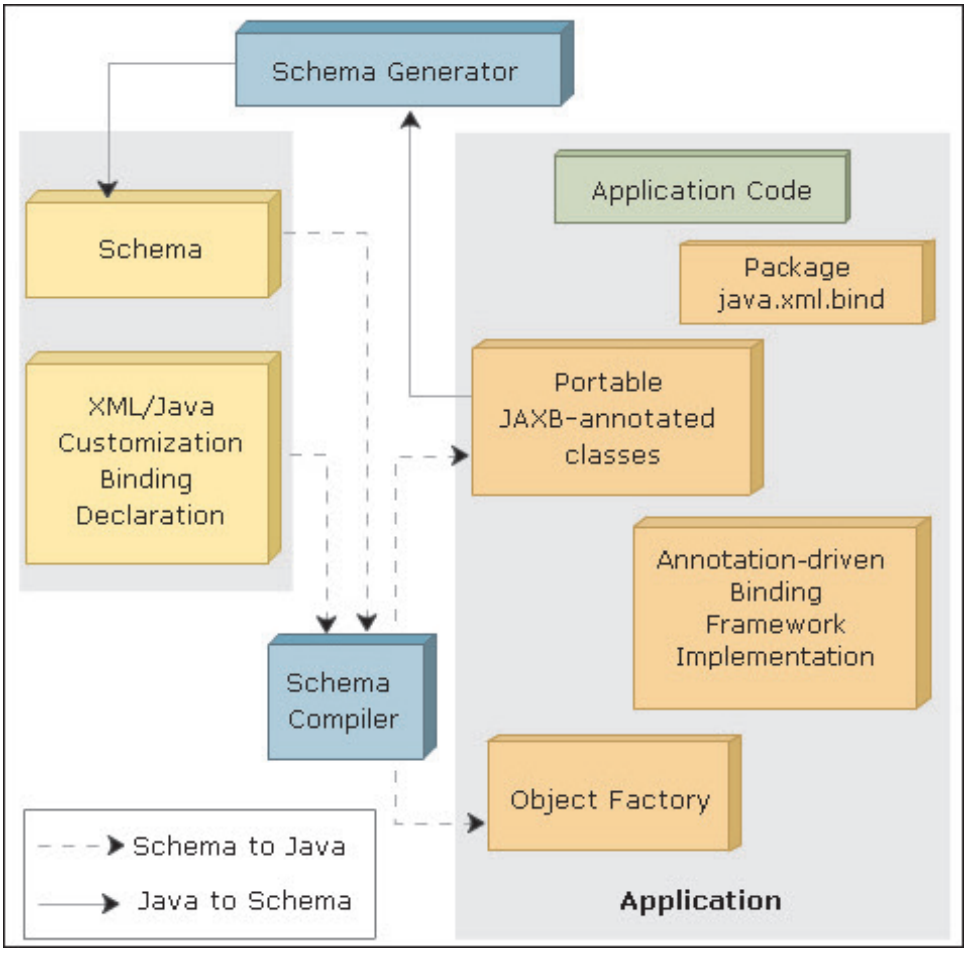


Figure 1.16: JAXB Architecture

A JAXB implementation consists of the following architectural components:

- **Schema compiler**

  Schema compiler binds a source schema to a set of schema derived program elements using an XML-based binding language.

- **Schema generator**

  Schema generator maps a set of existing program elements to a derived schema.

- **Binding runtime framework**

  Binding runtime framework accesses, manipulates, and validates XML content using the unmarshalling (reading) and marshalling (writing) operations.

1.11.2 JAXB Binding Process

JAXB presents an XML document in a Java format to the application to facilitate easy access to the contents. The schema for the XML document is bound into a set of Java classes that represents the schema. Figure 1.17 shows the JAXB binding process.

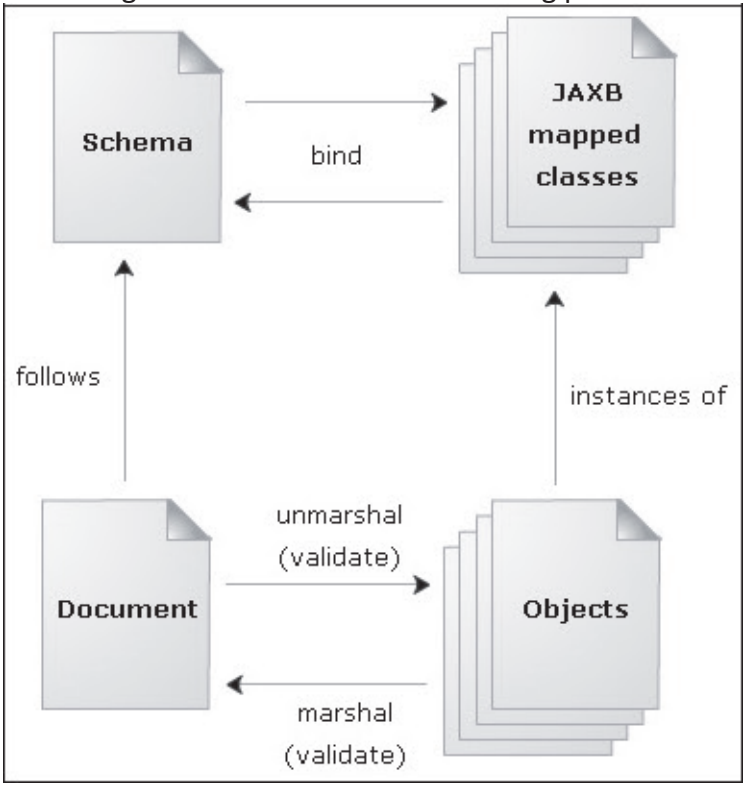


Figure 1.17: JAXB Binding Process

The steps in the JAXB data binding process are as follows:

1. **Generate classes**: As the first step, the XML schema is compiled by JAXB binding compiler to generate JAXB classes.
2. **Compile classes**: In the second step, all the generated classes, source files, and application code are compiled.

3. **Unmarshal**: In the third step, the XML documents are unmarshalled with the help of JAXB binding framework.

4. **Generate content tree**: In the fourth step, a content tree of data objects is generated by the unmarshalling process. This tree represents the structure and content of the source XML documents.

5. **Validate**: In the fifth step, the source XML documents are validated optionally by the unmarshalling process before generating the content tree.

6. **Process content**: In the sixth step, the XML data is modified by the client application. It is represented by the Java content tree.

7. **Marshal**: In the final step, the processed content tree is marshalled into XML documents. These may be further validated before marshalling.

Consider an XML document named employees.xml that stores information about employees, such as id, firstName, lastName, location, and gender as shown in figure 1.18.

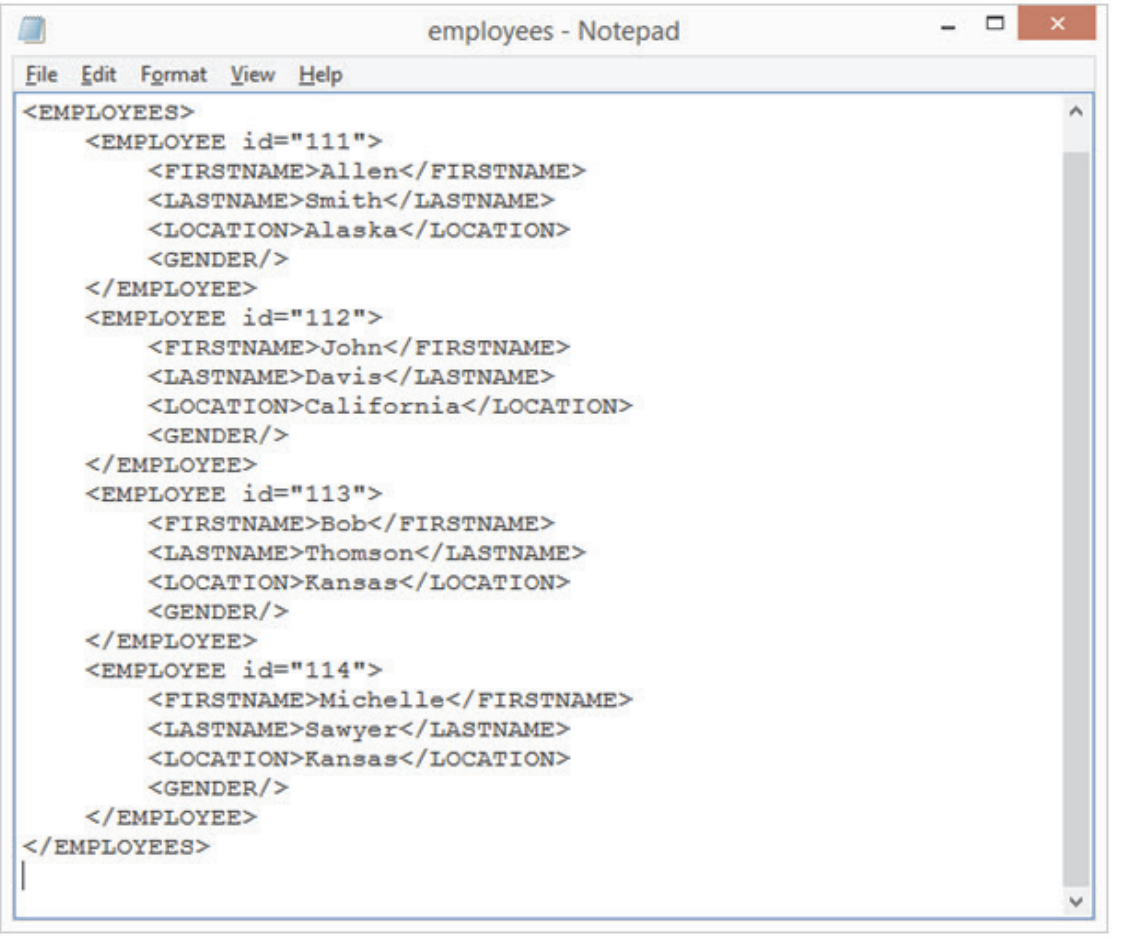


Figure 1.18: XML Document

An XML schema document that represents this XML document is shown in figure 1.19.

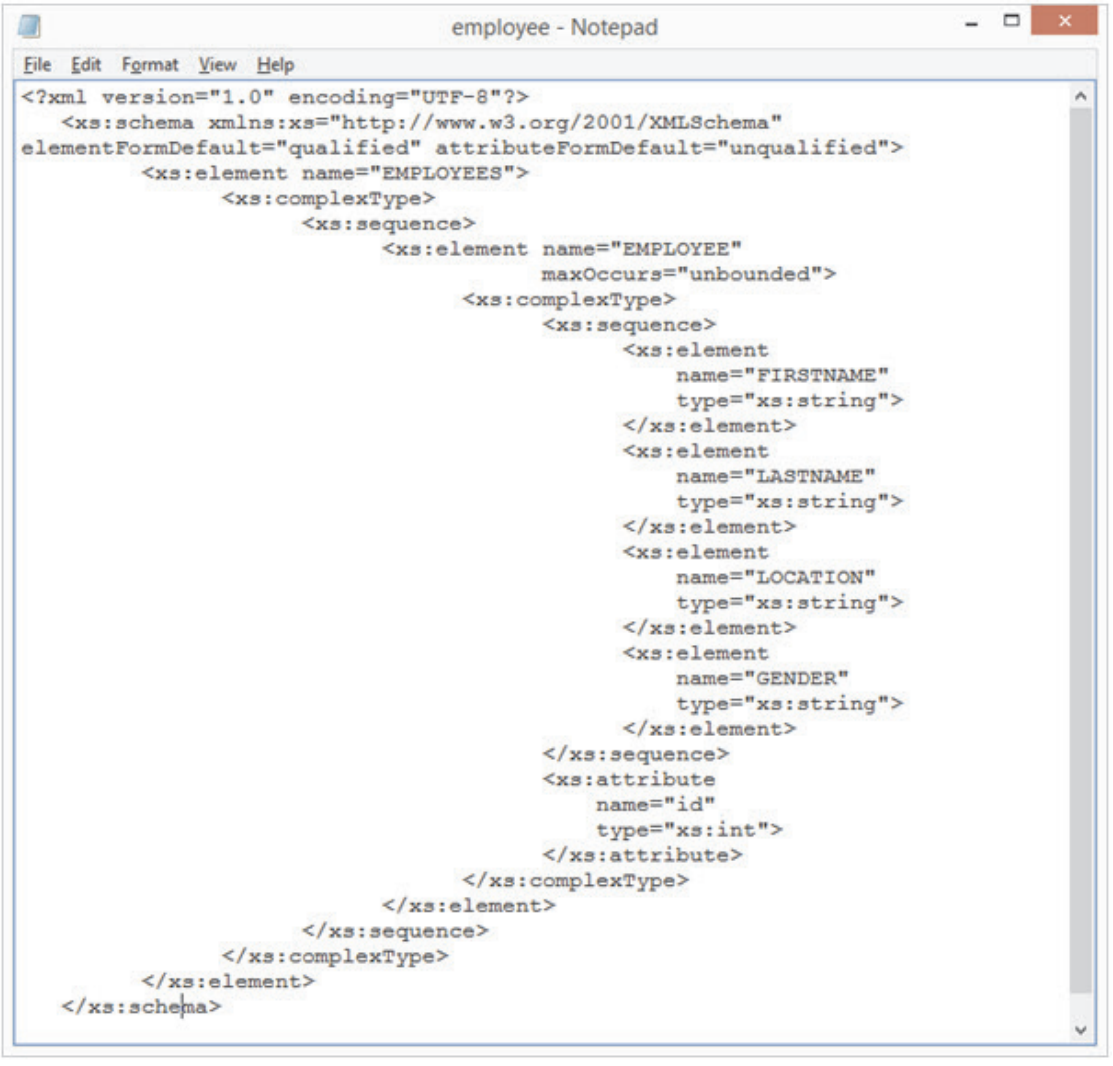


Figure 1.19: XSD Document

1.11.3 Concept of Unmarshalling

The process of converting an XML document into a content tree is termed as unmarshalling. This process includes creating content object tree that shows the content and organization of the document.

First a JAXBContext object is created. This is the entry point to the JAXB API. Then, a context path is created. This path lists the name(s) of the package(s) that contain interfaces generated by the binding compiler. Having multiple package names in the path implies that a combination of XML data elements that correspond to different schemas can be unmarshalled using JAXB. Figure 1.20 depicts unmarshalling.

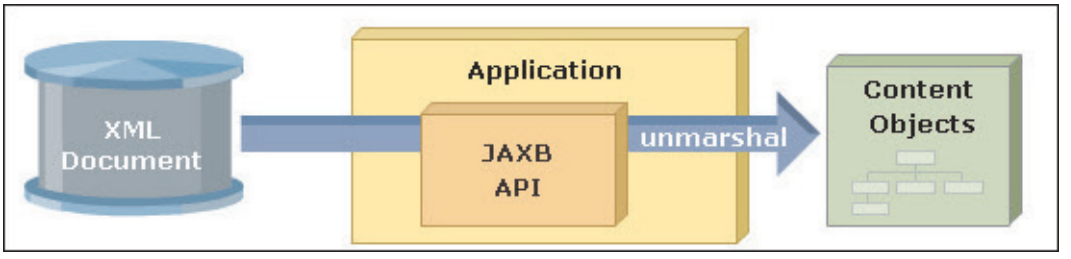


Figure 1.20: Unmarshalling

JAXB also allows accessing XML data without unmarshalling it. In that process, use the createCollection and createBookType methods to create a content objects tree. The program accesses the ObjectFactory class and uses the appropriate methods within it to create the required objects.

1.11.4 Unmarshalling Process

During unmarshalling, a DOM tree that represents the content of XML document is created. The content in the tree is in the form of nodes. Code Snippet 6 demonstrates how to parse an XML document using a DOM-based parser.

Code Snippet 6:

```
//Step1
import javax.xml.bind.JAXBContext;
JAXBContext jc = JAXBContext.newInstance(
          "Information.jaxb");
//Step2
import javax.xml.bind.Unmarshaller;
Unmarshaller unmarshaller = jc.createUnmarshaller();

//Step3
Collection collection = (collection)unmarshaller.unmarshal(new
          File("products.xml"));

//Step4
CollectionType.ProductsType productsType = collection.
getProducts();
List productList = productsType.getProduct();
```

This code performs the following tasks:

Step 1: An object of JAXBContext class is created whose context path is Information.jaxb.

The Information.jaxb is a package that contains the interfaces generated for the products.xsd schema.

Step 2: An object of the Unmarshaller class is created that controls the process of unmarshalling. In particular, it contains methods that perform the actual unmarshalling operation.

Step 3: The unmarshal() method is invoked, which performs the actual unmarshalling of the XML document, products.xml.

Step 4: The get() method in the schema-derived classes is used to access the XML data. Here, the code retrieves the types of products and stores them in a CollectionType object named productsType. It also retrieves a list of products and stores this list in a List object named productList.

1.11.5 Concept of Marshalling

Unlike the unmarshalling process, the user has to build an XML document using a content tree in the marshalling process.

For marshalling, the existing content tree is first modified or a new tree from the business logic output is created. Then, the content tree is validated against the source schema in memory. Finally, content tree is marshalled into an XML document. Data can be marshalled to other output formats such as an OutputStream object, a DOM node, or to a transformed data format such as javax.xml.transform.SAXResult. Marshalling can also be done to a content handler to process the data as SAX events.

In marshalling, an XML document is built using the DOM tree. Code Snippet 7 demonstrates how to create an XML document from a DOM tree.

Code Snippet 7:

```
//Step1
import javax.xml.bind.JAXBContext;
JAXBContext jc = JAXBContext.newInstance("Information.jaxb");
//Step2
import javax.xml.bind.Marshaller;
Marshaller marsh = jc.createMarshaller();
//Step3
marsh.setProperty(Marshaller.JAXB_FORMATED_OUTPUT, new
Boolean(true));
//Step4
marsh.marshal(collection, new Fileoutputstream("Output.xml"));
```

This code performs the following tasks:

Step 1: An object of the JAXBContext class is created and the appropriate context path of the package that contains the classes and interfaces for the bound schema is specified.

Step 2: An object of the Marshaller class is created which controls the process of marshalling.

Step 3: The Marshaller object sets its properties using the setProperty method. Here, the Marshaller.JAXB_FORMATED_OUTPUT property is set to true so that the output XML file will be formatted with proper line breaks and indentation.

Step 4: The marshal() method is invoked by specifying an object that contains the root of the content tree and the output target. Here, the code marshals the content tree whose root is in the collection object and writes it as an output stream to the XML file Output.xml.

1.11.6 Data Validation

Validation is the process of verifying if an XML document meets all the constraints expressed in the schema. It has to be done stringently while writing out data, but not when reading data. During unmarshalling, setValidating() method has to be used to validate source data against the associated schema. If an error is encountered, the data processing need not be stopped. Instead, a validation error report should be generated. JAXB specification authorizes all providers to report validation errors. If an XML document is found invalid, it will still be unmarshalled although the result will not be valid.

Validation is not done as part of marshalling process as setValidating() method does not exist for marshalling. Validation can be done at one time and marshalling at another time.

1.11.7 Limitations of JAXB API

JAXB has a few limitations. They are:

- JAXB requires a DTD and a subset of XML Schemas. Hence, it cannot be used to process generic XML, such as writing an XML editor or other tool.
- Additional work is required to tell JAXB what kind of tree it should construct to simplify the application.
- JAXB does not support the legal DTD constructs such as Internal subsets, NOTATIONs, ENTITY and ENTITIES, and Enumerated NOTATION types.

Check Your Progress

1. What is used to create WSDL file and mapping file while developing JAX-WS-based endpoint?

(A)	wscompile tool	(C)	Extensible Stylesheet Language Transformation (XSLT) engine
(B)	Graphical User Interface (GUI)	(D)	XMLReader implementation class

2. Which class is present in the java.io package?

(A)	SAXParserFactory	(C)	XMLReader
(B)	DefaultHandler	(D)	XMLStreamWriter

3. What does SAXParserFactory do?

(A)	It returns SAXParser and the exception classes.	(C)	It reads an XML document using callbacks.
(B)	It defines the API that wraps an XMLReader implementation class.	(D)	It acts as a default base class for SAX2 event handlers.

4. Which one of the following is not a benefit of JAX-WS?

(A)	Simplification of developers' responsibility	(C)	Standardization of parameter marshalling and unmarshalling
(B)	Support to inconsistencies of SOAP requests and responses	(D)	Support to different mapping scenarios

5. Which instance is used to retrieve service information from the registry while obtaining service object?

(A)	FindQualifier	(C)	BusinessLifeCycleManager
(B)	RegistryService	(D)	BusinessQueryManager

Answers

1.	A
2.	C
3.	A
4.	B
5.	D

Summary

- Java EE Web services are platform-independent and language-independent services available on the Web.
- Service Oriented Architecture (SOA) aids application development by a loosely-coupled modular approach. It permits the applications to extend and utilize services.
- SOAP is a Web service standard for packaging the XML data that are transferred between applications.
- The Java EE platform has two types of endpoints for Web services. One extends Stateless Session Beans into Web services and the other resembles a plain Java class.
- The Java API (JAXP) provides a framework for using SAX2 and DOM2 that read, write, and modify XML documents.
- JAX-WS provides a framework and runtime environment to create and execute XML-based Web services.
- The Java API for XML Registries (JAXR) API provides a single set of APIs to access a variety of XML registries.
- The Java Architecture for XML Binding (JAXB) API is a set of interfaces that enable communication through the code generated from a schema.

Tutor Chat

Onlinevarsity
your e-way to learning

are just a click away

To chat with a

login to **www.onlinevarsity.com**

02

SOAP, WSDL, and UDDI

Welcome to the Session, **SOAP, WSDL, and UDDI**.

This session explains the objective of using Simple Object Access Protocol (SOAP) in a Web Service and lists the advantages of using SOAP. It describes each part of a SOAP message. The session also explains the transportation of SOAP over HTTP protocol and describes SOAP fault codes. Further, it explains the purpose of a Web Service Description Language (WSDL) file and its elements. Finally, it explains the Universal Description, Discovery, and Integration (UDDI) model and ebXML Registry standards.

In this Session, you will learn to:

- Explain the purpose and advantages of using SOAP in Web Service
- Describe the different parts of a SOAP message
- Explain SOAP messaging with attachment
- Define Document/Literal SOAP and RPC/Literal SOAP messaging modes
- Explain transportation of SOAP over HTTP protocol
- List sub-elements of SOAP fault element and describe SOAP fault codes
- Explain the purpose of a WSDL file and describe its elements
- Describe UDDI model
- Explain the ebXML Registry standards

Session

02

SOAP, WSDL, and UDDI

2.1 Introduction to SOAP

SOAP stands for Simple Object Access Protocol. It is a communication protocol for communication between applications. It is based on XML and a W3C recommendation.

2.1.1 Information Exchange Approaches

Electronic Data Interchange (EDI) and Remote Procedure Call (RPC) are the common approaches used for information exchange between distributed systems.

- **EDI**

  EDI is a technique used by business partners to exchange business documents. The business documents include purchase orders, invoices, shipping notifications, financial payments, and so on.

  To send an EDI document, perform the following steps:

  1. Install translation software on the system. The translation software converts business documents into X12 format.
  2. Set up a private wide area network to send and receive the documents.

  The same process is repeated at the receiver's end.

  The EDI technique had the following drawbacks:

  - Cost involved in setting up private wide area networks was too high.
  - Business partners had to buy proprietary software for transmission of messages from their system to private network.
  - Each business partner had to buy propriety software to translate business documents to X12 format.

- **RPC and RMI**

  The Remote Procedure Call (RPC) approach involved invoking remote methods. These remote methods allowed information exchange in the form of parameters and returned values.

  The same concept of RPC is used in Remote Method Invocation (RMI). The RMI approach standardized the communication protocol and eliminated the need of private networks. The drawback of RPC and RMI approach are as follows:

  - Several vendors came up with RPC-based technologies, such as Common Object Request Broker Architecture (CORBA), RMI, and Distributed Component Object Model (DCOM). Each vendor provided its own protocol to communicate within the distributed systems. This allowed a corporate using CORBA to communicate only with a business partner using CORBA, RMI with RMI, and DCOM with DCOM.

- Communication between distributed systems required relaxation of security features.

2.1.2 SOAP

SOAP provides interoperability between applications using XML and HTTP. XML is platform and language independent. Hence, a Java program on a Linux platform could easily interpret a SOAP message created by a C# program on a Windows platform.

SOAP version 1.2 is a lightweight protocol that helps in exchange of information in a decentralized, distributed network. It uses XML technologies to define an extensible messaging framework. From version 1.2, SOAP is no longer an acronym 'Simple Object Access Protocol' as the original expansion is to some extent misleading.

SOAP 1.2 is based on XML Information Set. In this version of SOAP, a message is sent as an Infoset from one SOAP node to another. SOAP 1.2 allows the Infoset to be transported from one node to another using any protocol, such as HTTP and an XML 1.0 serialization. To support this protocol-independent feature, SOAP 1.2 defines a binding framework that specifies the responsibilities of the underlying transport mechanism for carrying SOAP messages from one node to another.

Figure 2.1 shows the use of SOAP.

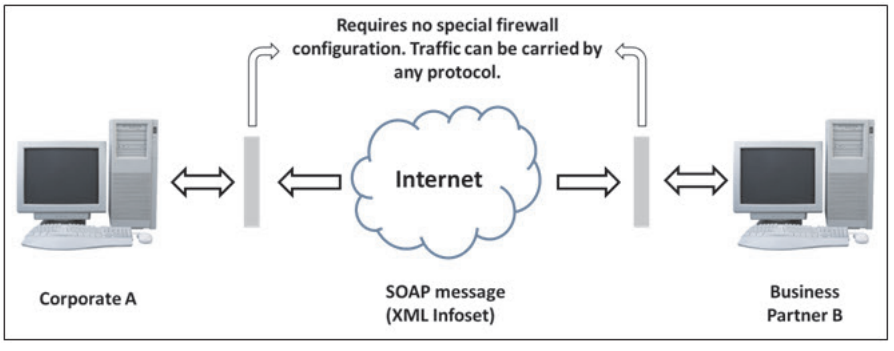


Figure 2.1: Use of SOAP

2.1.3 Advantages of SOAP

The advantages of SOAP are as follows:

- **Vendor Neutral** — SOAP is defined by World Wide Web Consortium (W3C), a non-profit organization that defines standards for the Web. Most vendors implement products as per the specifications or standards defined by W3C.
- **Transport Protocol Independent** — SOAP messages can be transmitted over HTTP, Simple Mail Transfer Protocol (SMTP), File Transfer Protocol (FTP), and Post Office Protocol (POP). However, Web Service Interoperability (WS-I) organization allows the use of only HTTP.

- **Platform Independent** — XML is platform independent. SOAP messages are constructed using XML Infoset, hence, SOAP is also platform independent.
- **Language Independent** — SOAP messages are XML Infosets. Several languages, such as Java, Perl, and C++ can be used to create these messages.
- **Interoperability** — SOAP transports XML Infosets using various protocols. This enables distributed applications to communicate without any issues.
- **Simple** — SOAP does not require any change in network infrastructure or security configurations and hence, is simple to use.

2.1.4 SOAP 1.1 vs. SOAP 1.2

Table 2.1 provides a comparison between the features provided by SOAP 1.1 and SOAP 1.2.

Feature	SOAP 1.1	SOAP 1.2
SOAP messages	Is based on XML 1.0	Is based on XML Infosets
SOAP elements	Allows additional elements to be used after the <body> element	Does not allow additional elements to be used after the <body> element
actor attribute	Indicates the receiver of a header element	Renamed to role and indicates the receiver of a header element
next role	Can be applied to the intermediary SOAP nodes	Can be applied to the intermediary SOAP nodes and the ultimate receiver
none role	Not supported	Used to indicate that the header block should not be processed by any of the SOAP nodes
ultimateReceiver role	Not supported	Used to indicate a header block must be processed only by the ultimate receiver of the SOAP message
Fault codes	Supports the VersionMismatch, MustUnderstand, Client, and Server fault codes	Supports the DataEncodingUnknown, VersionMismatch, MustUnderstand, Sender (previously, Client), and Receiver (previously, Server) fault codes
Fault code extensions	Allows a dot notation to be used to indicate fault code extensions	Provides an XML-like structure for the fault code extensions

Feature	SOAP 1.1	SOAP 1.2
Fault structure	Only the root element was namespace qualified (e:fault); the other elements are faultcode, faultstring, faultfactor, and detail	All the elements are namespace qualified with e:fault as the root element and e:Code, e:Subcode, e:Value, e:Reason, e:Node, e:Role, and e:detail as the child elements
Fault semantics	Allows the body of the fault message to include multiple child elements	Allows the body of the fault message to include only one child element
Misunderstood header	Not supported	Used for providing additional information in the MustUnderstand faults
Upgrade header	Not supported	Used to specify the supported envelope versions when the VersionMismatch fault is reported by a node
Binding Framework	Supports single binding to HTTP	Provides an abstract binding framework to support all protocols
encodingStyle attribute	Allows the attribute to be used on any element in the envelope	Allows the attribute to be used only on the child elements of the Body, Header, and fault Detail elements
Multi-reference values	Encodes multi-reference values in top-level elements	Allows in-place encoding of multi-reference values

Table 2.1: SOAP 1.1 vs. SOAP 1.2

2.2 SOAP Message Structure

A SOAP message is used to exchange data between applications.

2.2.1 SOAP Message

A SOAP message is an XML document. Examples of SOAP messages are as follows:

- Request to receive stock update
- Request to get a book price
- Query to a search engine
- Purchase order
- List of available flights

A SOAP message contains the following parts:

- XML declaration
- Envelope: Comprises Header and Body. The data is enclosed in the Body part of the SOAP message. For example, a request to receive stock update will be part of the Envelope.

Figure 2.2 shows the SOAP message structure.

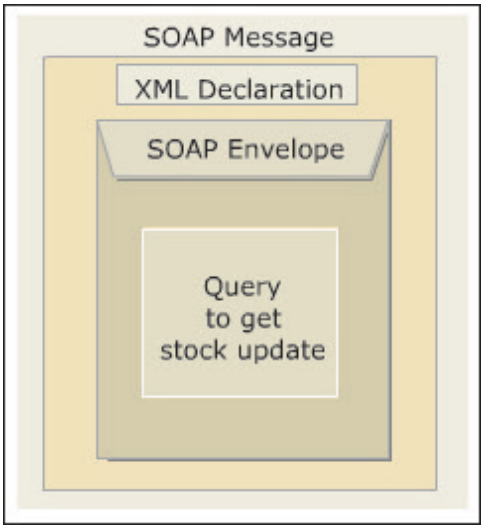


Figure 2.2: SOAP Message Structure

2.2.2 XML Declaration

A SOAP message begins with an XML declaration. The declaration states the version of XML and encoding used. The version of XML must be 1.0. The encoding value can be either Unicode Transformation Format (UTF)-8 or UTF-16. The absence of encoding attribute indicates that the document uses UTF-8 and the version of the document is 1.0.

Note - It is not mandatory to have an XML declaration in a SOAP message.

Code Snippet 1 demonstrates the XML declaration tag of a SOAP message.

Code Snippet 1:

```
<? Xml version= "1.0" encoding = "UTF-8"?>
```

Note - Unicode is an encoding standard that can represent the characters, digits, and symbols of world's major languages. UTF-8 is a Unicode character-encoding format created to provide backward compatibility with ASCII. UTF-8 and UTF-16 encoding formats can represent any character in Unicode character set.

2.2.3 SOAP Message Envelope

The SOAP envelope contains a message from one application to be sent to another application. The envelope part of the SOAP message acts as a container for the message. The Envelope element:

- is the root element of the message and is mandatory
- acts as a processing node to the receiving application. For example, the <Envelope> tag indicates the start of a SOAP message and the </Envelope> tag indicates the end. Once the receiving application encounters the </Envelope> tag, it starts processing the message
- contains two child elements, an optional <Header> element and a mandatory <Body> element
- SOAP 1.1 allows additional elements to be added after the SOAP Body element, SOAP 1.2 does not allow these

Code Snippet 2 demonstrates the Envelope tag in a SOAP message.

Code Snippet 2:

```
<Envelope ...>
  <Header>
    0 or more headers
  </Header>
  <Body>
    message body
  </Body>
</Envelope>
```

- **SOAP Namespaces**

  A SOAP Envelope contains several XML elements and it is possible to use the same name for one or more elements. SOAP prevents name collisions using XML namespaces.

  SOAP uses the namespace http://www.w3.org/2003/05/soap- envelope/ to qualify the XML elements and attributes in a SOAP message. Code Snippet 3 demonstrates a SOAP message with its elements qualified with respective namespaces.

Code Snippet 3:

```
<SOAP-ENV:Envelope
xmlns:SOAP-ENV= "http://www.w3.org/2003/05/soap-envelope"
xmlns:po= "http://www.flamingo.com/books/PO">
   ...
<!--SOAP Message goes here -->
       <po:orderDate> 2014/07/06 </po:orderDate>
   ...
</SOAP-ENV:Envelope>
```

In the given Code Snippet, the Envelope element is qualified by http://www.w3.org/2003/05/soap-envelope/ namespace. The prefix of this namespace is SOAP-ENV.

The namespace http://www.flamingo.com/books/PO is used to qualify the data elements of the SOAP message. The prefix for this namespace is po. The orderDate element is qualified with this namespace.

The namespace http://www.w3.org/2003/05/soap-envelope/ indicates that the message adheres to SOAP 1.2. This ensures that both the sender and the receiver adhere to same version of SOAP.

- **SOAP Message Header**

Header is the child element of Envelope element. It is an optional element. However, if present it should be the immediate child of Envelope element. The immediate child elements of the Header element are referred as header entries. Each header entry and its child elements must be qualified with a namespace. The header entries can have the attributes role and mustUnderstand. These attributes identify the consumer of the header entry and how to process the header entry.

Headers can be used to extend SOAP messages to include additional information and functionality to process a message. For example, a typical transfer payment service requires a from account number, to account number, and the amount to be transferred. However, to process such services, you need additional information, such as identity of the person requesting the service, account information, and so on. Such information is included in the header part of the SOAP message.

Headers can also include information such as digital signatures for password-protected service, authentication, authorization, transaction management, and routing path.

- **role Attribute**

A SOAP message travels from the original source to the ultimate destination. In the process, it passes through several intermediate entities referred as nodes or SOAP intermediaries. These intermediaries are applications capable of receiving SOAP message and forwarding them. The role attribute is used to specify the URI or role of such intermediaries and the ultimate destination of a message.

SOAP 1.2 defines the following three roles that have significant importance in a SOAP message:

- **next:** This role specifies that each SOAP nodes involved in transporting the message and the final SOAP receiver must process the message.
- **none:** This role specifies that each SOAP node involved in transporting the message must not process the message.
- **utimateReceiver:** This role specifies that the final SOAP receiver must process the message.

Code Snippet 4 demonstrates a SOAP message with various header entries.

Code Snippet 4:

```
<SOAP-ENV: Envelope
xmlns:SOAP-ENV= "http://www.w3.org/2003/05/soap-envelope"
xmlns:mid= "http://www.flamingo.com/books/message-id"
xmlns:m= "http://www.flamingo.com/books/monitored-by"
xmlns:md = "http://www.flamingo.com/books/monitoring-date"
xmlns:mu = "http://www.flamingo.com/books/mark"
      <SOAP-ENV: Header>
<mid:message-id
SOAP-ENV:role="http://www.flamingo.com/logger">
             11546544ea:b134534:f3sdas5342:4354
</mid:message-id>
           <m:monitored-by SOAP-ENV:role = "
http://www.w3.org/2003/05/soap-
envelope/role/next">
<node>
           <time> 1078753670000  </time>
```

```
                    <identity>austria</identity>
            </node>
            </m:monitored-by>
            <md:monitoring-date SOAP-ENV:role = "
http://www.w3.org/2003/05/soap-
envelope/role/none">
...
            </md:monitoring-date>
            <mu:mark SOAP-ENV:role =
"http://www.w3.org/2003/05/soap-
envelope/role/ultimateReceiver">
                    ...
            </mu:mark>
      </SOAP-ENV: Header>
</SOAP-ENV: Envelope>
```

In the given Code Snippet, the header element message-id has the role attribute set to http://www.flamingo.com/logger. Therefore, all the nodes identifying themselves by this URI are the recipients of this header entry.

The header element monitored-by uses the role attribute to refer to the role http://www.w3.org/2003/05/soap-envelope/next. This special URI indicates that this header entry should be processed by the very first node that receives the message.

The header element monitoring-date uses the role attribute none to refer to the role http://www.w3.org/2003/05/soap-envelope/role/none. This new SOAP 1.2 URI indicates that this header entry should not be directly processed by any of the SOAP nodes.

The header element mark uses the role attribute to refer to the ultimateReceiver http://www.w3.org/2003/05/soap-envelope/role/none. This new SOAP 1.2URI indicates that this header entry should be processed only by the final receiver of the message.

- **mustUnderstand Attribute**

  A header attribute may contain another attribute, mustUnderstand. This attribute is used to indicate to the recipient whether it is mandatory to process the header entry. This attribute allows two values false and true. The true value indicates that the recipient should process the header entry. A false value is equivalent to omitting the mustUnderstand attribute.

Code Snippet 5 demonstrates an example of a header entry with the mustUnderstand attribute.

Code Snippet 5:

```
<!-- SOAP Message Structure  -->
<?xml version ="1.0" encoding="UTF-8" ?><SOAP-ENV:Envelope
    xmlns:SOAP-ENV= "http://www.w3.org/2003/05/soap-
envelope">

  <SOAP-ENV:Header
xmlns:au="http://www.flamingo.com/books.authentication">
<au:Requestor mustUnderstand="true"
                 <name> John Smith </name>
       </au:Requestor>
               ...
     </SOAP-ENV:Header>
               ...
</SOAP-ENV:Envelope>
```

In the given Code Snippet, the mustUnderstand attribute is set to true indicating that the receiver of the message should process the header entry.

- **Body Element**

The Body element contains application-specific data to be exchanged between applications. The data in the Body element is an XML fragment containing information, such as billing address or parameters to a method call.

The Body element is the mandatory element of Envelope element. The immediate child elements of Body element must be namespace-qualified. The Body element must be placed as follows:

- If the Header element is not present, the Body element should be the immediate child of Envelope element.
- If the Header element is present, the Body element should immediately follow the Header element.

- **Need of Attachment**

SOAP messages contain XML fragment. However, at times you may want to send data that is not XML. For example, consider a scenario wherein you want to allow customers to book a room in a hotel.

This can easily be achieved through SOAP-based Web Service wherein a user sends the number of rooms to book and the service responds by sending the availability of the rooms. All this can be achieved through SOAP as the data exchanged is text.

Consider another scenario wherein the hotel management decides to upgrade the service. They would like the customers to have a glance of the interiors of the rooms before finalizing a booking. For this, the Web Service should be able to send images back to the customer. However, SOAP does not allow binary data such as images in the message.

Messages that require binary data are converted to a Multipurpose Internet Mail Extensions (MIME) message format and then sent. A MIME message can contain multiple parts and supports binary data as well.

2.2.4 MIME Message Structure

MIME helps to include attachment in an email. MIME message has the following parts:

- **MIME header:** Identifies the version of MIME and content type. A content type as Multipart/Mixed indicates that the email contains multiple parts of varying types
- **MIME package:** Can contain one or more MIME parts

An email with attachment contains two or more MIME parts. One part contains the text message and the other part contains the attached files. Each MIME part contains a header that identifies the type of content. The Content-type for text message is text/plain, PDF file has application/pdf and a GIF image file has image/gif. Each MIME part is separated from one another by a boundary.

Figure 2.3 shows different types of message attachments.

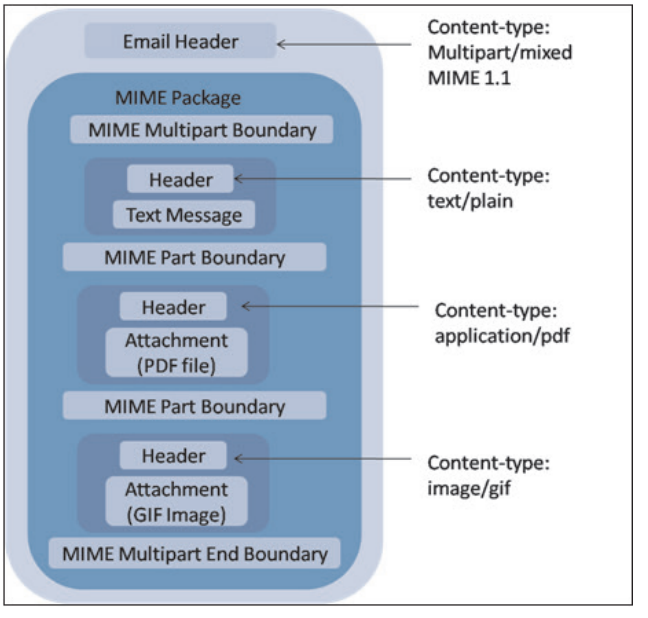


Figure 2.3: Different Types of Message Attachments

- **MIME Message with SOAP Attachment**

SOAP lacks the provision of including binary data in the message. The SOAP with Attachments (SwA) standard allows SOAP messages to be sent along with binary data. SwA puts the SOAP message along with binary data in the MIME package. Figure 2.4 shows a SOAP message with two attachments.

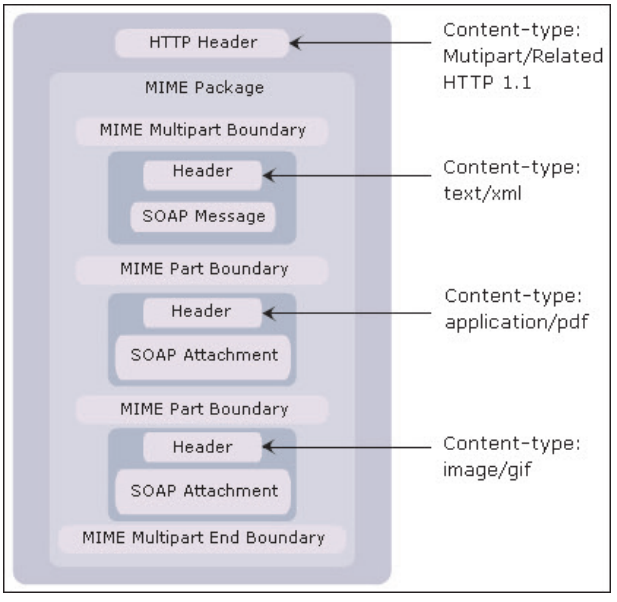


Figure 2.4: SOAP Message with Two Attachments

SwA sends these messages using HTTP. Therefore, the email header is replaced with HTTP header. The Content-type of the HTTP header is set to Multipart/Related. The Content-type of the part in which SOAP message is placed, is set to text/xml.

2.3 SOAP Messaging Modes

SOAP messaging modes are classified in terms of messaging style and encoding type. The messaging styles are RPC and Document. The encoding types are SOAP encoding and Literal encoding. Based on these messaging styles and encoding types, SOAP defines the following four messaging modes:

- Document/Literal
- RPC/Literal
- Document/Encoded
- RPC/Encoded

However, WS-I Basic Profile (WS-I BP) 2.0 does not support Document/Encoded and RPC/Encoded messaging modes.

Note - WS-I BP is a specification provided by Web Services Interoperability (WS-I) industry. This specification defines how the core Web Services specifications, such as SOAP, WSDL, and UDDI, must interoperate.

2.3.1 Document/Literal Messaging Mode

The Document/Literal messaging mode is used when you want to send XML fragments in a SOAP message. The XML fragment could be movie details, purchase order, billing address, book details, and so on. The schema and namespace of this XML fragment is different from that of SOAP elements.

Code Snippet 6 demonstrates a SOAP message with a Body element with the details about the Spiderman Digital Versatile/Video Disc (DVD).

Code Snippet 6:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://www.w3.org/2003/05/soap/envelope">
     ...
<SOAP-ENV:Body>
<d:dvd xmlns:d="http://www.flamingo.com/dvds/">
         <d:title>Spider-Man</d:title>
         <d:language>English</d:language>
         <d:discs>2</d:discs>
         <d:runtime>
              <d:minutes>121 </d:minutes>
          </d:runtime>
         <d:price>9.49</d:price>
      </d:dvd>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

2.3.2 RPC/Literal Messaging Mode

The RPC/Literal messaging mode is based on sending request messages and receiving response messages.

The request message represents a method invocation with zero or more parameters. Similarly, the response message represents a result or value returned by a method. The requested information and resultant information is sent as a part of the Body element.

Code Snippet 7 demonstrates SOAP request message.

Code Snippet 7:

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV=" http://www.w3.org/2003/05/soap/envelope"
    ...
 <SOAP-ENV:Body>
  <m:GetOrderStatus xmlns:m="http://www.flamingo.com/methods">
   <m:OrderNo>34347</m:OrderNo>
    </m:GetOrderStatus>
    </SOAP-ENV:Body>
 </SOAP-ENV:Envelope>
```

The given Code Snippet shows that in the request message, the GetOrderStatus element represents a method and the element OrderNo represents the parameter orderNo. The request message is requesting the status of the order numbered 34347.

Code Snippet 8 demonstrates SOAP response message.

Code Snippet 8:

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV=" http://www.w3.org/2003/05/soap/envelope"
    ...
 <SOAP-ENV:Body>
  <m:GetOrderStatusResponse xmlns:m="http://www.flamingo.com/
methods">
   <m:OrderStatus>Shipped on 2014-08-09 </m:OrderStatus>
    </m:GetOrderStatusResponse>
    </SOAP-ENV:Body>
 </SOAP-ENV:Envelope
```

The given Code Snippet shows that in the response message, the GetOrderStatusResponse element represents a method and the element OrderStatus represents the parameter OrderStatus. The response message provides the status of the order number 34347 as Shipped on 2014-08-09.

2.4 Transport Protocols

SOAP is an XML-based protocol. It follows the HTTP request and response model to transmit client request and obtain Web Service response.

SOAP messages over HTTP help in exchange of messages between clients and Web services, all running on different platforms and at various locations on the Internet.

2.4.1 SOAP HTTP Binding

HTTP is a standard protocol used worldwide to transfer data over the Web. HTTP tunneling feature allows HTTP to hide another protocol within the HTTP message so that the message passes through the firewall without any obstruction.

SOAP messages are transmitted as a payload of an HTTP message. A payload of an HTTP message contains form data such as username, password, credit card number, and so on. HTTP is a request-response protocol. A SOAP message is carried as a payload in an HTTP POST message. Figure 2.5 shows the HTTP POST message.

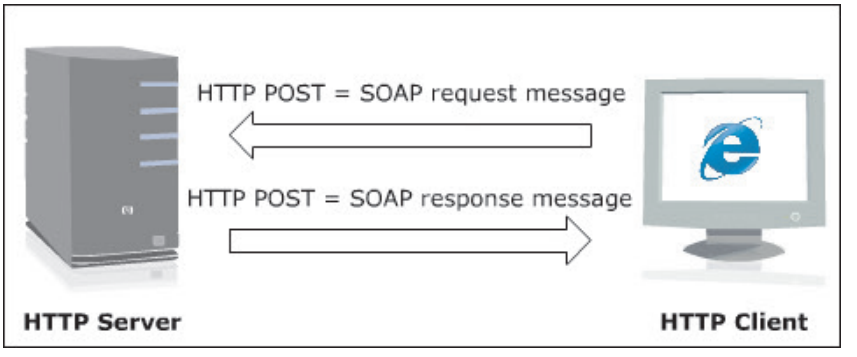


Figure 2.5: HTTP POST Message

HTTP is an application-level protocol. It is used in distributed, collaborative, and hypermedia information systems. The first version of HTTP is referred to as HTTP/0.9. It was a protocol for raw data transfer across the Internet. The next version, HTTP 1.0 permitted messages to be in the format of MIME-like messages. The MIME format message holds meta-information about the data transferred. It also had the modifiers on the request/response semantics. HTTP 1.0 cannot be used to handle hierarchical proxies, caching, and in persistent connections or virtual hosts. HTTP 1.1 version was defined to ensure reliable implementation of these features.

Some of the features of HTTP 1.1 are as follows:

- It requires a Host header. This header allows a message to be routed through proxy servers and helps the Web server to distinguish between the various sites on the same server.
- It supports persistent connections, which allows multiple request/response on the same HTTP connections.
- It provides the OPTIONS method that helps determine the capabilities of the HTTP server.
- It provides the entity tag, which expands on the caching support.
- It provides additional conditional headers such as If-Unmodified-Since, If-Match, and If-None-Match.

2.4.2 SOAP Request over HTTP

A SOAP message sent using HTTP POST message comprises the HTTP header and the SOAP message.

Code Snippet 9 demonstrates SOAP request over HTTP.

Code Snippet 9:

```
POST/orderstatus HTTP/1.1
Host:www.flamingo.com:80
Content-Type: text/xml; charset=utf-8
COntent-Length:482
SOAPAction:"http://www.flamingo.com/books/getOrderStatus"
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV=" http://www.w3.org/2003/05/soap/envelope">
    ...
    <SOAP-ENV:Body>
         <m:GetOrderStatus
           xmlns:m="http://www.flamingo,com/methods">
            <m:OrderNo>34347</m:OrderNo>
    </m:GetOrderStatus>
   </SOAP-ENV:Body>
   </SOAP-ENV:Envelope>
```

The line, POST/orderstatus HTTP/1.1, in the HTTP header indicates that it is an HTTP version 1.1 POST message targeted for the Web Service orderstatus. The second line, Host:www.flamingo.com:80 provides host location of the Web Service. The value text/ xml of Content-Type field indicate that the POST message contains XML content. The Content- Length field provides the number of characters in the entire message.

The SOAPAction field is a mandatory field for HTTP messages used for sending SOAP messages. It indicates to the HTTP server what it should do before processing the message. This field can contain arbitrary data, a URI, an empty string or can be blank. In the code, the SOAPAction field contains the namespace of the Web Service and the name of the method responsible for processing the message.

2.4.3 SOAP Response over HTTP

A SOAP message sent using HTTP also contains the HTTP header and the SOAP message. Code Snippet 10 demonstrates a SOAP response message embedded within an HTTP POST message.

Code Snippet 10:

```
HTTP/1.1 200 OK
Connection:close
Content-Length: 659
Content-Type:text/xml; charset=utf-8
<?xml version=”1.0” encoding=”UTF-8”?>
<SOAP-ENV: Envelope
    xmlns:SOAP-ENV=” http://www.w3.org/2003/05/soap/envelope”>
  ...
  <SOAP-ENV:Body>
   <m:GetOrderStatusResponse
        xmlns:m=”http://www.flamingo.com/methods”>
           <m:OrderStatus>
                Shipped on 2014-08-09
           </m:OrderStatus>
    </m:GetOrderStatusResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In the given Code Snippet, HTTP/1.1 200 OK in the header indicates that it is HTTP version 1.1 message.

The value 200 OK is an HTTP response success code indicating that message was received and processed successfully. Connection:close indicates that the connection between the HTTP client and server is closed. The Content-Length field provides the number of characters in the message. The Content-Type field indicates that the message content is XML.

In case the HTTP server fails to process the SOAP message, an HTTP response message embedded with a SOAP message is sent to the sender. This SOAP error information is included in the Fault element of Body element of the SOAP message.

2.4.4 Fault Element

A SOAP fault is generated when an error occurs in transmission of message. In SOAP, errors are generated by the nodes in the message path while processing a message. The causes for errors in SOAP are improper message formatting, version mismatch, trouble processing a header, and application specific errors.

An error is enclosed in a Fault element. The Fault element is enclosed in the Body element and the message formed is referred as a fault message. A Fault element if present in a SOAP message, must always be a child of Body element and it can appear only once in the Body element. Additionally, the Body element must contain only the Fault element and nothing else. Figure 2.6 shows SOAP message with Fault element.

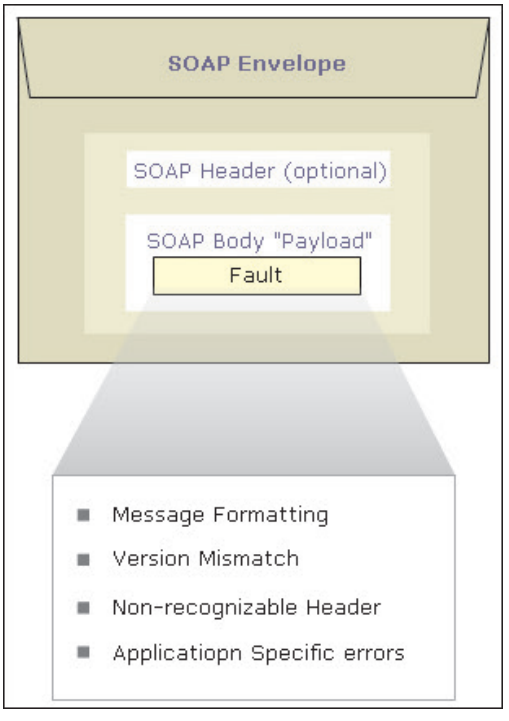


Figure 2.6: SOAP Message with Fault Element

2.4.5 Fault Subelements

The Fault element contains four subelements namely, env:Code, env:Reason, env:Detail, and env:Node. The env:Code and env:Reason are mandatory subelements, and the env:Detail and env:Node subelements are optional. Table 2.2 describes the Fault subelements.

Fault Subelement	Description
env:Code	Identifies an error in a SOAP message. This subelement contains two child elements e:Value and e:Subcode.
env:Value	Indicates a value that identifies the type of error that occurred.

Fault Subelement	Description
env:Subcode	Indicates additional error information.
env:Reason	Provides a user friendly message about an error.
env:Detail	Indicates the application-specific errors.
env:Node	Indicates the SOAP node that caused the error.

Table 2.2: Fault Subelements

2.4.6 Fault Codes

The faultcode subelement provides error information in the form of fault codes namely, VersionMismatch, MustUnderstand, Client, and Server. The fault codes must be namespace qualified in the actual message. Table 2.3 describes the fault code.

Fault Code	Description
VersionMismatch	Indicates that receiving application received a different version of SOAP message.
MustUnderstand	Indicates that receiving node or application was unable to process the header entry targeted for it. Note: If the mustUnderstand attribute in the message received was set to 1, the receiving node sends a fault message.
Sender	Indicates an error in the message or its data. That is, it indicates an error on the message sender's end.
Receiver	Indicates that the intermediary node or the ultimate receiver was unable to process the message.
DataEncoding Unknown	Indicates that the received messages are using an unrecognized value of the encodingStyle attribute.

Table 2.3: Fault Code

2.4.7 SOAP Fault over HTTP

An error in a SOAP message is communicated to the sender using a SOAP fault message. The SOAP fault message is sent using HTTP by embedding it in an HTTP response message.

Code Snippet 11 demonstrates an HTTP response message indicating mismatch in the version of SOAP message received.

Code Snippet 11:

```
 HTTP/1.1 500 Internal Server Error
Connection: close
Content-Length:659
Content-Type: text/xml; charset=utf-8
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
<SOAP-ENV:Fault>
                <Code>SOAP-ENV:VersionMismatch</Code>
                <Reason>Not a SOAP 1.2 message. </Reason>
                <Detail>Version MisMatch </Detail>
     </SOAP-ENV:Fault>
</SOAP-ENV: Body>
</SOAP-ENV:Envelope>
```

In the given Code Snippet, the response code is 500. The SOAP error information is included in a fault element. The fault element is enclosed in the Body element. The fault element includes the Code subelement that indicates an error caused by version mismatch. It then uses the Reason subelement to provide a message stating that the message is not a SOAP 1.2 message. It also uses the Detail subelement to specify the details of the error occurred.

2.5 Basics of WSDL

WSDL is a specification defined to describe information about a Web Service in XML format. A WSDL document consists of information on how to use a Web Service.

WSDL provides the common methods to represent the data types passed in the service, the functions that are available in the service, and the mapping of service onto the network protocol.

WSDL 2.0 divides the description of the abstract functionality offered by a service from the specific description of the service description such as 'how' and 'where' that functionality is offered.

2.5.1 WSDL for Describing Web Services

A typical WSDL document contains the following information about the Web Service:

- Available methods
- Type of protocol to be used
- Parameters and return type of methods
- Location of Web Service

2.5.2 WSDL for Service Providers and Consumers

A WSDL document is usually created by service providers and used by a service consumer. For example, after a Web Service is developed, the service provider publishes information about its organization and the service on the registry. In other words, the service provider creates a WSDL document either manually or through a tool and publishes a link along with other information to a registry. Next, a service consumer searches the registry to locate a service of interest. After identifying a suitable service, the service consumer downloads the WSDL document. The service consumer's application de-serializes this WSDL document and creates a Java class from it. This Java class can now invoke the methods provided by the Web Service and starts using the Web Service.

Figure 2.7 shows the WSDL for service providers and consumers.

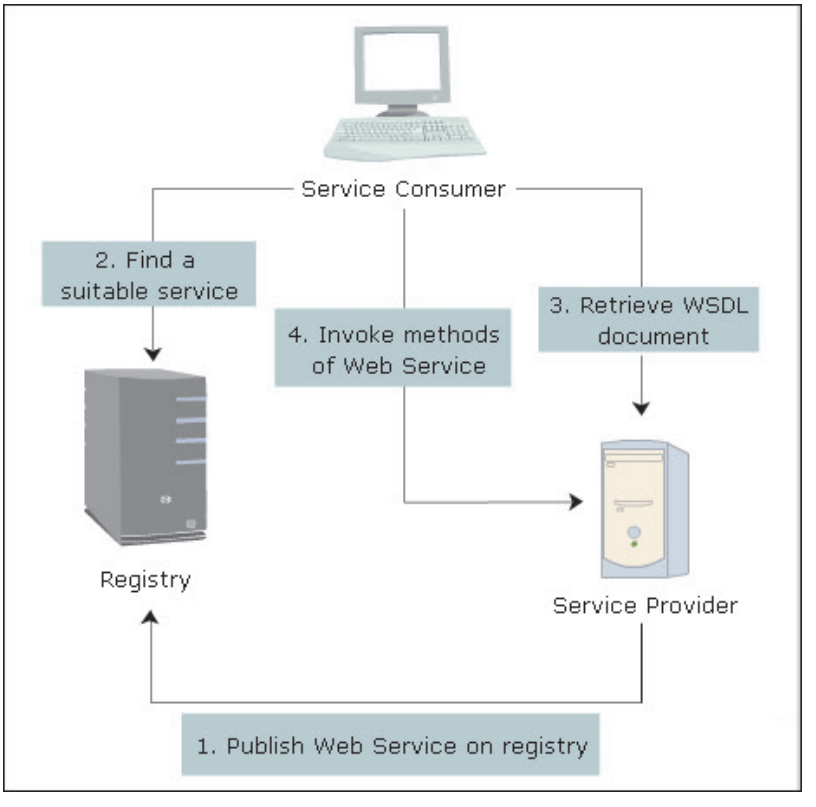


Figure 2.7: WSDL for Service Providers and Consumers

2.5.3 WSDL Document Structure

WSDL being an XML document begins with an XML declaration. The XML declaration specifies the XML version as 1.0 and the character encoding as UTF-8. A typical WSDL document consists of six elements namely, definitions, types, message, portType, binding, and service. Table 2.4 describes the WSDL document elements.

Elements	Description
description	• Acts as a container for the elements types, interface, binding, and service. • Defines the name of the Web Service and also one or more namespaces used by its child elements.
types	• Defines the data type of the information exchanged between applications. • Is mandatory only if the data type is other than the built-in data types of XML Schema. Example of XML schema's built-in types are string, integer, and so on.
interface	• Describes the operations that the Web Service includes and the input/output messages that are exchanged for each operation. • Describes the fault messages.
binding	Describes how the input and output messages of each operation will be transmitted over the Internet from one application to another.
service	Defines the address for invoking the Web Service. In other words, it provides the URL of the service provider's server on which the Web Service is hosted. The service consumer's application uses this URL to invoke the methods of the Web Service.
import	Used to import XML Schemas or WSDL document. This is an optional element.

Table 2.4: Elements of WSDL Document

Code Snippet 12 shows a sample WSDL document.

Code Snippet 12:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<description targetNamespace="http://ws.soap.syskan.com/"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-
200401-wss-wssecurity-utility-1.0.xsd"
xmlns:wsp="http://www.w3.org/ns/ws-policy"
xmlns:wsp1_2="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
xmlns:tns="http://ws.soap.syskan.com/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://www.w3.org/ns/wsdl"
xmlns:wsoap="http://www.w3.org/ns/wsdl/soap">
<types>
<xsd:schema>
<xsd:import
namespace="http://ws.soap.syskan.com/"
schemaLocation="http://localhost:8080/
SOAPWebService/SOAPWS?xsd=1"/>
</xsd:schema>
</types>
     <interface name="SOAPWS">
<operation name="add"
pattern="http://www.w3.org/ns/wsdl/in-out">
<input element="tns:add"
wsam:Action="http://ws.soap.syskan.com/
SOAPWS/addRequest"/>
<output element="tns:addResponse"
wsam:Action="http://ws.soap.syskan.com/
```

```
SOAPWS/addResponse”/>
</operation>
</interface>
<binding name=”SOAPWSPortBinding”
interface=”tns:SOAPWS”
type=”http://www.w3.org/ns/wsdl/soap”
wsoap:version=”1.1”
wsoap:protocol=”http://www.w3.org/2006/01/
soap11/bindings/HTTP/”>
<operation ref=”tns:add”>
<input/>
<output/>
</operation>
</binding>
      <service name=”SOAPWS” interface=”tns:SOAPWS”>
            <endpoint name=”SOAPWSPort”
binding=”tns:SOAPWSPortBinding”
address=”http://localhost:8080/
SOAPWebService/SOAPWS”/>
      </service>
</description>
```

The given Code Snippet first uses the description tag to specify the namespaces that will be used by the elements of the WSDL document. It then imports an XML schema that will be used by the WSDL document. Next, it uses an interface element to define an operation named add. Then, the binding element is used to specify that messages will be exchanged between the applications using SOAP. Finally, the service element is used to specify a name for the service and define an endpoint for the Web Service.

2.6 Web Service Registry

A dictionary consists of searchable collection of words in one or more specific languages. Similarly, a service registry consists of searchable collection of descriptions of Web Services.

2.6.1 What is UDDI?

Universal Description, Discovery, and Integration (UDDI) is a platform-independent, XML-based registry for businesses worldwide to list the business on the Internet. It provides standard mechanisms for businesses to describe and publish their Web Services, discover published Web Services and use them.

To use the registry, following steps are required:

- UDDI contains references to specifications called as Technical Models or tModels.
- The tModels describe the working of Web Services. They are built upon a programming model and schema that are platform and language independent.
- The software companies populate the registry by describing various tModels and specifications common to a business.
- UDDI programmatically assigns a Unique Universal Identifier (UUID) to each tModel and business registration.
- Marketplaces, search engines, and business applications query the registry to discover services of other companies and integrate this data with each other over the Web. Consequently, this becomes a process of searching and discovering automatically based on the available services. Figure 2.8 shows the UDDI registry.

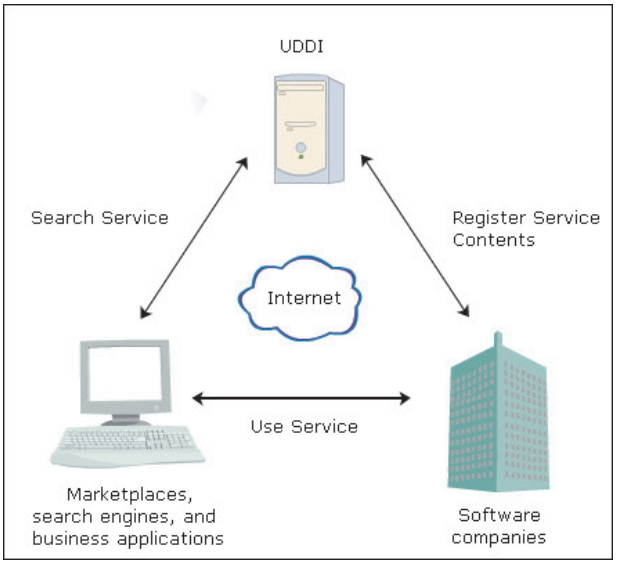


Figure 2.8: UDDI Registry

2.6.2 UDDI Data Structure

UDDI provides five core data structures. These are used to represent an organization, its services, implementation technologies, and relationships with the other businesses. The five data structures are as follows:

- **businessEntity:** Represents all the information about the business or the organization that provides the Web Service.
- **businessService:** Represents a Web Service.

- **bindingTemplate:** Represents the binding of a Web Service with its URL and its tModels.
- **tModel:** Provides information about a Web Service. It specifies the name of the service, brief description of the service, and a unique code that is used to identify a service in the registry.
- **publisherAssertion:** Represents a relationship between two business entities or service providers.

Figure 2.9 shows the UDDI data structure.

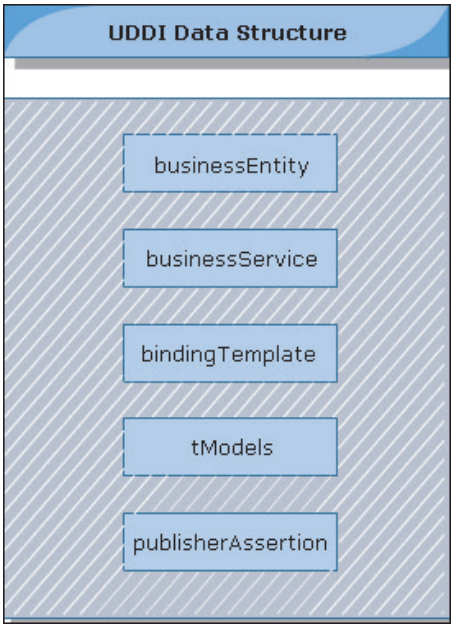


Figure 2.9: UDDI Data Structure

UDDI version 3 is the current specification of UDDI in use built on the vision of UDDI. The features of UDDI version 3 are as follows:

- It serves as a meta service to find Web services by enabling robust queries against rich metadata.
- It provides specification for building flexible, interoperable XML Web services registries useful in both private and public deployments.
- It consists of multi-registry topologies.
- It provides increased security features and improved WSDL support.
- It provides a new subscription API and core information model advances.
- It offers clients and developers a broad and complete blueprint of a description and discovery foundation for a diverse set of Web services architectures.

2.6.3 UDDI Publisher API

UDDI provides two APIs namely, Publisher API and Inquiry API. The Publisher API allows adding, modifying, and deleting service-related data from the registry.

Table 2.5 describes some of the methods available under UDDI Publisher API.

Methods	Description
Add_publisherAssertions	Adds relationship assertions to the existing set of assertions
save_business	Adds or updates one or more businessEntity entries
save_service	Adds or updates one or more businessService entries
save_binding	Adds or updates one or more bindingTemplate entries
save_tModel	Adds or updates one or more tModel entries
delete_business	Deletes one or more businessEntity entries
delete_service	Deletes one or more businessService entries
delete_binding	Deletes one or more bindingTemplate entries
delete_tModel	Deletes (or hides) one or more tModel entries
delete_publisherAssertions	Deletes specific publisher assertions from the assertion collection managed by a specific publisher
get_authToken (deprecated)	Logs into the registry
discard_authToken (deprecated)	Logs out of the registry
find_relatedBusinesses (deprecated)	Finds businesses that are related to a specified business key
get_publisherAssertions	Gets a list of publisherAssertion entries
get_registeredInfo	Gets an abbreviated list of businesses and tModels currently being managed by a given publisher
get_assertionStatusReport	Gets a summary of publisherAssertion entries
Set_publisherAssertions	Saves the entire set of publisher assertions for an individual publisher

Table 2.5: UDDI Publisher API Methods

2.6.4 UDDI Inquiry API

The Inquiry API is used to search and read data from a UDDI registry. It is used to query the UDDI registry and fetch specific UDDI data structures. Table 2.6 describes some of the methods available under the UDDI Inquiry API.

Methods	Description
find_business	Finds matching businessEntity entries
find_service	Finds matching businessService entries
find_binding	Finds matching bindingTemplate entries
find_tModel	Finds matching tModel entries

Methods	Description
find_relatedBusinesses	Finds information about businessEntity registrations that are related to a specific business entity whose key is passed in the inquiry
get_businessDetail	Gets businessEntity entries
get_serviceDetail	Gets businessService entries
get_bindingDetail	Gets bindingDetail entries
get_tModelDetail	Gets tModel entries
get_operationalInfo	Gets operational information related to one or more entities in the registry
get_registeredInfo (deprecated)	Gets an abbreviated list of businessEntity and tModel entries
add_publisherAssertions (deprecated)	Adds one or more publisherAssertion entries
set_publisherAssertions (deprecated)	Updates one or more publisherAssertion entries
delete_publisherAssertions (deprecated)	Deletes one or more publisherAssertion entries

Table 2.6: UDDI Inquiry API Methods

2.6.5 Electronic Business Scenario

Electronic business can be broadly defined as any business process that is carried out over the Internet.

Consider the case of two companies situated far away from each other and looking forward to start some business. Following are the details:

1. A small company in Los Angeles has a few skilled mobile application developers.
2. To establish a good international reputation, the company tries to achieve through Website.
3. Another small company in Norway is involved in mobile application development, but it does not have skilled developers.
4. When the Norway based company gets a new project, it looks for appropriate company.
5. The Norway based company checks the ebXML Registry for the desired company and finds the company in Los Angeles.
6. A contract is negotiated and both companies agree to do business together.

7. The Los Angeles based company sends the first mobile application back to Norway, which is accepted by Norway based company.

8. In return, the company transfers the money to Los Angeles.

Thus, both the companies work together in future. ebXML enabled both small businesses to meet and do electronic business together. Figure 2.10 shows an E-business scenario.

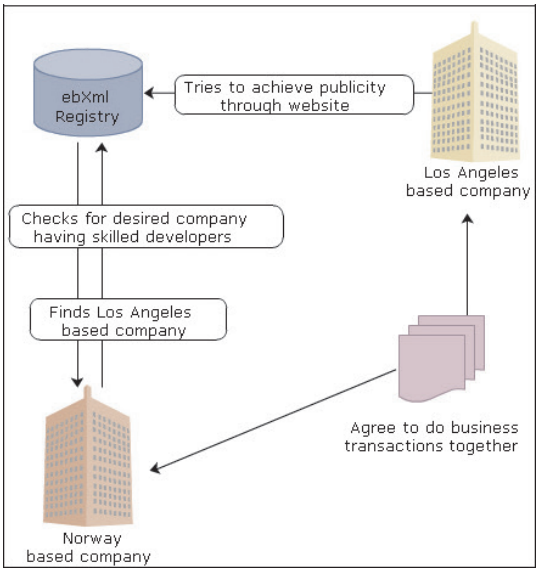


Figure 2.10: E-business Scenario

2.6.6 Electronic Business Process

Consider a scenario of how two companies, Exxon Corporation, a Los Angeles-based company and Nikido Electrics, a Norwegian company could implement a business that adheres to ebXML processes.

Following are the steps both the companies must do to do electronic business in ebXML format:

1. Exxon Corporation creates a Collaboration Protocol Profile (CPP) by defining itself in an ebXML compliant application.

2. Exxon Corporation submits its CPP to the registry and tries to achieve publicity through its Website.

3. Nikido Electrics, already registered at the ebXML registry/repository, is looking for new trading partners. It starts a new query and receives the CPP of Exxon Corporation. Using the CPP of Exxon Corporation and that of Nikido Electrics, ebXML derives a document called Collaboration Protocol Agreement (CPA).

4. Nikido Electrics contacts Exxon Corporation directly and sends the newly created CPA for acceptance.

5. After signing the contract, both the companies do electronic business directly following the business processes defined in the CPA.

Figure 2.11 shows the use of ebXML.

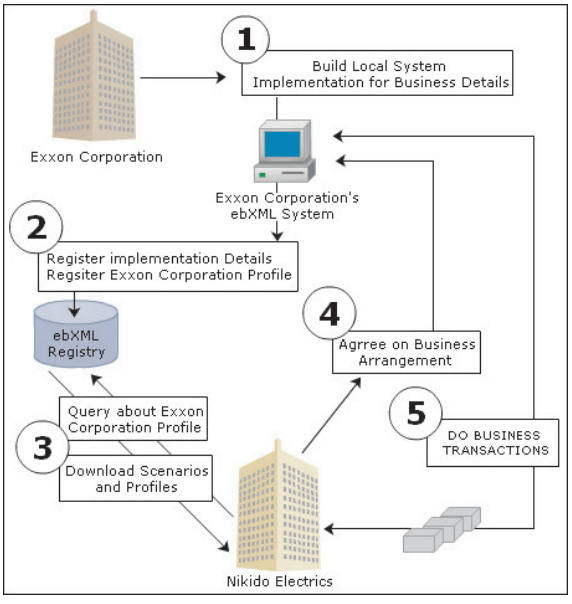


Figure 2.11: Use of ebXML

2.6.7 ebXML Registry and Repository

The ebXML architecture consists of components such as ebXML Registry and Repository, ebXML Business Processes, ebXML Collaboration Protocol Profiles and Collaboration Protocol Agreements, ebXML Core Components, and ebXML Messaging Service.

An ebXML registry and repository is a superset of UDDI. The ebXML registry stores information about the service provider and the Web Service. The ebXML repository also stores the business process documents of the service provider. Internally, the registry is connected to the repository. The clients communicate with the registry using the following two interfaces:

Object Manager

- Provides the methods to create new objects within the registry and affect state transitions on existing objects.
- Includes different methods namely, approveObjects(), deprecateObjects(), removeObjects(), submitObjects(), addSlots(), and removeSlots().

Object Query Manager

- Provides the methods to find and access the objects created within the registry.
- Includes different methods namely, getClassificationTree(), getClassifiedObjects(), getContent(), submitAdhocQuery(), and getRootClassificationNodes().

Figure 2.12 shows the ebXML architecture.

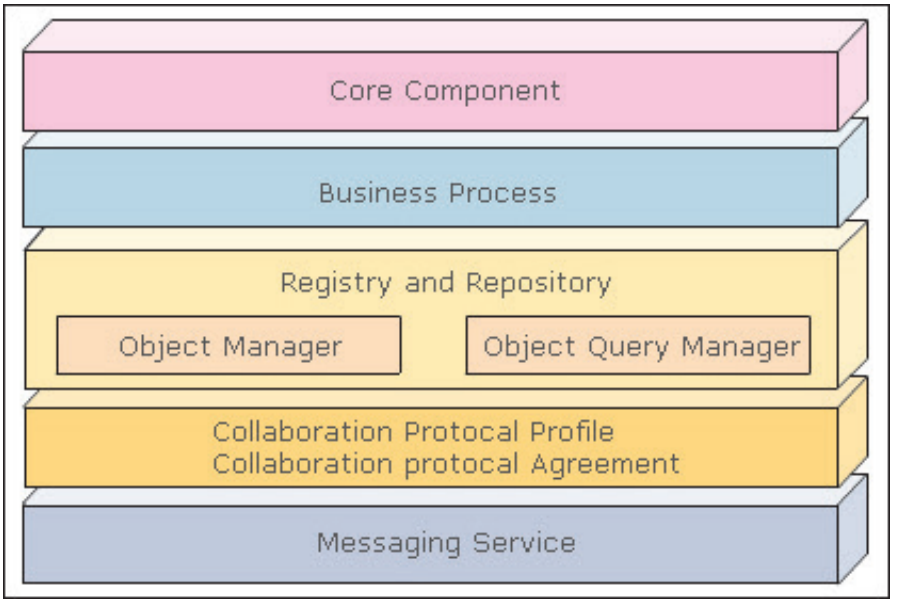


Figure 2.12: ebXML Architecture

2.6.8 ebXML Business Processes

The second component of ebXML architecture is Business Process. A Business Process is a set of individual messages exchanged among business partners. Thus, a Business Process is a business scenario that involves the exchange of information between two or more trading partners. It depicts who the trading partners are, their roles, the documents exchanged, and its structure. In other words, a Business Process defines the Business Collaboration.

Following are the main parts of Business Collaboration:

- **Business Collaboration Specification:** Is a set of roles interacting through a set of specialized protocol by exchanging Business Documents between business partners.
- **Business Transaction:** Is an atomic unit of work in a trading arrangement between two business partners and is conducted between two parties playing opposite roles in the transaction. The roles are always in the form of request and response.
- **Business Document Flows:** In Business Transaction, by default, each requesting role has one Business Document. The responding role may not have a Business Document. The Business Documents are composed from re-usable Business Information Objects. The ebXML reuses the Business Information Objects from a Core Component Library to create the Business Documents.
- **Choreography:** Is used to define which Business Transaction follows which Business Transaction. The realization of the choreography is done by the transition between two Business Transactions. A Binary Collaboration has several transitions and a transition has a 'from' Business State and a 'to' Business State.

- **Patterns:** Are a set of predefined transaction interactions. The use of predefined patterns combines the flexibility of specifying an infinite number of specific transactions and collaborations with a consistency. This facilitates faster design, faster implementation, and enables generic processing.

2.6.9 ebXML CPP and CPA

The third component of ebXML architecture includes the Collaboration-Protocol Profile (CPP) and Collaboration-Protocol Agreement (CPA) documents.

CPP:

- Is an XML document that contains information about a business and the way it exchanges information with other businesses
- Is an intersection of two CPP documents

CPA:

- Is derived from two or more CPPs

The working of CPP and CPA is as follows:

1. Two companies or trading partners create CPPs by adding information about the company.
2. Both companies submit their CPPs to the ebXML registry.
3. The CPPs are stored in the repository.
4. A company then queries the ebXML registry to get a suitable CPP from a potential trading partner.
5. Using the CPP of the two companies, ebXML derives the CPA.
6. The company sends the CPA to the potential trading partner for acceptance.
7. Upon agreement, the CPA gets registered in ebXML registry, and both the companies start the business.
8. After registering the CPA in the ebXML registry, both companies configure their Business Service Interface (BSI) software with the newly created CPA.
9. After the configuration, the BSI software is used to carry out electronic business.

Figure 2.13 shows ebXML CPP and CPA.

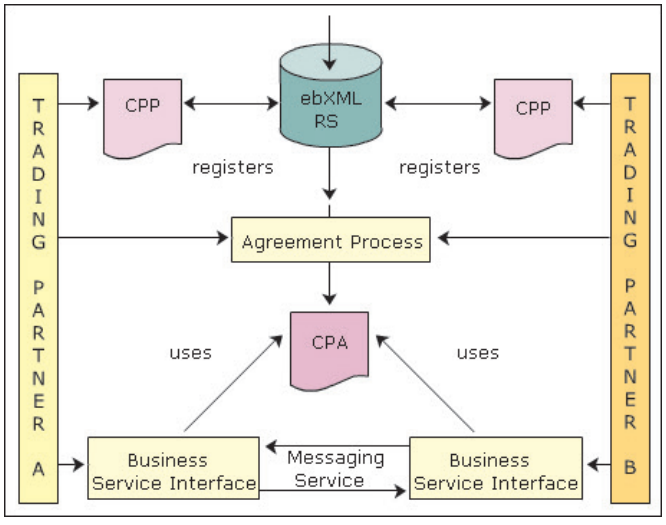


Figure 2.13: ebXML CPP and CPA

2.6.10 Core Components and Messaging Service

The last two components of ebXML architecture are Core Components and the Messaging Service. ebXML supports the concept of re-usability. Hence, in a CPP document, all kinds of objects are referenced.

Core component:

- Is the last entity that gets referenced in a CPP document
- Is also called Aggregated Component
- Is a reusable building block containing information about a business concept. For example, for a concept such as purchase order, the core components are date of purchase order, sales tax, and total amount

ebXML Messaging Service (ebXML MS):

- Provides the message exchange functionality within the ebXML infrastructure
- Is an entity used by the Business Service Interface software to send and receive XML messages from one point to another
- Is based on TCP/IP, FTP, HTTP, and SMTP protocols
- Uses packages data in SOAP messages and transmits them using HTTP, TCP/IP, FTP, or SMTP protocol

2.6.11 Java API for XML Registries (JAXR)

Java provides the JAXR API for accessing XML registries. JAXR provides a unified JAXR information model that describes the data and the metadata within the registries. This provision makes it easy for Java programmers to write programs that access a variety of XML registries, such as UDDI registry and ebXML registries. The JAXR metadata provides enhanced classification, association, and search capabilities.

JAXR is an abstraction-based API therefore; Java programmers can use this API to develop registry client programs that can be used across different target registries. JAXR works in conjunction with Java API for XML Processing (JAXP), Java Architecture for XML Binding (JAXP), Java API for XML-based Web Services (JAX-WS), and Java API for XML Messaging (JAXM) to enable Web services in Java EE7.

Code Snippet 13 shows how to use UDDI to publish and query a Web service.

Code Snippet 13:

```
public class publishWebService
{
public static void main(String[] args) throws
JAXRException
{
//Configure the ConnectionFactory to use JAXR Provider
// for UDDI
          System.setProperty("javax.xml.registry.
ConnectionFactoryClass", "com.ibm.xml.registry.
uddi.ConnectionFactoryImpl");
     ConnectionFactory connFac =
ConnectionFactory.newInstance();
//Configure URLs for UDDI inquiry and publish APIs.
  Properties p = new Properties();
  p.setProperty("javax.xml.registry.queryManagerURL", query_url);
  p.setProperty("javax.xml.registry.lifeCycleManagerURL", publish_
url);
connFac.setProperties(p);
//Create a connection to the UDDI registry
     Connection conn = connFac.createConnection();
//Specify credentials for accessing UDDI registry.
     PasswordAuthentication passwdAuth = new
PasswordAuthentication("Aptech_user", new char[]
{'p', 'a', 's', 's', '@', '1', '2', '3' });
Set userCredentials = new HashSet();
           userCredentials.add(passwdAuth);
           conn.setCredentials(userCredentials);
```

```
//Retrieve the
//javax.xml.registry.BusinessLifeCycleManager interface
            RegistryService regServ =
conn.getRegistryService();
            BusinessLifeCycleManager bLCM =
regServ.getBusinessLifeCycleManager();

//Create an Organization named "Aptech".
            Organization org =
bLCM.createOrganization("Aptech");
//Add the Organization to a Collection
            Collection coll = new ArrayList();
            coll.add(org);
//Save the Organization to the UDDI registry.
            BulkResponse bRes =
bLCM.saveOrganizations(coll);

//Obtain the Organization's Key from the response.
            if (bRes.getExceptions() == null)
{
            Collection res =
bRes.getCollection();
                  Key orgaKey =
(Key)res.iterator().next();
                  System.out.println("\nOrganization Key = " +
orgKey.getId());
            }
      }
}
```

The given Code Snippet first sets the ConnectionFactory to use the JAXR provider for UDDI by setting the ConnectionFactoryClass property. The code then specifies the URLs of the UDDI API by setting the queryManagerURL and lifeCycleManagerURL properties. It then creates a connection to the UDDI registry using the createConnection() method of the ConnectionFactory class. Then, the UDDI access credentials are specified using the PasswordAuthentication class and these credentials are passed to the setCredentials() method of the ConnectionFactory object.

Next, the code retrieves the BusinessLifeCycleManager interface that contains the methods corresponding to the UDDI publish API calls. Then, an organization is created using the createOrganization() method of the BusinessLifeCycleManager instance. This organization is then added to the Collection and made ready to be saved to the UDDI registry. Finally, the organization is saved to the UDDI registry and the organization's key is retrieved from the response.

Check Your Progress

1. Which of the following are methods of Object Manager interface?

(a)	getContent()
(b)	removeObjects()
(c)	removeSlots()
(d)	getClassifiedObjects()

(A)	a, b	(C)	c, d
(B)	b, c	(D)	a, d

2. Identify the UDDI Inquiry API method that finds matching businessEntity entries.

(A)	find_business	(C)	finding_binding
(B)	find_service	(D)	find_tModel

3. Which element of a WSDL document describes the various operations or methods provided by a Web Service and the input-output messages involved in each operation?

(A)	description	(C)	interface
(B)	types	(D)	service

4. Match the terms with their appropriate functionalities.

Term		Functionality	
(a)	ebXML	(1)	Provides standards for business
(b)	UDDI	(2)	Transfers data over Web
(c)	WSDL	(3)	Provides interoperability between applications using XML and HTTP
(d)	HTTP	(4)	Supports re-usability
(e)	SOAP	(5)	Describes Web Services

Check Your Progress

(A)	a-5, b-3, c-2, d-4, e-1	(C)	a-4, b-1, c-5, d-2, e-3
(B)	a-1, b-2, c-4, d-3, e-5	(D)	a-2, b-5, c-1, d-3, e-4

5. Arrange the code statements to establish a connection to the registry in the correct sequence.

(a)	connectionFactory.setProperties(props);
(b)	props.setProperty("javax.xml.registry.lifeCycleManagerURL", http://localhost:9080/uddisoap/publishapi");
(c)	Connection connection = connectionFactory.createConnection();
(d)	Properties props = new Properties();
(e)	props.setProperty("javax.xml.registry.queryManagerURL", "http://localhost:9080/uddisoap/inquiryapi");

(A)	d, e, b, a, c	(C)	a, b, c, d, e
(B)	a, b, d, e, c	(D)	c, d, a, c, e

Answers

1.	B
2.	A
3.	C
4.	C
5.	A

Summary

- SOAP overcomes the problems of EDI and RPC by using XML and HTTP.
- The structure of a SOAP message consists of Envelope element, Header element, and body element.
- Document/Literal messaging mode is used to transmit data in XML fragments, whereas RPC/Literal messaging mode is used to transmit method call and the values returned by a method.
- Transport Protocols deals with transportation of SOAP messages over HTTP.
- Errors in SOAP messages are sent as fault messages over HTTP.
- A WSDL document is an XML document that adheres to the WSDL XML schema. It generally consists of six elements namely, definitions, types, message, portType, binding, and service.
- An ebXML registry is similar to the UDDI registry only with the difference that along with storing information about the WSDL document it also stores the document too.

ASK to LEARN

Onlinevarsity
your e-way to learning

Questions *in your* mind?

are here to HELP

Post your questions in the ASK to LEARN section for solutions.

03 Web Service Endpoints

Welcome to the Session, **Web Service Endpoints**.

This session introduces the concept of Web Service Endpoints. Further, it explains packaging and deploying of a Web service. Lastly, the session explains Web Service invocation.

In this Session, you will learn to:

- Explain the guidelines to design Web service endpoints
- Describe the method to package to deploy a Web service
- Explain the process of invoking Web service

3.1 Web Service Endpoints

A Web service endpoint is a program that implements a Web service and carries out Web service requests. That is, the Web service endpoint is a URL where the service can be accessed by the client application. While designing the Web service, the developer has to understand the nature and kind of service provided by the Web service. Implementing the Web service design guidelines will simplify the process of creating Web service endpoints.

3.1.1 Web Service Design Guidelines

As stated earlier, to design an efficient Web service, the developer needs to understand the nature of the service. The client application sends the request to the Web service. The Web service processes the request and responds to the request. The implementing class must be annotated with either the javax.jws.WebService or the javax.jws.WebServiceProvider annotation.

The steps to create a Web service and a client are:

- Code the implementation class
- Compile the implementation class
- Package the files into a WAR file
- Deploy the WAR file. The Web service artifacts, which are used to communicate with clients, are generated by GlassFish Server during deployment
- Code the client class
- Use the wsimport Maven goal to generate and compile the Web service artifacts needed to connect to the service
- Compile the client class
- Run the client

3.1.2 Web Service Design Decisions

The Web service is available to the client along with the details of the service. The client locates the services according to the needs and makes the appropriate request. The business logic of the Web service processes the request and responds to the client. The steps to decide the design of the Web service is as follows:

1. **Decide whether and how to publish a Web Service:** The interface of the Web service depends on the following factors:

   - The type and nature of client calls to the service
   - The type of service endpoints used that is EJB or JAX – RPC service endpoint
   - The level of interoperability achieved

   The Web service can be published with restrictions that disallow the client availability or visibility.

2. **Determine how requests are received:** The request sent in by the client has to be converted into an internal format. This helps the business logic to process the client request quickly.

3. **Which protocol to be used for delegating request:** The incoming request can be sent to the business logic in many ways. Determining a protocol to achieve the same may reduce time and discrepancy.

4. **How requests are processed:** A Web service offers only an interface to the business logic. This step helps in determining how the interface can be used to handle Web service requests.

5. **Decide how responses are formulated and sent:** The client response formulated should be such that the client understands the response.

6. **Determine how problems are reported:** Errors can occur in any application. Hence, deciding how to throw or handle exceptions or errors and on the system or service levels, is of utmost importance while designing a Web service. This step also includes formulating a plan for recovering from errors and exceptions.

3.1.3 Layered View of Web Service

A Web service implementation involves two parts: an interaction layer and processing layer.

- **Interaction layer:** The interaction layer consists of the service endpoint interface that the service exposes to the clients. This layer contains the logic for delegating requests to the business logic and formulating responses. The major responsibility of a service's interaction layer is the design of the interface presented to the client.
- **Service processing layer:** The service processing layer consists of the business logic that is used to process client requests.

Implementing the Web service into a layered view helps to:

- get clarity on the division of responsibilities
- designate a single location for all request processing logic
- describe existing business logic as a Web service

Figure 3.1 shows the layered view of the Web service.

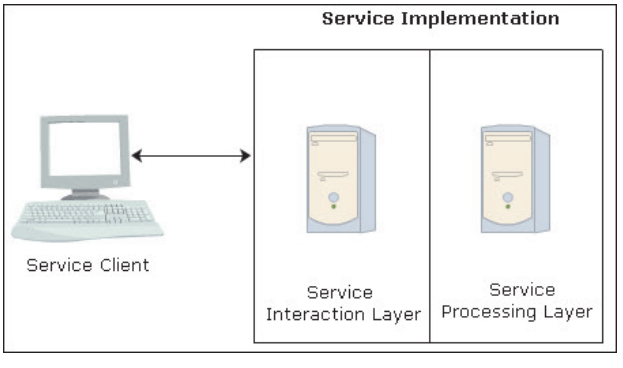


Figure 3.1: Layered View of Web Service

A Web service interface definition can be developed using any one of the two approaches, Service Interaction Layer approach or the Service Processing Layer approach. In the first approach, the Web Service Description Language (WSDL) is developed from the Java interfaces for the Web service. The second approach is a lesser used approach. It involves developing the WSDL document first and then building the corresponding Java interfaces from this description.

Some of the factors that influence the design of the interface are as follows:

- **Choice of interface endpoint type:** Two types of service endpoints, a JAX-RPC service endpoint and EJB service endpoint can be developed using the J2EE platform. A JAX-RPC service endpoint is used when the processing (or pre-processing) happens within the Web tier. An EJB service endpoint is used only when the processing happens on the EJB tier.
- **Granularity of service:** A Web service operation is a method call involving parameters, return values, and any errors or exception that it may generate. Each method call involves marshalling and unmarshalling of XML documents, remote procedure calls, and processor time. Hence, granularity of a Web service is always a trade-off between client-side flexibility and Web service performance.
- **Parameter types for Web service operations:** In a Web Service method, call parameters and return values are mapped to XML and sent as SOAP messages between the client and the service. At the receiving end, parameters have to be mapped back into the proper types or objects. These parameters can be standard Java objects, XML documents, or other non-standard types.
- **Interfaces with overloaded methods:** Overloaded methods present limitations to creating the WSDL description. In WSDL, each method call and the response are converted to SOAP messages. WSDL version 2.0 does not support SOAP messages with the same name. If a Web service developed using the Java-WSDL exposes over-loaded methods, there is a possibility of vendor–specific tools representing this in ways, which might not be compatible with the application.

3.1.4 Other Design Considerations

The interaction layer receives client requests in the form of SOAP messages and delegates them to the Web service business logic.

Other factors that influence the design of the service interaction layer are as follows:

- **Receiving requests**

  Some validation, transformation, and other required pre-processing of parameters is done before the call is delegated to the business logic. All such security checks, logging, auditing, input validation should be performed at the interaction layer as soon as the request arrives.
- **Delegating requests to processing Layer**

  Web service requests are processed either synchronously or asynchronously. If a request can be processed in a short time, then the client can be blocked until the processing is completed and the response is sent.

If a request has a longer turn-around time, then it is not advisable to block the client until the request completes processing.

- **Formulating response**

  The response to a method call is an XML document containing the return values. However, the response includes any difference that may result in the schema, after it is used by this XML document and then, by the client application. This should be done near the endpoint as it allows data caching and reduces extra trips to the processing layer.

3.2 Packaging and Deployment

A Web service becomes available to clients only after packaging the required files in the proper folders and deploying them on a server. The various deployment descriptors and other files needed for operating a Web service are packaged into archive files depending on the type of endpoint and then, finally deployed.

3.2.1 Web Service Annotations

The deployment descriptors that were used in J2EE 1.4 have been replaced by annotations in Java EE platform. Deployment descriptors are used only for web.xml to define the servlet specification.

Web service annotations are modifiers that are used to indicate the following:

- Web services
- Web methods
- Parameters used in Web methods
- Parameters used to initialize Web services or Web methods
- The result of the Web services

Annotations are prefixed with the @ symbol. JAX-WS 2.0 specification defines several annotations that can be used to define and use Web services. These annotations are available in the javax.jws package. This package provides APIs that are used to map the Java annotations to the WSDL file elements. Some of the annotations in the javax.jws package are as follows:

- **javax.jws.WebService (@WebService):** This annotation is used to specify that the Java Web Service (JWS) file implements a Web service. This annotation has five optional attributes namely, name, targetNamespace, serviceName, wsdlLocation, and endpointInterface.
- **javax.jws.WebServiceProvider (@WebServiceProvider):** This annotation is used to specify that a Web service is provided in the Provider implementation class. In this case, the class exposes only one Web service method unlike multiple Web service methods that can be exposed when using the @WebService annotation. The attributes of this annotation are the same as that of the @WebService annotation.

- **javax.jws.WebMethod (@WebMethod):** This annotation is used to specify that the method is a public operation offered by the Web service. This annotation has two optional attributes namely, operationName and action.
- **javax.jws.WebParam (@WebParam):** This annotation is used to specify the parameters required by the Web service and the behavior of the parameters. This annotation helps map the input parameters of the Web service with the WSDL elements. This annotation has four optional attributes namely, name, targetNamespace, mode, and header.
- **javax.jws.WebResult (@WebResult):** This annotation is used to specify the parameter that is returned by the Web service. It helps map the return parameter of the Web service with the corresponding element in the WSDL file. This annotation has two optional attributes namely, name and targetNamespace.
- **javax.jws.soap.SOAPBinding(@SOAPBinding):** This annotation is used to specify the mapping of the Web service with the SOAP message protocol. This annotation has three optional attributes namely, style, use, and parameterStyle.
- **javax.jws.soap.SOAPMessageHandler (@SOAPMessageHandler):** This annotation is used to specify a SOAP message handler in a SOAPMessageHandler array. The attributes of this annotation are name, className, initParams, roles, and headers. All these attributes except className are optional.
- **javax.jws.soap.initParams (@initParams):** This annotation is used to specify the array of name/value pairs that are passed to the handler during initialization. It is used as a value for the initParams attribute of the SOAPMessageHandler annotation. The attributes of this annotation are name and value, which must be specified whenever the initParams annotation is used.

3.2.2 JAX-WS Endpoint

In JAX-WS, an endpoint can be implemented based on the standard service endpoint or a provider-based endpoint using the @WebServiceProvider annotation. However, in JAX-WS, the implementation of a Service Endpoint Interface (SEI) is optional. A JAX-WS Web service that does not have an associated SEI is regarded as having an implicit SEI. A JAX-WS that has an associated SEI is regarded as having an explicit SEI. In addition, JAX-WS requires generic service endpoint interfaces unlike JAX-RPC that required specific service endpoint interfaces.

The service endpoint can be implemented as Web service by annotating the Java classes. Annotations can be used to describe the Web service in a service endpoint implementation without using the WSDL file. All the information that is generally provided in a WSDL file can be specified using the attributes of the annotations. These annotations can be specified on the service endpoint client and server or on the server-side class that implements the Web service.

When developing a JAX-WS Web service, you must annotate the Java class using the @WebService or @WebServiceProvider annotations. The @WebService annotation is used to define an SEI-based endpoint, while the @WebServiceProvider annotation is used to define a Provider-based endpoint.

When creating a SEI-based endpoint, you can explicitly add reference to a service endpoint interface using the endPointInterface attribute of the @WebService annotation. If this attribute has not been configured, the Web service will use an implicit SEI. You can then use the @WebMethod annotation to mark the methods that will be exposed by the service endpoint. All the public methods defined in the service endpoint are also exposed even if the @WebMethod annotation is not used.

When creating a Provider-based endpoint, you need to use a class that implements a strongly typed javax.xml.ws.Provider interface, such as Provider<Source> or Provider <SOAPMessage>. If the Provider implementation returns a null value, no response is required. However, if the @WebServiceProvider annotation does not specify a WSDL file and the Provider implementation returns a null value, JAX-WS sends a response that contains a SOAPEnvelope that includes an empty SOAPBody element.

3.2.3 Deployment Process

The process of deployment of the Web service depends on the sequence of the development of WSDL and creating service implementation. A Web service can be created by starting with developing the WSDL file first and then creating the service implementation or vice versa. However, the process of deployment is different for each of the approaches.

- **Starting with the service implementation:** JAX-WS helps to develop the endpoint interface and implementation. The WSDL document is developed using guidelines specified by JAX-WS. These guidelines involve mapping methods to the various elements of the WSDL file. However, a complete WSDL document cannot still be constructed from the endpoint interface alone. Implementation specific details like the binding and service elements which give information about messaging mode protocols and the actual URL of the endpoint have to be provided separately. The endpoint binding information is used to generate static JAX-WS stubs. When a client uses these stubs to access the Web service, it is known as static invocation of the Web service. Figure 3.2 shows a service endpoint.

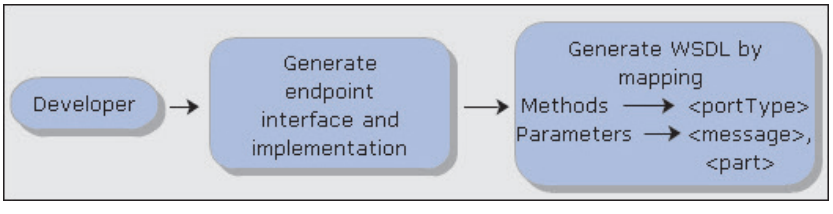


Figure 3.2: Service Endpoint

- **Starting with the WSDL:** Developing the WSDL document independent of the implementation helps create a WSDL document which is neutral in the XML types, idioms, and error handling capabilities. If the complete WSDL document is created along with the appropriate elements, the code required for marshalling SOAP messages to endpoint invocations can be generated. The JAX-WS compilers of some vendors are capable of generating an implementation class for the endpoint. The developer then needs only to implement the endpoint methods. Figure 3.3 shows WSDL endpoint implementation.

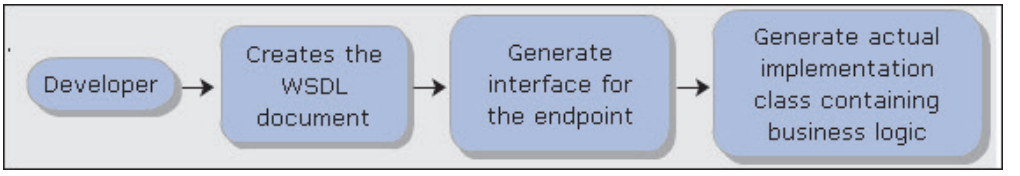


Figure 3.3: Endpoint Implementation

3.2.4 Publishing Web Service

A Web service is published in a registry to make it available to clients. Depending on the intended clientele for the Web service, the service is published on either a public registry, private registry (inter-enterprise registry), or an intra-enterprise registry. Such Web services can be used by the general public, trusted business partners, or just two entities within the same organization. Publishing a Web service involves making the details about the Web service, such as its interfaces, methods, parameters, service location, and so on available to clients. This description is made available in the WSDL document which is published in the registry. A registry may hold only the Web service's WSDL description or it may also optionally hold the XML schemas referenced by the service description. Undeploying a Web service involves disabling and removing a service endpoint from the Web container. All the associated files are removed from the server and other server resources if used are freed.

Code Snippet 1 demonstrates a simple Web service that takes two integer parameters and provides the sum of the two integers.

Code Snippet 1:

```
@WebService(serviceName = "CalculatorWS")
public class CalculatorWS
{
    /**
     * Web service operation
     */
@WebMethod(operationName = "add")
    public int add(@WebParam(name = "num1") int num1,
@WebParam(name = "num2") int num2)
{
```

```
                int sum = num1 + num2;

            return sum;

    }

}
```

The code shows a simple Web service named, CalculatorWS with one Web method named add(). This method accepts two integers, num1 and num2 from the user, calculates the sum of the input integers, and stores it in integer variable, sum. The code then returns the value of sum. Code Snippet 2 demonstrates the WSDL file of the CalculatorWS Web service.

Code Snippet 2:

```
This XML file does not appear to have any style information associated
with it. The document tree is shown here.

<!--

 Published by JAX-WS RI at http://jax-ws.dev.java.net. RI's version
is  Metro/2.3  (tags/2.3-7528;  2013-04-29T19:34:10+0000)  JAXWS-
RI/2.2.8 JAXWS/2.2 svn-revision#unknown.

-->

<!--

 Generated by JAX-WS RI at http://jax-ws.dev.java.net. RI's version
is  Metro/2.3  (tags/2.3-7528;  2013-04-29T19:34:10+0000)  JAXWS-
RI/2.2.8 JAXWS/2.2 svn-revision#unknown.

-->

<definitions     xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-utility-1.0.xsd"xmlns:wsp="http://
www.w3.org/ns/ws-policy"     xmlns:wsp1_2="http://schemas.xmlsoap.
org/ws/2004/09/policy"xmlns:wsam="http://www.w3.org/2007/05/
addressing/metadata"       xmlns:soap="http://schemas.xmlsoap.org/
wsdl/soap/"xmlns:tns="http://DJWS.com/"         xmlns:xsd="http://
www.w3.org/2001/XMLSchema"      xmlns="http://schemas.xmlsoap.org/
wsdl/"targetNamespace="http://DJWS.com/" name="CalculatorWS">

<types>

<xsd:schema>

<xsd:import namespace="http://DJWS.com/" schemaLocation="

http://localhost:8080/CalculatorWS/CalculatorWS?xsd=1"/>
```

```
</xsd:schema>
</types>
<message name="add">
<part name="parameters" element="tns:add"/>
</message>
<message name="addResponse">
<part name="parameters" element="tns:addResponse"/>
</message>
<portType name="CalculatorWS">
<operation name="add">
<input wsam:Action="http://DJWS.com/CalculatorWS/
addRequest" message="tns:add"/>
<output wsam:Action="http://DJWS.com/CalculatorWS/
addResponse" message="tns:addResponse"/>
</operation>
</portType>
<binding name="CalculatorWSPortBinding" type="tns:CalculatorWS">
<soap:binding transport="http://schemas.xmlsoap.org/soap/http"
style="document"/>
<operation name="add">
<soap:operation soapAction=""/>
<input>
<soap:body use="literal"/>
</input>
<output>
<soap:body use="literal"/>
</output>
</operation>
</binding>
```

```
<service name="CalculatorWS">
<port name="CalculatorWSPort" binding="tns:
CalculatorWSPortBinding">
<soap:address location="http://localhost:8080/
CalculatorWS/
CalculatorWS"/>
</port>
</service>
</definitions>
```

The WSDL document uses the definition element to define the name of the Web service as CalculatorWS and the various namespaces that is used in the document. It then uses the types of element to define the XML schema to be used for the Web service. Next, the message element is used to define the messages that will be mapped to the method invocation. Then, the portType element is used to map the add operation of the Web service to the input endpoint, addRequest, and the output endpoint, addResponse. The binding element is used to define the protocols and the data formats to be used for the messages and the operations. Finally, the service element is used to map the binding to the port.

3.3 Web Service Invocation

Invoking a Web service refers to the actions that a client application performs to use the Web service. To invoke a Web service, a Web Service Provider publishes the WSDL description and the referenced XML schemas for a Web service at a specific location on the Service Provider's server (typically a registry).

A client trying to access a Web service typically has to perform the following steps for accessing a Web service:

1. **Discover the Service Definition Interface (SDI):** A client must know the parameters required and the return types of a Web service's methods to make a valid invocation. This process of determining method signatures is known as discovering the service definition interface.

2. **Discover the Service Implementation:** The process of locating the actual Web service's address is known as discovering the service implementation.

3. **Bind with the Service Provider:** A client must bind to the specific location of the service to start invoking methods on it. This binding can be performed when a client is developed or deployed (static binding) or at runtime (dynamic binding). The type of binding that is, static binding or dynamic binding depends on whether the client is designed for use with a specific service or usable with all services.

Figure 3.4 shows the Web service invocation process.

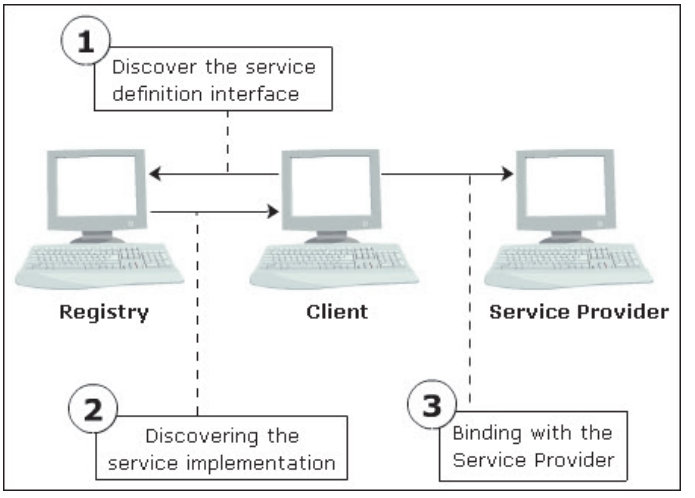


Figure 3.4: Web Service Invocation Process

3.3.1 Discovering the Service Definition Interface (SDI)

The three ways in which a SDI can be obtained from the Service Provider by a client are:

- **Direct:** A Web service client can directly retrieve the service description from the provider by using email, FTP, and so on.
- **HTTP GET Request:** A client can obtain the service description from the provider over the Web page by using a HTTP GET request.
- **Dynamic Discovery:** The service descriptions are stored in local or public registries such as UDDI or ebXML. A client looks up a Web service from these registries at runtime using a specialized set of API's. This is the most commonly used method of communicating amongst Web services and clients nowadays.

3.3.2 Discovering the Service Implementation

Web service clients query public or private registries to find Web service descriptions. These queries take the form of well-formatted XML messages, which are transmitted using standard protocols, such as SOAP or XML-RPC. Some common criteria used to find a service are service response time, accuracy of results, supported protocols amongst others.

When the appropriate service is located, the actual location of the service is returned to the requestor. Thus, the actual service implementation is obtained by the service requestor. Figure 3.5 shows the process of discovering the service implementation.

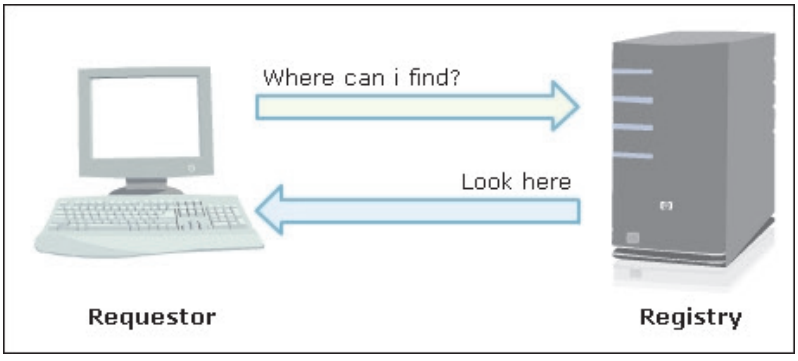


Figure 3.5: Discovering the Service Implementation

3.3.3 Binding with the Service Provider

After locating the service implementation, the client creates a message to be sent to the Service Provider. This message is sent to the provider by using the network protocols specified in the WSDL documents. Finally, the client of a Web service makes calls to the Web service using the API specified in the WSDL document. Figure 3.6 shows the process of binding to a service.

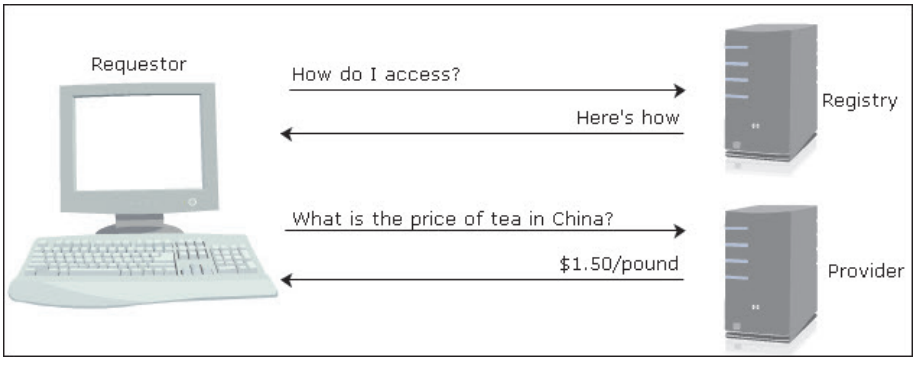


Figure 3.6: Binding to a Service

Code Snippet 3 demonstrates how to invoke a Web service using a JSP client.

Code Snippet 3:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
   <head>
<<meta http-equiv="Content-Type" content="text/html; charset=UTF-
8">
```

```
        <title>JAXWS Web Service Client </title>
    </head>
    <body>
        <h1>Accessing JAXWS Web Service CalculatorWS.</h1>
        <%-- start Web service invocation --%><hr/>
<%
try
{
//Instantiating the service and the port
com.djws.CalculatorWS_Service service = new
com.djws.CalculatorWS_Service();
com.djws.CalculatorWS port =
service.getCalculatorWSPort();

// initializing WS operation arguments
      int num1 = 25;
      int num2 = 15;
// processing result
      int result = port.add(num1, num2);
      out.println("Result = "+result);
}
catch (Exception ex)
{
      out.println("Exception :" + ex);
}
%>
<%-- end Web service invocation --%><hr/>
</body>
</html>
```

This JSP code first instantiates the Web service using the Service() method of the Web service, CalculatorWS. It then initializes the port for using the Web service using the getPort() method of the Web service. Next, the code invokes the Web service by calling the add() method of the Web service using the port instance. The method parameters are declared before calling the Web service method and are sent to the Web service while invoking it. The response of the Web service is then displayed in browser.

Check Your Progress

1. Which of these statements about packaging and deployment of a Web service are true?

(a)	The appropriate deployment descriptors are placed in the WEB-INF folder.
(b)	Starting the development of a Web Service with the WSDL file does not ensure neutrality in naming conventions, exception handling of the Web Service implementation.
(c)	A JAX-WS mapping file maps endpoint-interfaces, methods, and method parameters to portType, operation, and message definitions in the WSDL file.
(d)	The use of a registry cannot be eliminated while using a Web service.
(e)	A public registry may hold only the Web service's WSDL description or it may also optionally hold the XML schemas referenced by the service description.

(A)	a, e	(C)	b, c
(B)	a, b	(D)	c, e

2. Which of the following factor must be considered while deciding the Web service design?

(A)	Perform security checks, logging, auditing, and input validation at the processing layer.	(C)	Dividing the service implementation into layers.
(B)	Determine how problems are reported.	(D)	Describe existing business logic as a Web service.

3. "This is made up of the service endpoint interface that the service exposes to the clients. This layer contains the logic for delegating requests to the business logic and formulating responses." Which of the following layer suits the given description?

(A)	Interaction Layer	(C)	WSDL Layer
(B)	Service Processing Layer	(D)	Web Service Layer

Check Your Progress

4. To specify an explicit SEI for an SEI-based endpoint, which of the following attributes should be set?

(A)	serviceName	(C)	endPointInterface
(B)	wsdlLocation	(D)	operationName

5. Which of the following annotations is used to specify the parameter that is returned by a Web service?

(A)	@WebParam	(C)	@initParams
(B)	@WebResult	(D)	@WebMethod

Answers

1.	A
2.	B
3.	A
4.	C
5.	B

Summary

- A Web service endpoint is a program that implements a Web service and carries out Web service requests.
- To design an efficient Web service, the developer needs to understand the nature of the service.
- The Web service designed should be dynamic to work in all the applications efficiently.
- A Web service is available to clients only after packaging the required files in the proper folders and deploying them on a server.
- The process of deployment of the Web service depends on the sequence of the development of WSDL and creation service implementation.
- Invoking a Web service refers to the actions that a client application performs to use the Web service.

Onlinevarsity
your e-way to learning

WRITE-UPS BY

EXPERTS AND **LEARNERS**

TO PROMOTE NEW AVENUES AND
ENHANCE THE LEARNING EXPERIENCE

FOR FURTHER READING, LOGIN TO

www.onlinevarsity.com

04 Designing Web Service Clients

Welcome to the Session, **Designing Web Service Clients**.

This session begins with introduction to Web Service Clients. The session then, further describes the modes of communication. It also explains methods to locate and access the Web services. Lastly, the session explains handling of Web Service Exceptions.

In this Session, you will learn to:

- Explain Web Service Clients
- Describe the Modes of Communication
- Explain how to Locate and Access the Web Services
- Explain how to handle Web Service Exceptions

Session

04

Designing Web Service Clients

4.1 Web Service Clients

A Web service is invoked when the client accesses the Web service. Users can access a Web service through standalone clients or dynamic clients.

A Dynamic client is a Web-based application that uses JSP or servlets to access the Web services. This type of client can be of two types – Dynamic Proxy client and Dynamic Invocation Interface (DII). A Dynamic Proxy client creates a proxy of the service interface and then uses this interface to access the Service's methods. A DII uses Call objects to dynamically invoke a Web service. This enables the client to invoke methods of a service even without knowing their endpoint addresses until runtime.

A Standalone client can be a Java client or a Swing client. This type of client involves creating classes that enable the client application and the Web Service to communicate. Then, the class is typecast as an interface, which in turn is used to access the service methods.

4.1.1 Features of Dynamic Proxy Clients

The Dynamic proxy clients allow accessing the Web services by using their proxy interface. This makes the client applications more portable. This method of access is better than using standalone clients, which restrict the portability of the client applications. The Dynamic Proxy approach also provides the ease of development and increased performance rate as compared to a standalone client. The Dynamic Proxy approach is best suited to the applications that require being portable.

4.1.2 Creating Dynamic Proxy Clients

To access a Web Service using the Dynamic Proxy approach, a client developer needs to implement the following in the client application:

1. Create a Service object
2. Create a proxy using the Service objects getPort() method
3. Type cast the proxy as an interface
4. Invoke Web service using proxy object

After these steps are accomplished, the client developer must perform the following steps:

1. Import the interface class created by the wscompile tool.
2. Create the ObjectFactory object. This object is used to map the Web service methods with the objects in the client that is consuming the Web service.
3. Create a Service object with the help of ObjectFactory object's createService() method. This Service object represents the Web service that the client is

consuming. This object requires the WSDL file location, and a QName constructor instance. QName stands for qualified name which defines the attributes and URI for the methods exposed by the Web service. QName in turn requires a namespace URI and the service name.

4. After creating the Service object, use its getPort() method to create the proxy. The proxy can be used to access the Web Service's methods, once it has been typecast with the imported interface class.

Figure 4.1 shows the use of Dynamic Proxy Client.

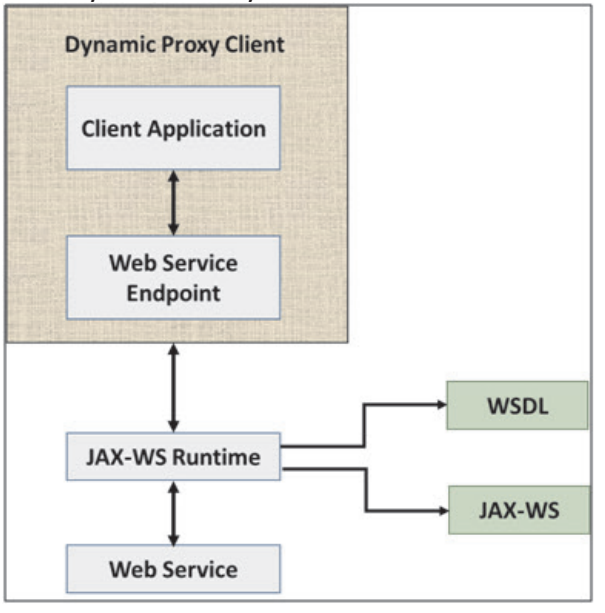


Figure 4.1: Dynamic Proxy Client

Code Snippet 1 demonstrates the use of ObjectFactory object.

Code Snippet 1:

```
@XmlRegistry

public class ObjectFactory

{

private final static QName _Add_QNAME = new

QName("http://ws.soap.djws.com/", "add");

private final static QName _AddResponse_QNAME = new

QName("http://ws.soap.djws.com/", "addResponse");

/*** Create a new ObjectFactory that can be used to create new
instances of schema derived classes for package: com.djws.soap.
ws **/

public ObjectFactory()
```

```
{
}
/*** Create an instance of {@link Add } **/
public Add createAdd()
{
return new Add();
}
/*** Create an instance of {@link AddResponse } **/
public AddResponse createAddResponse()
{
return new AddResponse();
}
/*** Create an instance of {@link JAXBElement }{@code <}{@link
Add }{@code >}} **/
@XmlElementDecl(namespace = "http://ws.soap.djws.com/", name =
"add")
public JAXBElement<Add> createAdd(Add value)
{
return new JAXBElement<Add>(_Add_QNAME, Add.class, null, value);
}
/*** Create an instance of {@link JAXBElement }{@code <}{@link
AddResponse }{@code >}}**/
@XmlElementDecl(namespace = "http://ws.soap.djws.com/", name =
"addResponse")
public  JAXBElement<AddResponse>  createAddResponse(AddResponse
value)
{
return new JAXBElement<AddResponse>(_AddResponse_QNAME,
AddResponse.class, null, value);
}}
```

The code shows the usage of the ObjectFactory class. It has two final static QName, QName_Add_QName for calling the add() method and QName_AddResponse_QName for the addResponse() method. The QName represents the attributes and namespace URI of the Web methods. It also has methods createAdd() and createAddResponse() to create an instance of add() and addResponse(), respectively. The code also has an annotated JAXBElement to dynamically link the Web service methods, add() and addResponse() to the similar methods of the client.

4.1.3 Features of DII

DII methods enable clients to invoke the methods of Web services, which is unknown at the compile time. The client application can lookup methods dynamically and access them through Call objects. This ability also provides the client developer with complete control over the client application. The DII method is the only method that supports one-way invocation. However, the client developer needs to develop very complicated code for DII. The effort for the same is much greater than the Static Stub and Dynamic Proxy methods.

4.1.4 Creating DII Clients

To access a Web service using the DII method, perform the following tasks:

- Create a Service object
- Create a QName class instance using the generated interface class
- Set Web Service operation name
- Invoke Web Service method

The Service object can be created using the createService() method with a QName object(service name) as the input parameter. Then, the Call object can be created. The Call object will enable the client to access to the Web service methods. Before using the Call object, the TARGET_ENDPOINT_ADDRESS and a few other properties such as SOAPACTION_USE_PROPERTY, SOAPACTION_URI_PROPERTY, and ENCODING_STYLE_PROPERTY of the Call object need to be set. Once the properties are set, the Call object can be used to invoke the Web Service's methods.

Figure 4.2 shows the use of DII Client.

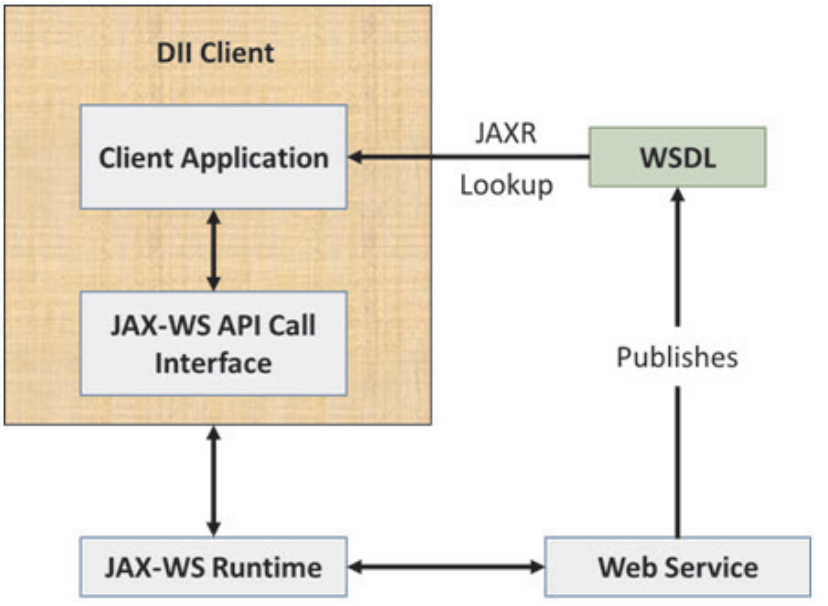


Figure 4.2: DII Client

Code Snippet 2 shows creation of a DII client.

Code Snippet 2:

```
@WebServiceClient(name = "CalculatorWS", targetNamespace =
"http://ws.soap.syskan.com/", wsdlLocation = "http://
localhost:8080/SOAPWebService/CalculatorWS?wsdl")
public class CalculatorWS_Service extends Service
{
private final static URL CalculatorWS_WSDL_LOCATION;
private final static WebServiceException CalculatorWS_EXCEPTION;
private final static QName CalculatorWS_QNAME = new
QName("http://ws.soap.djws.com/", "CalculatorWS");
static
{
URL url = null;
WebServiceException e = null;
try
{
```

```
url = new
URL("http://localhost:8080/SOAPWebService/
CalculatorWS?wsdl");
}
catch (MalformedURLException ex)
{
            e = new WebServiceException(ex);
}
CalculatorWS_WSDL_LOCATION = url;
CalculatorWS_EXCEPTION = e;
}
public CalculatorWS_Service()
{
super(__getWsdlLocation(),CalculatorWS_QNAME);
}
public CalculatorWS_Service(WebServiceFeature... features)
{
super(__getWsdlLocation(),CalculatorWS_QNAME, features);
}
public CalculatorWS_Service(URL wsdlLocation)
{
super(wsdlLocation, CalculatorWS_QNAME);
}
publicCalculatorWS_Service(URLwsdlLocation,WebServiceFeature...
features)
{
super(wsdlLocation, CalculatorWS_QNAME, features);
}
public CalculatorWS_Service(URL wsdlLocation, QName serviceName)
{
```

```
super(wsdlLocation, serviceName);

}

public CalculatorWS_Service(URL wsdlLocation, QName serviceName,
WebServiceFeature... features)

{

super(wsdlLocation, serviceName, features);

}

/**** @return returns CalculatorWS**/

@WebEndpoint(name = "CalculatorWSPort")public SOAPWS
getCalculatorWSPort()

{

return super.getPort(new QName("http://ws.soap.djws.com/",
"CalculatorWSPort"), CalculatorWS.class);

}

/**** @param features - A list of {@link javax.xml.
ws.WebServiceFeature} to configure on the proxy. Supported features
not in the <code>features</code> parameter will have their default
values. * @return returns CalculatorWS**/

@WebEndpoint(name = "CalculatorWSPort")public SOAPWS getCalculat
orWSPort(WebServiceFeature... features)

{

return super.getPort(new QName("http://ws.soap.syskan.com/",
"CalculatorWSPort"), CalculatorWS.class, features);

}

private static URL __getWsdlLocation()

{

if (CalculatorWS_EXCEPTION!= null)

{

throw CalculatorWS_EXCEPTION;

}

return CalculatorWS_WSDL_LOCATION;

}
```

This code shows the client implementation of a SOAP Web service, names CalculatorWS that is annotated with @WebServiceClient. This client maps the target namespace of the Web service with the WSDL location to access the Web Service methods using the CalculatorWSPort that is retrieved by the get CalculatorWSPort() method. This method contains overloaded Service Methods that are used to access the Web service. In addition, the code handles CalculatorWS_EXCEPTION.

This client implementation helps in accessing the Web service through its WSDL by using the Web service port.

4.1.5 Features of Standalone Clients

Standalone implementations provide the best performance compared to the other two dynamic approaches of accessing Web Services. The fact that developers can directly code their clients against the class also makes the standalone client method the easiest to code and implement.

However, this ease of programming also comes with a drawback. Since the client application is tied to the class, any change in the service interface would require the client developer to start all over again. Major changes in the service interface could also result in the development of the entire client application all over again.

Standalone clients are best suited to applications where the chances of the service interface changing are very rare. Using the standalone client approach in applications where the service interface frequently changes could heavily cost client developers in terms of development effort.

4.1.6 Creating Standalone Clients

To access a Web Service using the standalone client, a client developer needs to implement the following in the client application:

1. Create a Java class to invoke the Web service method.
2. Typecast the object as an interface.
3. Set the endpoint address for the generated object.
4. Access Web Service methods using generated object.

The object can then be used in the client application code to access the Web Service.

Figure 4.3 shows a standalone client.

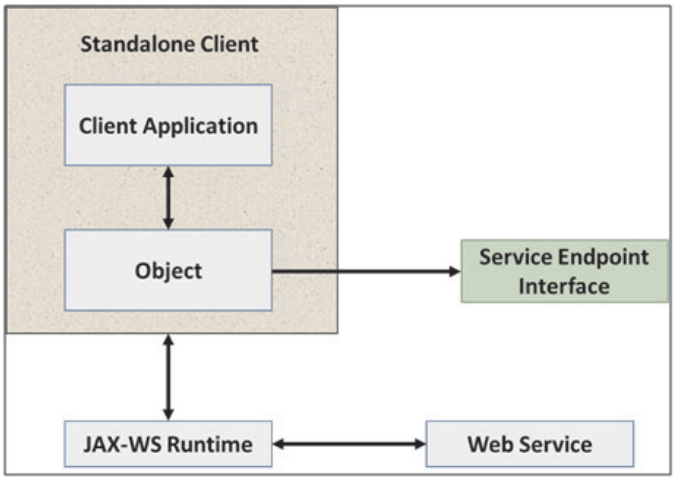


Figure 4.3: Standalone Client

Code Snippet 3 shows creation of a static stub client.

Code Snippet 3:

```
// Standalone Client
...
public static void main(String[] args)
{
try
{
int num1 = 5;
int num2 = 7;
int result = addNum(num1, num2);
System.out.println("Result = " + result);
}
catch (Exception ex)
{
System.out.println("Exception: " + ex);
}
}
```

```
private static int addNum(int i, int j)
{
com.djws.soap.ws.CalculatorWS_Service service = new
com.djws.soap.ws.CalculatorWS_Service();
com.djws.soap.ws.CalculatorWS port=service.getCalculatorWSPort();
return port.add(i, j);
}
...
```

The code shows creation of a standalone client. In the main method of this code, two integer variables, num1 and num2 have been declared and initialized with the values 5 and 7 respectively. The code then calls the add() method of the Web service by passing the two initialized variables i and j to consume the Web service in a static way. The add method opens the Web service port and calls its addNum() method to get the result.

4.1.7 Creating Web Tier Clients

Consider the scenario of Web client accessing the CalculatorWS Web service. The client has to enter two numbers on the browser page of the Web service. The Web component displays a browser page with the help of which end users can enter the two numbers. The client passes two numbers to the Web Service. The Web Service returns the sum of the two numbers, which are displayed to the user. Here, the Web component (such as servlet or JSP) plays the role of the Web service client. Large portions of Web applications, today, use Web services for various operations.

To create a Web Service client using a servlet, perform the following steps:

1. Generate the client stub
2. Write the Web client as a servlet or a JavaServer Page
3. Deploy the Web client

Code Snippet 4 shows creation of Web client as a servlet.

Code Snippet 4:

```
// Servlet Web client
...
@WebServlet(name = "JAXWSServlet", urlPatterns = {"/JAXWSServlet"})
```

```
public class JAXWSServlet extends HttpServlet
{
     @WebServiceRef(wsdlLocation = "WEB-INF/wsdl/localhost_8080/
CalculatorWS/CalculatorWS.wsdl")
    private CalculatorWS_Service service;
    /**
       * Processes requests for both HTTP <code>GET</code> and
<code>POST</code>
     * methods.
     *
     * @param request servlet request
     * @param response servlet response
* @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
      protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException
    {
response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter())
        {
            /* TODO output your page here. You may use following
sample
code. */
out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet JAXWSServlet</title>");
            out.println("</head>");
```

```
out.println("<body>");
            out.println("<form method=\"post\" action=\"\">");
              out.println("<h2>Consuming CalculatorWS Web Service
Using a Servlet Client</h2>");
                 out.println("First number: <input type=\"text\"
id=\"txtbox1\"
name=\"num1\" size=\"6\" maxlength=\"6\" value=\"\">");
            out.println("<br></br>");
                out.println("Second number: <input type=\"text\"
id=\"txtbox2\"
name=\"num2\" size=\"6\" maxlength=\"6\"value=\"\" >");
            out.println("<br></br>");
            out.println("<input type=\"submit\" name=\"btn\"
value=\"Add\">");
            out.println("<br></br>");
            int i=0;
int j=0;
          if (request.getParameter("btn").equalsIgnoreCase("Add"))
            {
               if(request.getParameter("num1").isEmpty() ||
request.getParameter("num2").isEmpty())
                 {
                out.println("Specify the two numbers to be added");
               }
                 else
                 {
                      i = Integer.parseInt(request.getParameter("n
um1"));
               j = Integer.parseInt(request.getParameter("num2"));
                    int result = addNum(i,j);
```

```
out.println("<h3><u>Result</u></h3>" );
                        out.println("Sum of the two numbers is " +
result);
                    out.println("</form>");
                    out.println("</body>");
                    out.println("</html>");
                }
            }
        }
    }
...
    private int addNum(int num1, int num2)
    {
        // Note that the injected javax.xml.ws.Service reference
as well as port objects are not thread safe.
         // If the calling of port operations may lead to race
condition some synchronization is required.
      com.djws.CalculatorWS port = service.getCalculatorWSPort();
        return port.add(num1, num2);
    }
}
```

The code shows how to consume the Web service using a servlet. While processing, the servlet sends a request to add two integer variables, num1 and num2 which are passed by the user at runtime by calling the addNum() method which in turn invokes the add() method of the CalculatorWS Web service. The Web service sends the result, which is displayed in the browser. The code also includes exception handling using try-catch block.

4.2 Modes of Communication

The client can communicate with the Web services using three methods. They are Dynamic Proxy, Dynamic Invocation Interface, and Standalone Client.

4.2.1 Using Dynamic Proxy Communication

The client directly accesses the Web Service through its proxy, that is, the client-side representation of the service endpoint interface. The Web Service client is programmed to interact with the endpoint service. This ensures that the client application is portable across different JAX-RPC runtimes. The JAX-RPC runtime is responsible for all communications between the service endpoint interface and the Web Service. The JAX-RPC runtime uses WSDL document and the JAX-RPC mapping files.

This invocation mode should be used only when the WSDL contains primitive schema types. Developers using dynamic proxies must create Java classes from client-side interface matching with service endpoint interface.

4.2.2 Using DII Communication

The DII approach enables clients to dynamically access Web Services through programmatic invocation of JAX-WS requests. Based on the Web Service WSDL document, a tool is used to create the service endpoint interface and other necessary class files. Even though the need for the JAX-RPC APIs is ruled out by the extra class files, DII uses them.

The DII client uses JAXR to search the registry for services. The client constructs a call from the information it receives from the service registry. The client then uses the constructed call to access the Web Service. Thus, like Dynamic Proxies, DII also uses JAX-RPC for its communication with the service endpoint interface.

4.2.3 Using Standalone Communication

The JAX-WS runtime generates a local object that acts as a proxy for the service endpoint. It enables communication between the client and the service. It converts client requests to SOAP messages and passes them onto the service. It reconverts the SOAP messages it receives from the service and passes them back as responses to the client.

4.3 Locating and Accessing Web Services

Regardless of which type of Web service client is used, the procedure to locate and access a Web service is the same. Code Snippet 5 demonstrates how to locate and access a Web service.

Code Snippet 5:

```
@WebMethod
@WebResult(targetNamespace = "")
@RequestWrapper(localName = "add", targetNamespace =
```

```
"http://ws.soap.djws.com/", className =
"com.djws.soap.ws.Add")
@ResponseWrapper(localName = "addResponse", targetNamespace =
"http://ws.soap.djws.com/", className =
"com.djws.soap.ws.AddResponse")
@Action(input = "http://ws.soap.djws.com/CalculatorWS/
addRequest", output =
"http://ws.soap.djws.com/CalculatorWS/addResponse")
public int add
(
@WebParam(name = "num1", targetNamespace = "")
int num1;
@WebParam(name = "num2", targetNamespace = "")
int num2;
);
```

Code Snippet 5 explains the details about Web methods and its parameters. The @RequestWrapper annotation is used to indicate the Java class that implements the wrapper for the parameters included in the request message that is sent to invoke a Web method. Similarly, @ResponseWrapper annotation is used to indicate the Java class that implements the wrapper for the parameters included in the response message that is sent by the invoked Web method. The @WebParam annotation is used to indicate the parameters of the Web method.

4.4 Handling Web Service Exception

Exceptions are errors or unexpected events encountered by an executing program. Consider there is an exception during the execution of Web service. The Web service is expected to capture not only the exception but also inform the client about the exception.

4.4.1 Types of Web Service Exceptions

There are two types of exceptions that can be thrown by a Web Service endpoint, System and Service-specific.

All exceptions thrown by errors beyond the control of the application can be categorized as System exceptions.

That is, all unanticipated errors that can occur at the time of service method invocation are called System exceptions.

For example, passing incorrect parameters during service invocation, unavailability of the server, network failure, network timeout, a SOAP message fault are examples of unanticipated errors that give rise to System exceptions. These errors can be handled by the client applications by,

- prompting the user to retry
- displaying the kind of exception occurred
- translating the system exception to an unchecked exception

Every exception would require a particular solution. Figure 4.4 shows the types of Web Service exceptions.

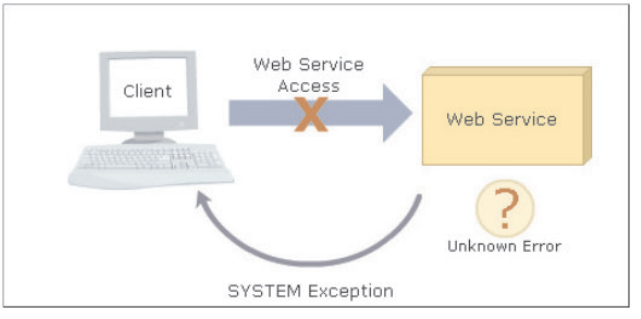


Figure 4.4: Web Service Exceptions

4.4.2 System Exceptions in Web Service

Dynamic proxy clients, DII Call objects, and static stubs, all three types of Web Service clients face different issues during execution resulting in different types of System exceptions.

- A javax.xml.ws.WebServiceException is one of the most common system exceptions thrown by dynamic proxy clients. Insufficient data to create the proxy is what prompts the getPort() method to throw this exception.
- IOException and javax.xml.ws.WebServiceException are common system exceptions thrown by DII Call interfaces. Unavailability of Service, network failures, and so on give rise to IOException while the use of invalid property names and setting them with invalid values are reasons for javax.xml.ws.WebServiceException.
- Standalone clients mostly face problems during the configuration of the stub. Configuration errors such as invalid property names, invalid property values, type mismatch, result in javax.xml.ws.WebServiceException.

Code Snippet 6 explains how runtime exceptions are handled in deployment descriptor of a Web application.

Code Snippet 6:

```
<error-page>

<exception-type>java.lang.Runtime</exception-type>
```

```
<location>/exception.jsp</location>
</error-page>
```

This code explains how runtime exceptions are handled in deployment descriptor of a Web application.

Figure 4.5 shows the dialog box to add exception page in the Deployment Descriptor (Web.xml) using NetBeans IDE.

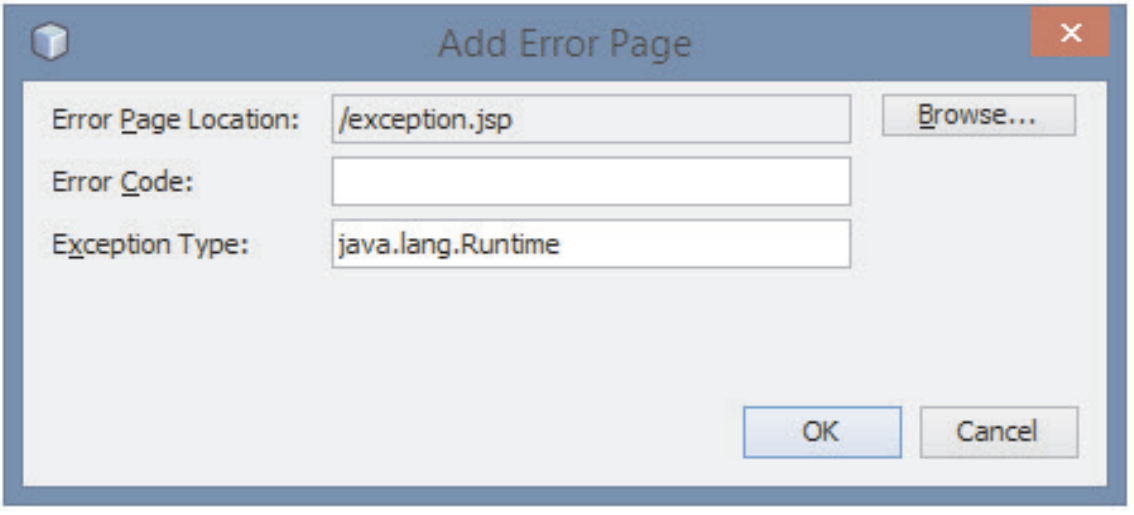


Figure 4.5: Add Error Page

Figure 4.6 shows the result after setting up Error Page for Web.xml in NetBeans.

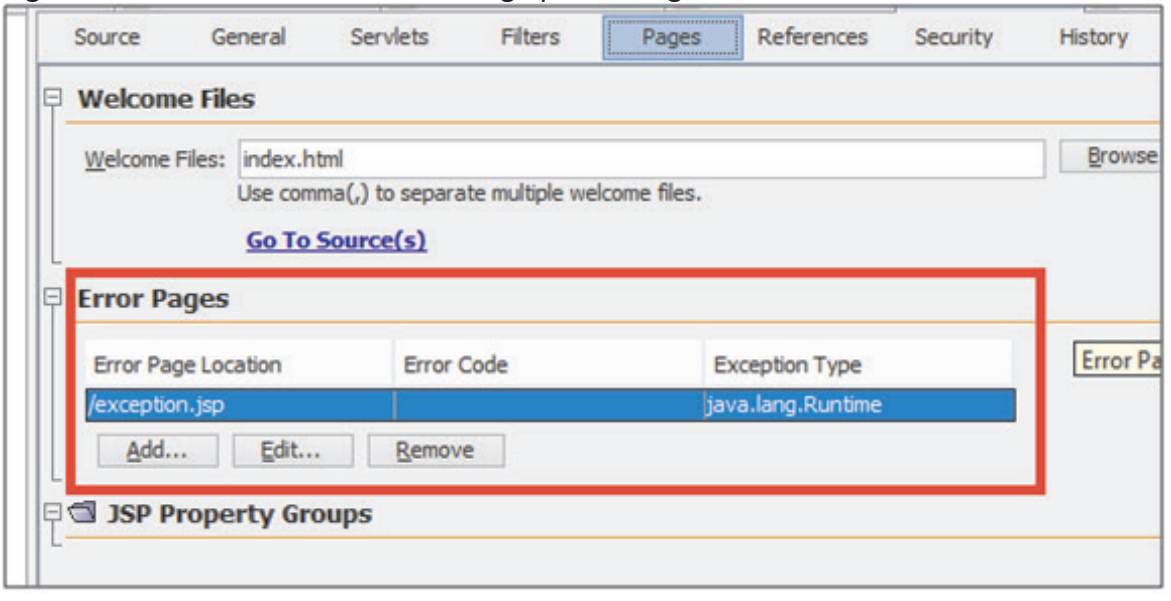


Figure 4.6: Error Pages

4.4.3 Service Exceptions During Web Service Calls

Unlike System exceptions, Service exceptions are thrown by faults or errors generated by the client application itself. These exceptions are also called checked exceptions in client applications. These faults or errors are a result of improper data passed to the Web Service. Since they are generated by the application itself, it is easy to determine these errors, and handle them. These exceptions are generally listed as operation elements in a WSDL file, and these are known as wsdl:fault elements.

JAX-WS tools can be used to map faults or errors to Java objects. These tools generate necessary exception classes and parameters to handle. These exception classes extend java.lang.Exception. The client application is responsible for handling these checked exceptions. The client application should also provide means to recover from such exceptions. DII communication mode returns all exceptions as java.io.IOException.

A client accessing the CalculatorWS Web service may pass to the service two non-numeral characters. The client will receive an illegalArgumentException, because the error message is defined in the WSDL document. Code Snippet 7 shows the use of illegalArgumentException.

Code Snippet 7:

```
<fault name=" illegalArgumentException" message="tns:
illegalArgumentException"/>
```

4.4.4 Exception Handling Mechanism

Client application developers are responsible for handling Service exceptions. They need to provide appropriate mechanisms to recover from these exceptions.

In case of Java EE 7 Web Components, clients may handle service exceptions as unchecked applications, such as javax.servlet.ServletException or may divert it to an error page.

Client developers can resort to boundary checking for input values, for example, they can check if credit card numbers that are entered, are 16-digit integers of particular range. Developers can use JavaScript to validate the boundaries before sending requests to a service. This validation will minimize multiple trips to the service, thereby reducing network traffic, and increasing service access speed.

A CalculatorWS Web service client may include the code to handle cases where a non-numeral input is provided. The CalculatorWS Web service throws an illegalArgumentException in the case of non-numeral character entry, it is handled as demonstrated in Code Snippet 8.

Code Snippet 8:

```
try{
com.djws.soap.ws.CalculatorWS_Service service = new com.djws.soap.ws.CalculatorWS_Service();
com.djws.soap.ws.CalculatorWS port = service.getCalculatorWSPort();
      int result = port.add(num1, num2);
}
catch (illegalArgumentException cc) {
System.out.println(cc);
}
```

In the code, the client passes strings instead of two numbers.

Web-tier clients can be redirected to an error page in case of an exception as demonstrated in Code Snippet 9.

Code Snippet 9:

```
<error-page>
<exception-type> illegalArgumentException</exception-type>
<location>/exception.jsp</location>
</error-page>
```

After the Web service client for the CalculatorWS Web service is created, it can be executed to test the Web service. For example, the Servlet Web service client code will produce an output as shown in figure 4.7 when the page is first loaded.

Figure 4.7: Output of Servlet Web Service Client

If the Add button is clicked without entering any numbers in the text boxes, an error message is displayed as shown in figure 4.8.

Figure 4.8: Error Message Displayed

After the numbers to be added are entered in the text boxes, the Web page appears as shown in figure 4.9.

Figure 4.9: Input Values Entered

After the Add button is clicked the sum of the two numbers is displayed as shown in figure 4.10.

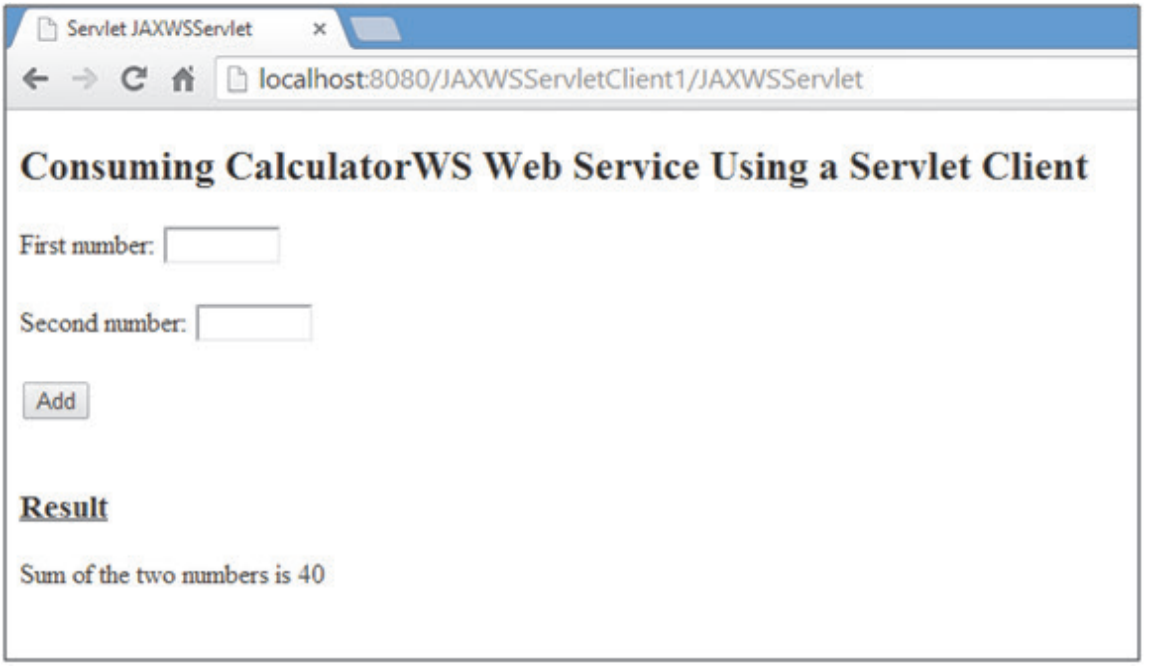


Figure 4.10: Result of the Addition Operation

Check Your Progress

1. Which of the statements about the features of the different types of Web service clients are true?

(a)	Standalone clients best suit situations where the interface changes frequently.
(b)	In the Dynamic Proxy method, the client creates a proxy of the service interface and then uses this interface to access the Service's methods.
(c)	Dynamic Proxy clients have the best performance compared to other approaches.
(d)	DII clients enable clients to invoke only those methods of Web services which are known during compile time.
(e)	The standalone method involves creating classes that enable the client application and the Web Service to communicate.

(A)	a, e	(C)	b, c
(B)	a, b	(D)	c, e

2. Match the Web Service Client communication approaches to their correct descriptions.

Approach		Description	
(a)	Standalone	(1)	The client directly accesses the Web Service through its proxy, that is, the client-side representation of the service endpoint interface.
(b)	Dynamic Proxy	(2)	This approach enables clients to dynamically access Web Services through programmatic invocation of JAX-WS requests.
(c)	DII	(3)	The JAX-WS runtime generates a local object that acts as a proxy for the service endpoint and enables communication between the client and server.

(A)	a-2, b-1, c-3	(C)	a-1, b-2, c-3
(B)	a-3, b-1, c-2	(D)	a-3, b-2, c-1

Check Your Progress

3. Dynamic Proxy clients throw the _____ system exception when a network failure occurs.

(A)	illegalArgumentException	(C)	javax.xml.ws.WebServiceException
(B)	RemoteException	(D)	IOException

4. The two types of exceptions that can be thrown by a Web Service endpoint are _____ and _____.

(A)	Dynamic and System	(C)	System and Service-specific
(B)	Proxy and Standalone	(D)	Service and Web specific

5. _____ and _____ are the common system exceptions thrown by DII Call interfaces.

(a)	IOException
(b)	JAX-WSException
(c)	javax.xml.ws.WebServiceException
(d)	RemoteException

(A)	a, c	(C)	b, d
(B)	a, b	(D)	c, d

Answers

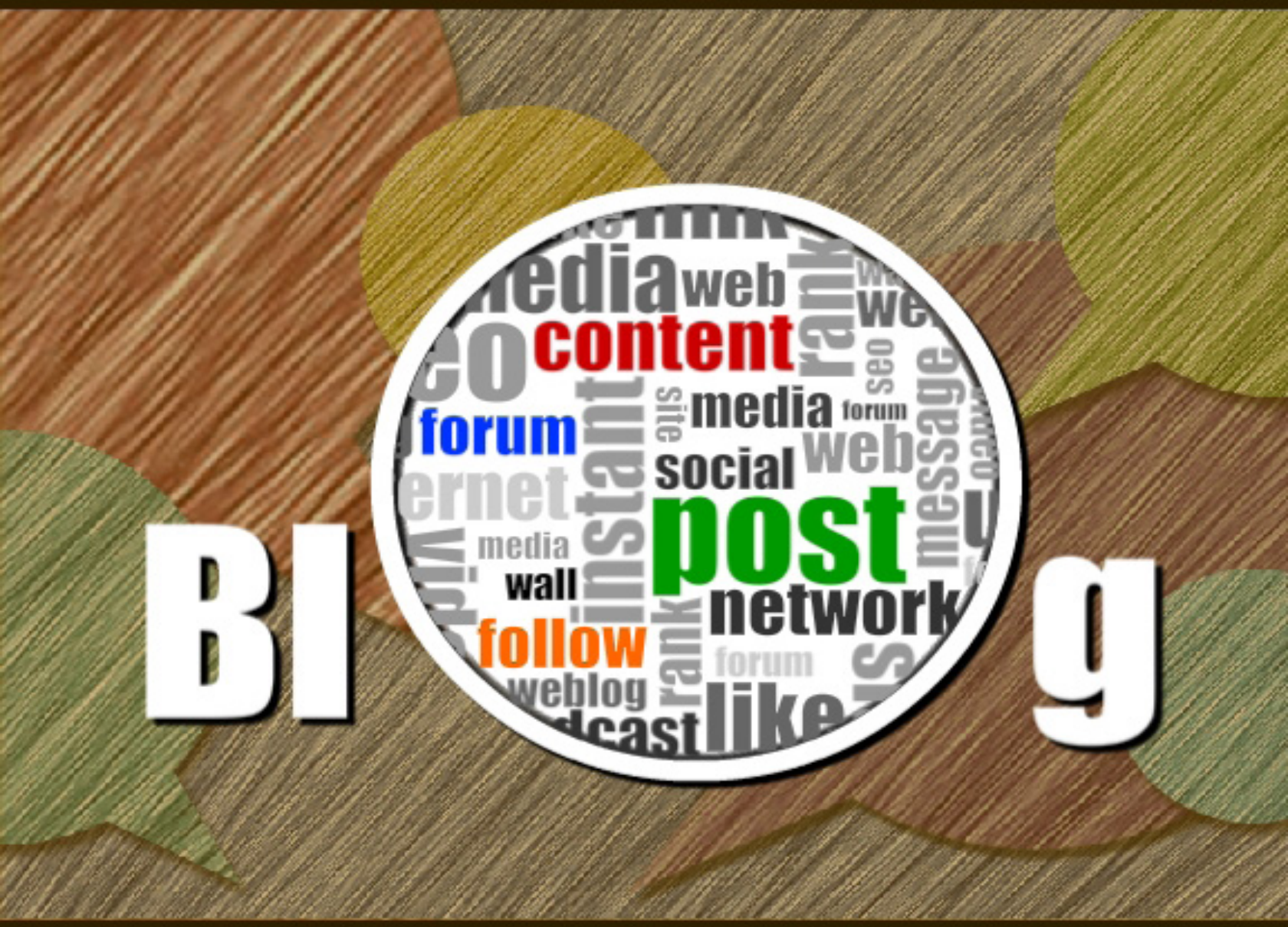
1.	A
2.	B
3.	D
4.	C
5.	A

Summary

- A Web service is invoked when the client accesses the Web service. Users can access a Web service through standalone clients or dynamic clients.
- A Dynamic client is a Web-based application that uses JSP or servlets to access the Web services. This type of client can be of two types – Dynamic Proxy client and Dynamic Invocation Interface (DII).
- A Dynamic Proxy client creates a proxy of the service interface and then uses this interface to access the Service's methods.
- A DII uses Call objects to dynamically invoke a Web service. This enables the client to invoke methods of a service even without knowing their endpoint addresses until runtime.
- A Standalone client can be a Java client or a Swing client.
- There are two types of exceptions that can be thrown by a Web Service endpoint, System and Service-specific.
- All exceptions thrown by errors beyond the control of the application can be categorized as System exceptions.
- Service-specific exceptions are thrown by faults or errors generated by the client application itself.

Onlinevarsity

your e-way to learning



Balanced Learner-Oriented Guide

for enriched learning available @

www.onlinevarsity.com

05

JAX-WS

Welcome to the Session, **JAX-WS**.

This session introduces Web services on the Java Enterprise Edition platform. It discusses Java API for XML based Web services (JAX-WS). It explains the various aspects of JAX-WS.

In this Session, you will learn to:

- Explain Web services on Java Enterprise Edition platform
- Explain Java API for XML based Web services (JAX-WS)
- Explain the need of JAX-WS
- Explain the features and standards of JAX-WS
- Explain the JAX-WS architecture
- Explain JAX-WS annotations
- Explain JAX-WS programming model

5.1 Enhancements of Web Services on Java EE Platform

Today, there is a need to develop enterprise applications with fewer resources in lesser time and effort. One of the platforms used to develop enterprise applications and Web services is Java Enterprise Edition (Java EE).

Java EE platform has powerful APIs that reduce the development time, build complex applications easily, and increase the productivity of the developer. It has more annotations, lesser XML configurations, more Plain Old Java Object (POJO), and simple packaging. The presence of annotations has enabled the removal of XML deployment descriptors. Annotations have metadata of information, which are configured by Java EE server at deployment and run time.

Some of the new technologies included on the Java EE platform are given in table 5.1.

Technology	Description
Java API for XML based Web Services (JAX-WS)	It is a Java API, to create Web services on a Java platform.
Java API for RESTful Web Services (JAX-RS)	It is a Java API to support creation of Web services based on REpresentational State Transfer (REST) architecture.
Dependency Injection	It is a technique that provides the required objects to the software components.

Table 5.1: Technologies in Java EE Platform

5.2 Types of Web Services

Web services are classified as 'Big' Web services and RESTful Web services based on their implementation method.

5.2.1 'Big' Web Services

'Big' Web services or 'SOAP-based' services are based on JAX-WS in Java EE. SOAP is a standard protocol that defines the architecture and message formats in XML language. It follows HTTP protocol for request–and–response model on the Web.

Web services are written in Web Services Description Language (WSDL). WSDL is a standard XML format that describes the network service elements such as messages, bindings, location, and ways to communicate with the service. This WSDL service description is published on Web to be accessed by clients.

Figure 5.1 shows SOAP-based communication.

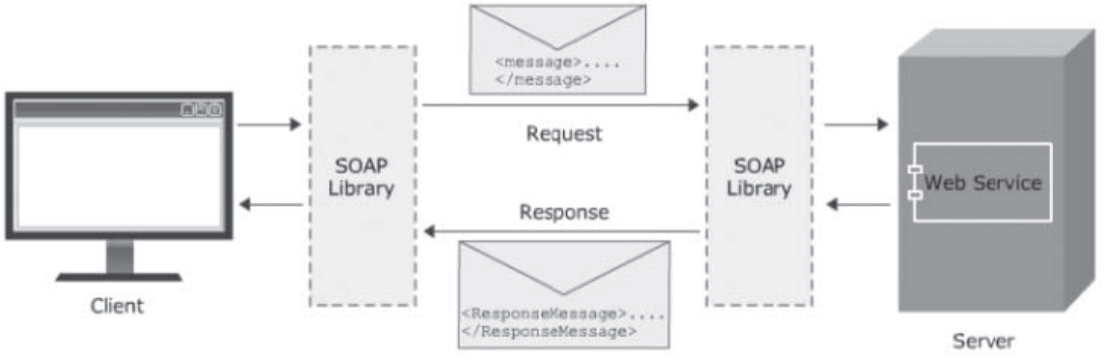


Figure 5.1: SOAP-based Method of Communication

From the figure, it can be seen that the Web service requestor uses the SOAP libraries to send a SOAP message. The requestor uses SOAP libraries to receive the message as a response from the Web service.

5.2.2 RESTful Web Services

RESTful Web service is a Web application that is based on client-server architecture called as the REST architecture. This architecture has a request-and-response model. It uses HTTP, which is based on Uniform Resource Identifier (URI) on the hyperlink, to access resources. The resources are in the form of text/html, text/xml, audio, and images.

RESTful Web services applications use Web URIs to expose resources (data and functionality). Resources are Created, Retrieved, Updated, and Deleted (CRUD) by means of four main HTTP methods.

Table 5.2 shows the mappings of HTTP methods to their CRUD actions.

HTTP Method	CRUD Action
POST	Create a new resource from the request data
GET	Retrieve a resource
PUT	Update a resource from the request data
DELETE	Delete a resource

Table 5.2: HTTP Methods

REST applications use standard interfaces and protocols to exchange resources on client and server-side. They are simple and have high performance. Unlike SOAP, REST has no official standard. It uses Web standards such as HTTP, URL, and XML. The functionality for RESTful Web services in Java EE is provided by JAX-RS.

In REST-based communication, a client initiates the request for the resource. In response, the client receives representation of the resource.

Figure 5.2 shows REST-based communication.

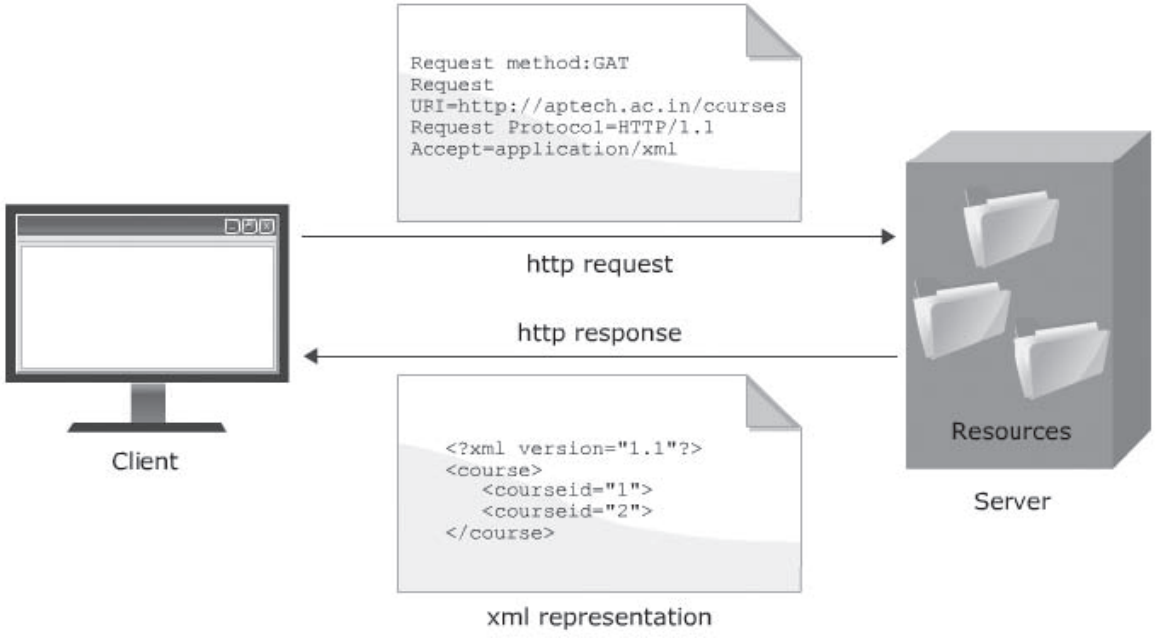


Figure 5.2: REST-based Method of Communication

5.3 Java API for XML Web Services (JAX-WS)

Web services support a few standards that help to develop interoperable applications. The key standards are as follows:

- WSDL – To define the service
- SOAP protocols – To exchange XML messages over different transport protocols (HTTP, SMTP, and JMS)
- Java API for XML-based Remote Procedure Call (JAX-RPC) – To invoke a Web service on Java platform

To invoke Java Web services, JAX-RPC has defined a few APIs based on its own data bindings. These APIs support SOAP protocol over HTTP to exchange XML messages. To align with the support for new industry standards, JAX-RPC was modified and was renamed with JAX-WS. Hence, JAX-WS is the successor of JAX-RPC 1.1 aligned with the new standards to develop Web services on Java EE platform.

JAX-WS supports annotations that make development of Web service applications simple and easy. It uses JAXB 2.0 for data binding and helps in customization to control generated Service Endpoint Interface (SEI). It reduces the size of runtime jar files. It assists the document-oriented Web services and Remote Procedure Call (RPC) Web services.

5.3.1 JAX-WS Standards

Table 5.3 shows the new standards of JAX-WS programming model.

Standard	Description
SOAP 1.2	It is a lightweight protocol for structured information exchange of text and binary data.
XML/HTTP	It assists in transforming XML messages over transport protocol, HTTP without SOAP.
Web Services Interoperability (WS-I)	It has WS-I Basic Profile document that has specifications for SOAP and WSDL. WS-I Basic Profile 2.0 has the specified encoding styles, proxy generations, and dynamic invocation of interfaces.
Data Mapping Model	It specifies the mapping of XML elements to Java classes. JAXB 2.0 promotes the mappings for all XML schemas. It is supported by JAX-WS for data mapping.
Interface Mapping Model	It is used to map service interfaces with service implementation classes.
Dynamic Programming Model	It is used for message-oriented invocations and asynchronous invocations.
Handler Model	It processes the SOAP message before and after the messages are sent over the network in the Web service development.

Table 5.3: Standards for JAX-WS Programming Model

5.3.2 Features of JAX-WS

Following are the key features of JAX-WS programming model that help in developing the Web services on the JEE platform:

- **Platform Independence:** The development of Web services for server and client-side applications is simplified to achieve better platform independence. A proxy class called as delegate class is generated internally by JAX-WS. It is an improvement over vendor-specific stubs used in JAX-RPC implementation. This class has all the methods of the Web service and processed annotations to identify it as Web service.
- **Annotations:** JAXWS has a library of annotations that help to convert Plain Old Java Objects (POJO) to Web services. These annotations define the information obtained from the deployment descriptors. In the Web service class, additional metadata information can be added on method level or on the parameters of the methods. Annotations help to expose Java artifacts as Web service.
- **Invocation of Web Services:** There are two types of client-side invocations supported by JAX-WS. They are synchronous and asynchronous invocation.

- **Synchronous invocation** – In this, a client sends a request for invocation of a service and waits till the response is received from the server.

  If there is an immediate response, then synchronous communication has to be used. RPC-oriented Web services are based on synchronous invocation.

- **Asynchronous invocation** – In this, the client sends the request to the service endpoint. The client need not wait for the response to be returned from the Web service. Document-oriented Web services are based on asynchronous invocation. There are two types of asynchronous invocation approaches. They are:

  - **Polling approach** – In this, the client first sends a request to the Web service endpoint and gets an object as response. This response is polled to know whether the server has responded. When the server sends an actual response, it is received in the response object.

  - **Callback approach** – In this, a callback handler, provided by the client receives the response object and processes it.

- **Data Binding with Java Architecture for XML Binding (JAXB 2.0):** JAX-WS, the successor to JAX-RPC, does not define any Java/XML binding. JAX2.0, defines standards for binding Java and XML schemas as part of JAX-WS deployment. JAXB2.0 abstracts the conversion of XML schema components and assists JAXB binding customization.
- **Support for SOAP 1.2:** JAX-WS supports SOAP 1.2. Binary attachments such as images or files are sent as Web service request using SOAP 1.2.
- **Support for Message Transmission Optimized Mechanism (MTOM):** MTOM is a mechanism supported by JAX-WS for optimized transmission of binary data in Web service.
- **Dynamic and Static Clients:** JAX-WS supports a dynamic client API called javax.ws. Dispatch. Dispatch API is a low-level API, which requires construction of a XML based messages by the client. The data is sent in PAYLOAD mode or MESSAGE mode. These modes are defined as constants in the dispatch client. The constants are as follows: javax.xml.ws.Service.Mode.PAYLOAD and javax.xml.ws.Service.Mode.MESSAGE.

  - **PAYLOAD mode** – In this mode, the client provides only the contents of soap:Body element. The soap:Envelope and soap:Header elements are added by the JAX-WS in the SOAP:Envelope element.

  - **MESSAGE mode** – In this mode, the client provides the entire soap envelope. The soap envelop has soap:Envelope, soap:Header, and soap:Body elements. JAX-WS adds no extra information in the message. It supports static client programming model known as proxy client. In this model, proxies are created from the generated SEI at client. Then the methods are invoked on the proxies in the client application.
- **Development Tools:** JAX-WS supports two command tools for Web service development.

They are as follows:

- **wsimport tool:** This tool is used when Web service is developed with the WSDL file. It generates the Web services portable artifacts. The tool processes WSDL file to generate Web service implementation class, service interface, and the underlying JAXB2.0 Java classes from XML schema elements.
- **wsgen tool:** This tool is used when the Web service development is with implementation class. It generates Web service artifacts such as Java interfaces and WSDL from Web service implementation class.

5.3.3 Web Service Stack

The JAX-RPC has its own data binding and parsing mechanism. However, JAX-WS data binding and parsing are based on the JAXB and StAX, the standards proposed by Sun.

Figure 5.3 shows the Web service stack for JAX-RPC and JAX-WS proposed by Sun. It shows the standards of JAX-WS and its comparison with JAX-RPC, which is the initial implementation of Web services.

Figure 5.3: Web Service Stack for JAX-RPC and JAX-WS

5.3.4 JAX-WS Architecture

The JAX-WS architecture is based on the generation of dynamic proxy class instance. This instance is responsible to send and receive the SOAP request and SOAP response to the client and server. Figure 5.4 shows JAX-WS architecture.

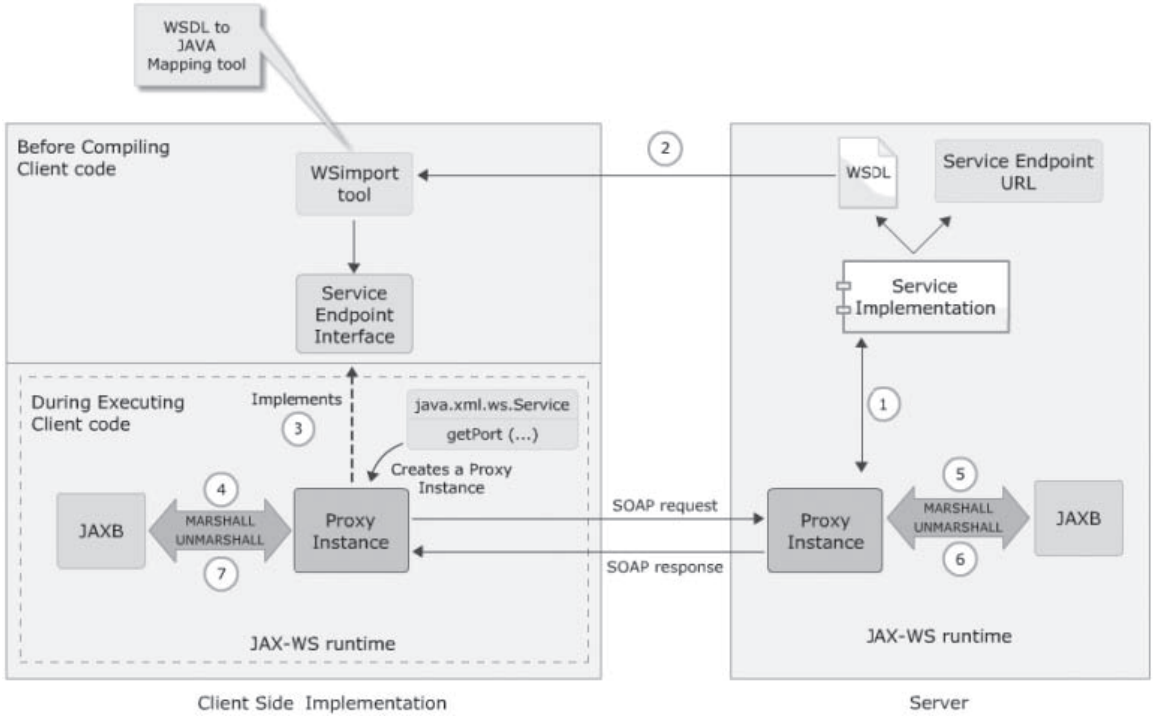


Figure 5.4: JAX-WS Architecture

Following are the steps to invoke the Web:

1. At the server-side, the development of JAX-WS starts with an annotated Web service implementation class. This implementation class is deployed on an application server. Then, the wsgen tool generates the WSDL file, which has information on service methods and the SEI. While deploying, a server side proxy instance is generated to handle the request and response.

2. At the client-side, the Web service is invoked using a command tool called wsimport. This tool is a WSDL to Java mapping tool. It generates the SEI on the client-side. Then, the client code for accessing the Web service is compiled and executed.

3. After the execution of client application code, JAX-WS runtime generates a proxy class instance. This instance internally implements the generated SEI. The proxy class is invoked by the getPort() method from javax.xml.ws.Service class. The generated proxy instance is used for invoking the Web service method.

4. After invoking the method on Web service by the client, the JAX-WS runtime uses JAXB to marshal the parameters of the method. Then, these objects are encapsulated in SOAP message in XML format. These are sent as a SOAP request across the network. The parameters are passed to proxy instance to be passed to the service present at the server.

5. After the SOAP request is received by the server the JAX-WS runtime environment uses JAXB to unmarshal the SOAP request.

Unmarshalling converts the SOAP message having XML schema elements into Java objects. The message is given to the server-side proxy to execute the Web service method.

6. At the server, the result of the Web service method is marshalled again using JAXB. JAXB wraps the response in the SOAP message, to enable the server-side proxy to send it back to the client.

7. At the client-side, the JAX-WS runtime unmarshalls the SOAP response message using JAXB. Then it passes the unmarshalled Java object to the proxy. The proxy returns the result from the Java object to the client application.

5.4 JAX-WS Annotations

JAX-WS has a library of annotations to simplify Web services development. Annotations enable easy conversion of a POJO class to a Web service. At the server, annotations describe the way of accessing an implementation class as a Web service. At the client, annotations describe the way a client-side Java class accesses Web service. The application uses the metadata information defined on service implementation class or service interface. Specialized tools process the metadata to generate the underlying code.

Annotations provide attributes that can be used to keep the required information in the implementation class itself. Therefore, it is optional to have webservices.xml deployment descriptor file.

5.4.1 Web Services Metadata Annotations

Web service metadata annotations describe the way to expose a service implementation class over networking protocols as a Web service. These annotations are available on J2SE platform. They are supported for both JAX-WS and JAX-RPC application. These annotations are present in javax.jws package.

- **@WebService Annotation:** This annotation marks a Java class as a Web service or a Java interface as a Web service interface. @WebService annotation is present in all endpoint Java implementation classes. The end point indicates a specific location where one can access a service using a specific protocol and data format. Table 5.4 shows the attributes of javax.jws.WebService annotation.

Attribute	Description
name	It specifies the name of the Web service. It is represented by wsdl:portType. Its default value is the name of Java class/interface.
targetNamespace	It specifies the XML namespace of the XML elements. Its default value is the namespace mapped to a package name of Web service.

Attribute	Description
serviceName	It specifies the service name of the Web service. It is mapped in the WSDL file using wsdl:service. Its default value is the Java class/ interface and service.
endpoint Interface	It helps to separate the service interface from the implementation class. If this property is not specified, then the Web service interface is generated by the implementation class.
wsdlLocation	It indicates the relative/absolute Web address of the WSDL file that defines the Web service.

Table 5.4: WebService Annotation – Attributes

These attributes are optional.

- **@WebMethod Annotation:** This annotation exposes a method as a Web service operation. This annotation can be used on all methods of classes that use @ WebService annotations. However, the rules defined in the JAX-RPC1.1 for return types have to be followed. The method need not throw exceptions.java.rmi. RemoteException exception. Table 5.5 shows the attributes of javax.jws.WebMethod annotation.

Attribute	Description
operationName	It maps the method to wsdl:operation element in the WSDL file. It defines the operation type performed by the Web service method. The default value is the name of the Java method as a string.
action	It defines the action for this operation. It finds out SOAP action header value for SOAP bindings. Its default value is a string.

Table 5.5: WebMethod Annotation – Attributes

These attributes are optional.

- **@WebParam Annotation:** This annotation defines the parameters of Web operation method of the service endpoint implementation class. It enables the mapping between the Web method parameters and XML message element in the SOAP. Table 5.6 lists the attributes of javax.jws.WebParam annotation.

Attribute	Description
name	It specifies the name of the parameter.
targetName space	It specifies the XML namespace for the parameter. This attribute has to be applied only when the operation is document-style or the parameter maps to a header. Its default value is the Web service's namespace.
mode	It shows the direction in which the parameters are passed and returned in the method. Its values can be IN, OUT, and INOUT. Its default value is IN.

Attribute	Description
header	It tells whether the parameter is present in the message header/ body. The default Boolean value is false, which shows that the parameter is present in message body.

Table 5.6: WebParam Annotation – Attributes

These attributes are optional.

- **@WebResult Annotation:** This annotation is used to customize the mapping of the return value to a WSDL part or XML element. Table 5.7 lists the attributes of javax.jws.WebResult annotation.

Attribute	Description
name	It specifies the variable name that is returned after invoking the Web service method. In document style operations, the name parameter is the local name of XML element, which is the return value.
targetNamespace	It specifies the XML namespace for the value returned. When the return value maps to an XML element, this is used to bind documents. The default value of targetNamespace is empty namespace, represented as “”.

Table 5.7: WebResult Annotation – Attributes

These attributes are optional.

- **@SOAPBinding Annotation:** This annotation is in the javax.jws.soap package. It is used to map the Web service to the SOAP message protocol. Table 5.8 lists the attributes of javax.jws.soap.SOAPBinding annotation.

Attribute	Description
style	It defines the message encoding style of Web services. Its value can either be DOCUMENT or RPC. Its default value is javax.jws.soap.SOAPBinding.Style.DOCUMENT.
use	It defines the message formatting style in Web services. Its default value is LITERAL (javax.jws.soap.SOAPBinding.Use.LITERAL.
parameterStyle	It determines the way the method parameters are placed in the message body. Its default value can be BARE or WRAPPED. If the value is set to BARE, it implies that each parameter is placed as a child element in the message body. If the value is set to WRAPPED, it means that all input/output parameters are in a single element in the respective request/response message. Its default value is javax.jws.soap.SOAPBinding.ParameterStyle.WRAPPED.

Table 5.8: SOAPBinding Annotation – Attributes

These attributes are optional.

- **@SOAPMessageHandler Annotation:** This annotation is used to specify a SOAP message handler in a SOAPMessageHandler array. Table 5.9 lists the attributes of javax.jws.soap.SOAPMessageHandler annotation.

Attribute	Description
name	Specifies the name of SOAP message handler. The default value of this attribute is the name of the class that implements the Handler interface.
className	It defines the message formatting style in Web services. Its default Specifies name of the Handler class.
initParams	Specifies the array of the name/value pairs that will be passed to the handler during initialization.
roles	Specifies the list of SOAP roles that the handler implements.
headers	Specifies the list of headers that the handler processes.

Table 5.9: SOAPMessageHandler Annotation – Attributes

These attributes, except className, are optional.

- **@initParams Annotation:** This annotation is used to specify the array of name/value pairs that are passed to the handler during initialization. It is used as a value for the initParams attribute of the SOAPMessageHandler annotation. Table 5.10 lists the attributes of javax.jws.soap.initParams annotation.

Attribute	Description
name	Specifies the name of the initialization parameter.
value	Specifies the value of the initialization parameter.

Table 5.10: initParams Annotation – Attributes

These attributes must be specified when using the initParams annotation.

5.4.2 Core JAX-WS API Annotations

There are a few core annotations of JAX-WS API that specify the metadata associated with Web service implementations. At runtime, these annotations simplify the process of developing Web services. They help to map Java to WSDL and schema. They also respond to Web service invocations and help to control the JAX-WS runtime processes.

- **@RequestWrapper Annotation:** This annotation is used to annotate the methods of the Service Endpoint Implementation (SEI) class. It is wrapped in a request wrapper bean generated at the run time. For serialization and deserialization, it provides the element name and namespace to the JAXB generated request wrapper bean. This information is used by the request wrapper bean at runtime. Table 5.11 lists the attributes of javax.xml.ws.RequestWrapper annotation.

Attribute	Description
localName	It describes the local name of the wrapper element in the message request. It is in the form of XML. Its default value is the name of the method annotated with @WebMethod.
targetNamespace	It indicates the namespace of the request wrapper element. Its default value is the targetNamespace of the SEI.
className	It indicates the name of the request wrapper class.

Table 5.11: RequestWrapper Annotation – Attributes

- **@ResponseWrapper Annotation:** This annotation is used to annotate the methods of the SE. These methods are wrapped in a response wrapper bean generated at the run time. Table 5.12 shows the attributes of javax.xml.ws.ResponseWrapper annotation.

Attribute	Description
localName	It describes the wrapper element's local name in the XML message response. Its default value is the method name annotated with @WebMethod.
targetNamespace	It indicates the namespace of the request wrapper element. Its default value is the targetNamespace attribute value of the SEI.
className	It indicates the name of the response wrapper class.

Table 5.12: ResponseWrapper Annotation – Attributes

- **@WebServiceProvider Annotation:** This is also called as @WebService. It can be used to annotate Java implementation class. The @WebServiceProvider annotation describes the class as a JAX-WS Provider implementation class which supports a message-oriented approach and generates the provider endpoints. A class with the @WebServiceProvider annotation must override the invoke() method.

Table 5.13 lists the attributes of javax.xml.ws.WebServiceProvider annotation.

Attribute	Description
wsdlLocation	It is a mandatory attribute. It defines the Web location of the WSDL file that describes the Web service.
portName	It defines the service end point and maps to the wsdl:port element of WSDL file.
serviceName	It defines the service endpoint and maps to the wsdl:service element of WSDL file. Its default value is the unqualified name of the Java class, which is appended with the service keyword.
targetNamespace	It specifies the WSDL generated from Web service and the XML namespace of the XML elements. Its default value is the namespace, which is mapped to a package name of the Web service.

Table 5.13: WebServiceProvider Annotation – Attributes

- **@ServiceMode Annotation:** This annotation is used to develop JAX-WS Web services implementation class with the help of the Web service provider. It specifies whether the service provider can access the entire message or the message body only. The attribute of javax.xml.ws.ServiceMode annotation is value. This attribute indicates the amount of message to be considered. This property is used by the provider class to access that part of message that is marked as PAYLOAD. It can also access the entire message if it is marked as MESSAGE. Its default value is a string representing PAYLOAD.
- **@WebServiceRef Annotation:** This annotation is used at the client-side. It specifies the Web service that has to be invoked and can be applied at field level, method level, and class level. Table 5.14 lists the attributes of javax.xml.ws.WebServiceRef annotation.

Attribute	Description
name	It specifies the JNDI name of the resource. It maps the annotated method to its related Java bean property name. It maps to the default field name in case of field annotation.
type	It specifies the resource of Java type. It maps the annotated method type to its related Java bean property type. It maps to the default field name in case of annotated field.
mappedName	It specifies that a resource is mapped to a product specific name.
value	It specifies that javax.xml.ws.Service is extended by the service class. Its reference is of SEI.
wsdlLocation	It defines the location of the WSDL file for the Web service invocation.

Table 5.14: WebServiceRef Annotation – Attributes

5.5 Design Requirements of a JAX-WS Application

The two types of service endpoint implementations supported by JAX-WS, which make services to work at XML message level are as follows:

- The standard JavaBeans service endpoint interface
- A new Provider interface

A JAX-WS Web service can be developed by using either javax.jws.WebService or JAX-WS javax.jws.WebServiceProvider annotation. @WebService may be used for JavaBeans endpoints and @WebServiceProvider may be used for provider endpoints.

5.5.1 Requirements of Service Endpoints in JAX-WS

Table 5.15 lists the requirements for JAX-WS service endpoints as Dos and Don'ts.

Dos	Don'ts
1. Use javax.jws.WebService or javax.jws.WebServiceProvider annotation to annotate the Web service implementation class. 2. Declare the business methods as public. 3. Use javax.jws.WebMethod annotation to declare the business methods. 4. javax.jws.WebMethod annotation should have JAXB compatible parameters and return types. This has to be done for business methods that are exposed to Web service clients. 5. The life cycle event callback methods can be contained in implementing class. The methods include javax.annotation.PostConstruct or javax.annotation.PreDestroy.	1. Do not mention the endpoint interface element explicitly in @WebService annotation as the SEI is implicitly defined. An implementation class can reference the SEI. 2. Do not declare business methods as static or final in the implementation class. 3. Do not declare the implementing class as abstract or final. 4. Do not have finalize() method in the implementing class.

Table 5.15: Requirements for JAX-WS Service Endpoints

5.5.2 JAX-WS Client Model

JAX-WS supports different types of clients in Web service client programming model. Following are the types of clients in Web Service programming model:

- **Dynamic proxy client:** It is also called as static stub client model in JAX-WS. SEI is used to invoke Web services. Dynamic Proxy Client is used to implement Web services. In JAX-RPC, tools are used to generate stub client. In JAX-WS Java 5 Dynamic Proxy generation functionality is used to generate stubs dynamically at run time. In the Java Runtime Environment (JVM), the proxy instances extend from java.lang.reflect.Proxy class.
- **Dispatch client:** It is also called as dynamic dispatch client model in JAX-WS. This model is flexible to write XML constructs and is generic. The service endpoints are dynamically invoked by the dynamic client API javax.xml.ws.Dispatch. This client writes XML representations of the message to support service invocation.

Data can be sent by the dispatch client API to construct XML message in any of the following modes:

- **PAYLOAD mode:** In the javax.xml.ws.Service.Mode.PAYLOAD mode, the dispatch client provides the content of the soap:Body element. The JAX-WS includes the payload in the soap:Envelope element.

- **MESSAGE mode:** In the javax.xml.ws.Service.Mode.MESSAGE mode, the dispatch client provides the entire SOAP envelope. It includes the soap:Envelope, soap:Header, and soap:Body elements.

Table 5.16 lists the methods of javax.xml.ws.Dispatch interface.

Attribute	Description
T invoke(T msg)	It specifies synchronous invocation of Web service operation. It has a parameter msg. This parameter represents a message object that can have the entire message or payload section of the message. The msg object is marshalled as per the underlying protocol binding.
Response<T> invokeAsync(T msg)	It specifies asynchronous invocation of the Web service operation. It returns to the client without waiting for the response from the Web service method. The response is received by polling.
java.util.concurrent.Future<?> invokeAsync (Tmsg,AsyncHandler<T> handler)	It specifies asynchronous invocation of the Web service operation. It returns to the client without waiting for the response from the Web service method. The response is received by the handler object.

Table 5.16: javax.xml.ws.Dispatch Interface Methods

5.6 Creating a Simple Web Service Application Using JAX-WS

A Web service application can be developed using JAX-WS by annotating the Java class with the @WebService annotation. By annotating, the class can be defined as a Web service endpoint. The files required to develop a Web service application using JAX-WS implementation are:

- Service Interface
- Service Implementation Class

Java interface or Java class is also called as SEI. It has the methods defined in service class which that can be invoked by the client. The service class can be built by JAX-WS.

If JAX-RPC is used, then each Web Service implementation class has to implement the service interface exposed to the client. However, if JAX-WS is used, there is no need to have service implementation class to implement service interface. The SEI is built by JAX-WS runtime. Consider a scenario where employee details, such as employee ID, employee name, and employee salary are stored in a database table named Employee, as shown in figure 5.5.

select * from APP.EMPLOYE... ×

Page Size: 20 Total Rows: 4 Page: 1 of 1

#	EMPID	EMPLOYEENAME	EMPLOYEESALARY
1	EMP111	John Smith	8000.0
2	EMP112	Michelle Sawyer	8500.0
3	EMP113	Brenda Taylor	9000.0
4	EMP114	Alex Johnson	9500.0

Figure 5.5: Employee Table

A Web service needs to be created to retrieve the employee name and employee salary based on the specified employee ID.

Following are the steps to create a Web service application and client:

1. Create the Web service implementation class.
2. Build, package (war files), and deploy the files. The Glassfish server automatically generates the artifacts required for communicating with the clients.
3. Create the client class.
4. Build and execute the client application.

5.6.1 Create Web Service Implementation Class

The Web service implementation class contains the method definition or implementation code for the method, which is accessed by the client.

To create the Web service implementation class, perform the following steps:

In the NetBeans IDE, click File → New Project → Others and then in the New Project wizard, select Java EE from the Categories list and Enterprise Application from the Projects list, as shown in figure 5.6.

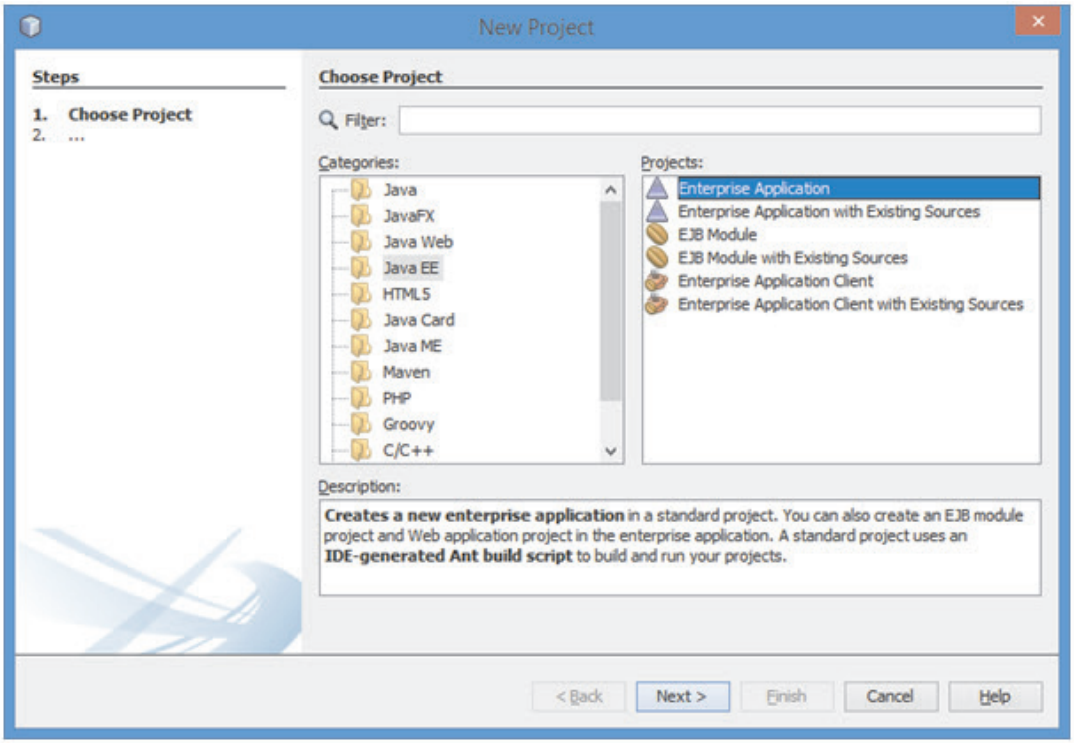


Figure 5.6: Creating an Enterprise Application

Specify a name for the Enterprise Application, as shown in figure 5.7.

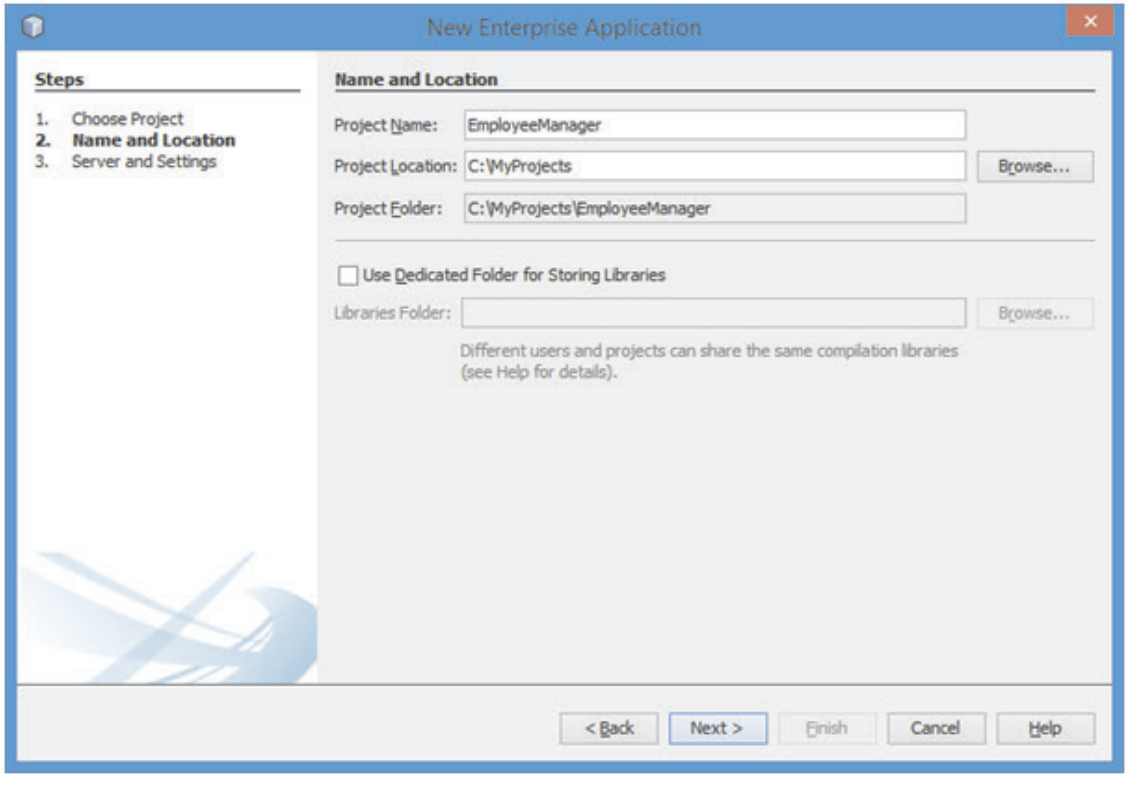


Figure 5.7: Specifying a Name for the Enterprise Application

Ensure that the Glassfish Server is selected in the Server list and the Context Path is specified, as shown in figure 5.8. The Web service will be deployed to the Glassfish Server.

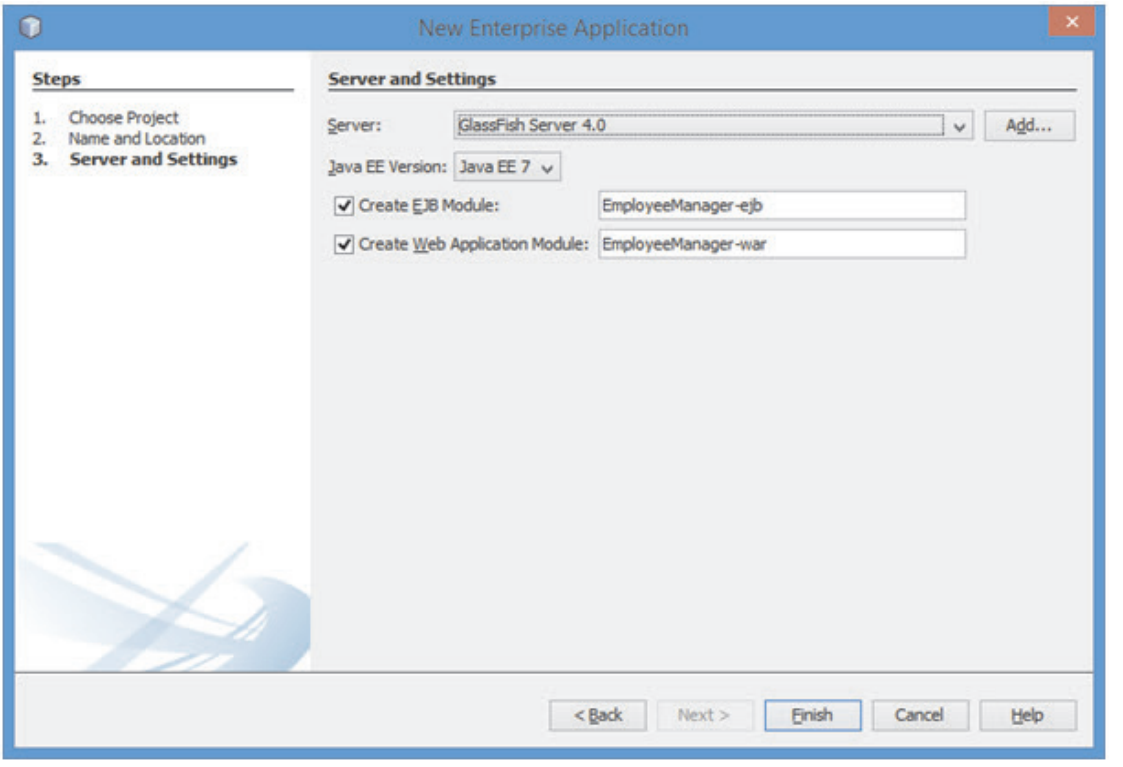


Figure 5.8: Selecting Glassfish Server and Setting Context Path

To use entity beans to create the Web service, entity classes need to be created from the Employee database. To do this, right-click EmployeeManager-ejb and then click New → Other. Then, in the New File wizard, select Persistence in the Categories list and Entity Classes from Database in the File Types list, as shown in figure 5.9.

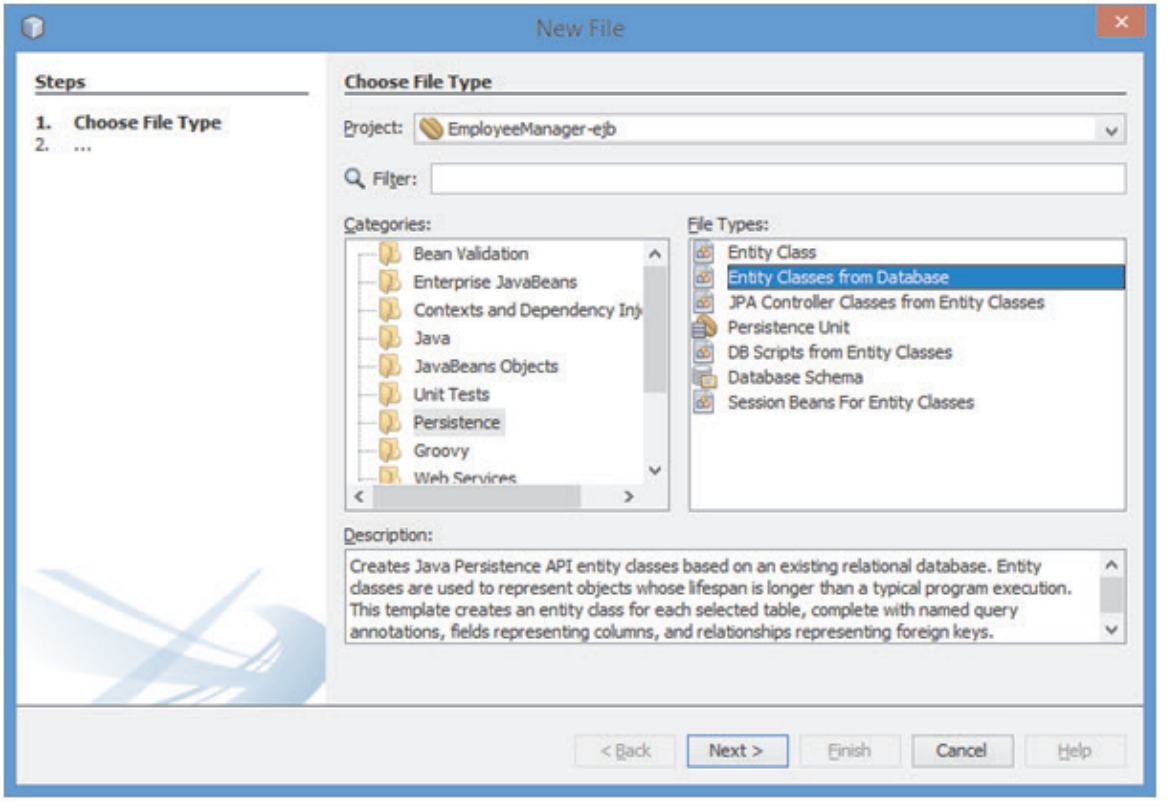


Figure 5.9: Creating Entity Classes from Database

Next, to select the database to use, in the Database Source drop-down list, select jdbc/sample. All the tables associated with the selected source will be displayed in the Available Tables list. Select EMPLOYEE and then click the Add button to add the table to the Selected Tables list, as shown in figure 5.10.

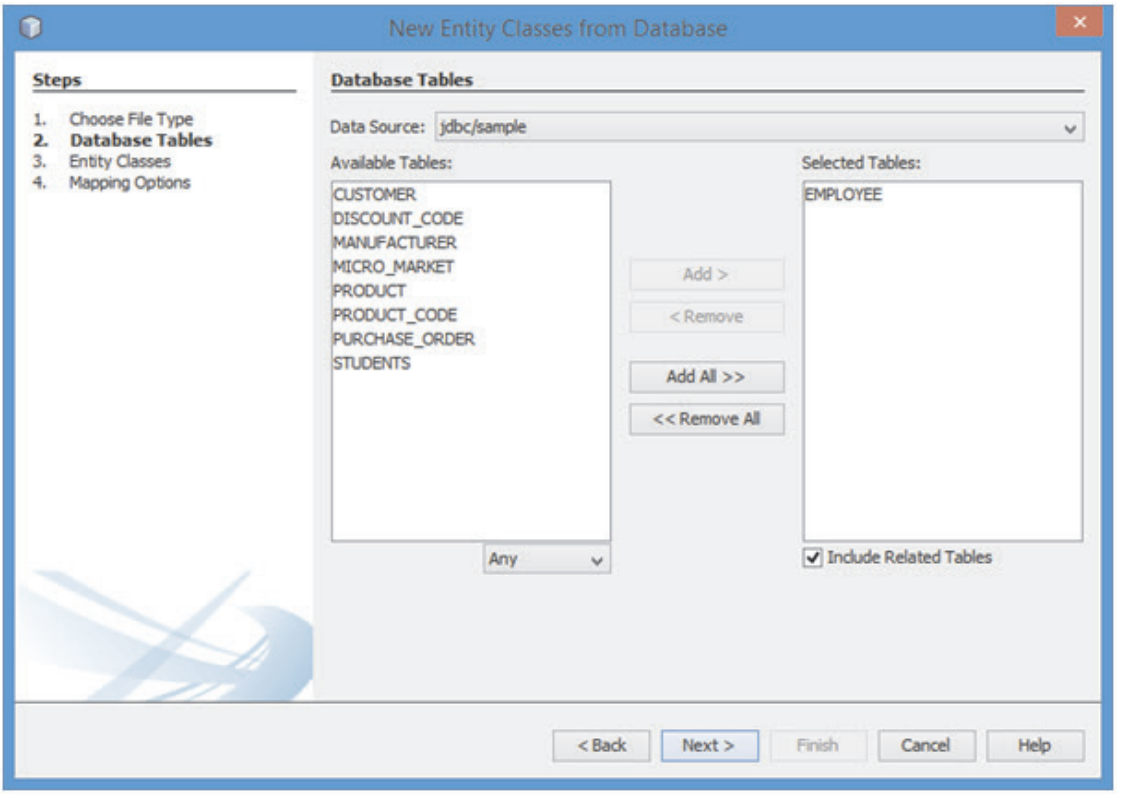


Figure 5.10: Selecting Database for Creating Entity Classes

On the next page of the wizard, specify a package for the entity classes and then click Finish, as shown in figure 5.11.

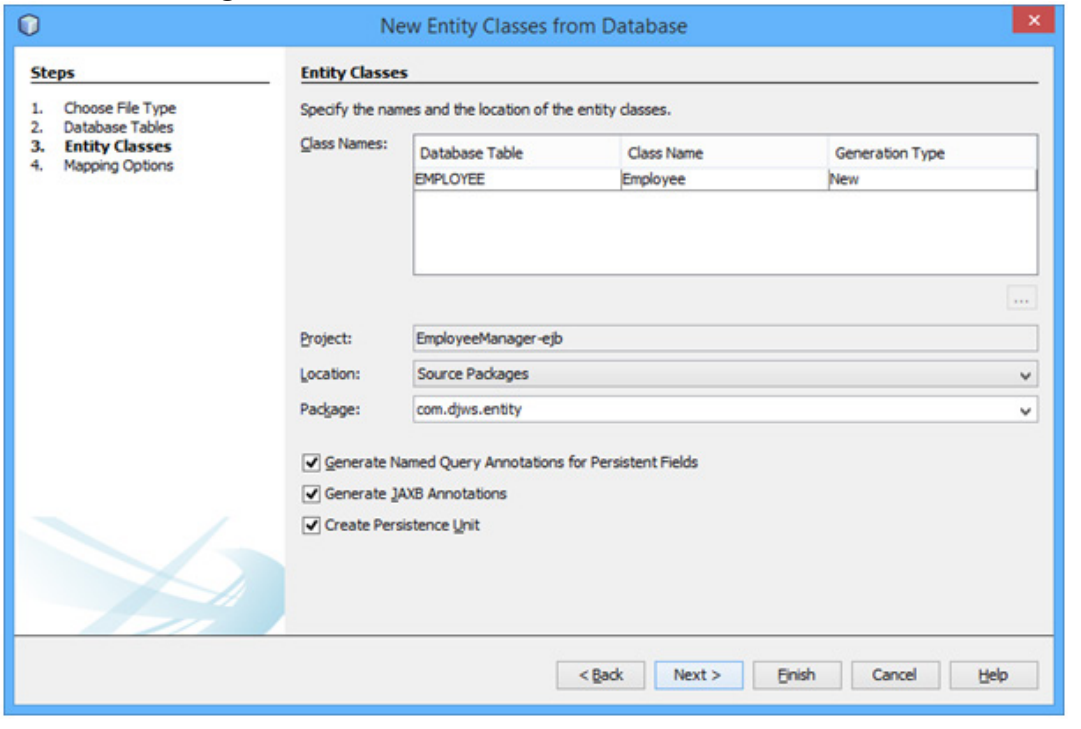


Figure 5.11 Specifying Package for Entity Classes

To create session beans from the entity classes, right-click EmployeeManager-ejb and then click New → Others. Then, in the New File wizard, select Enterprise JavaBeans from the Categories list and Session Beans for Entity Classes from the File Types list, as shown in figure 5.12.

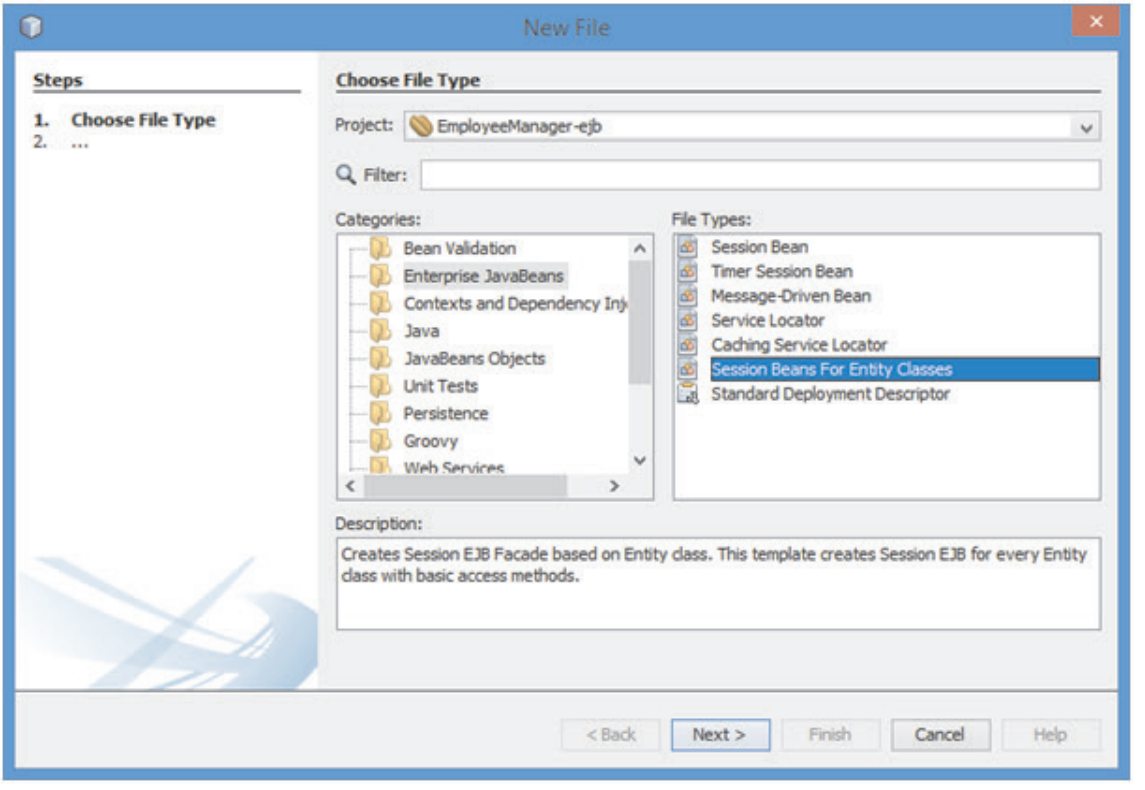


Figure 5.12: Creating Session Beans for Entity Classes

On the next page of the wizard, select the required entity classes from the Available Entity Classes list and then, click the Add button, as shown in figure 5.13.

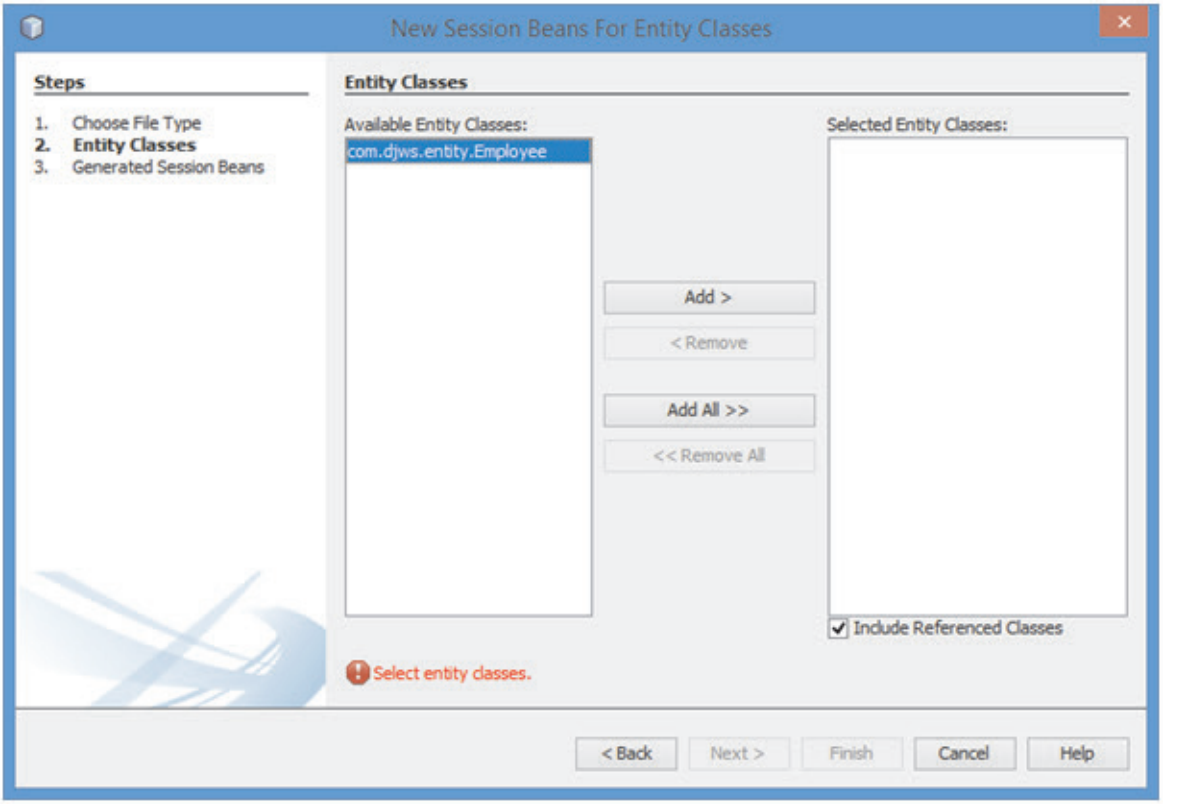


Figure 5.13: Selecting Entity Classes to Create Session Beans

Next, specify a package name for the session beans and select Local to create local interfaces, as shown in figure 5.14.

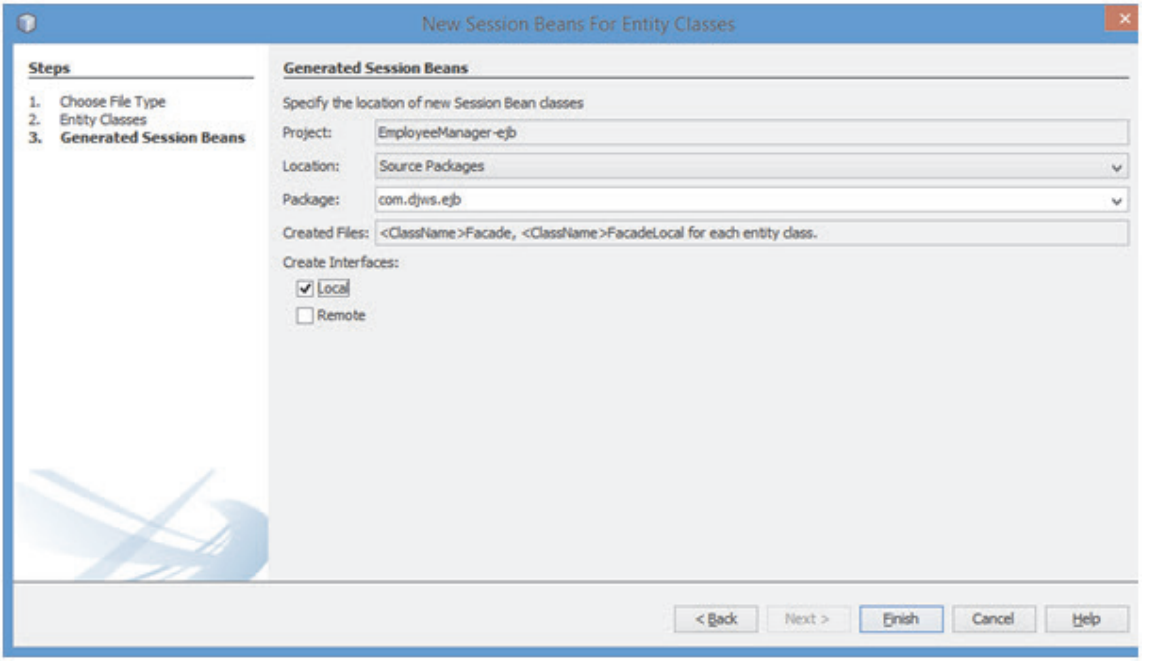


Figure 5.14: Specifying Package Name and Interface Type

Three Java classes will be created. In this example, the three Java classes are AbstractFacade, EmployeeFacade, and EmployeeFacadeLocal, as shown in figure 5.15.

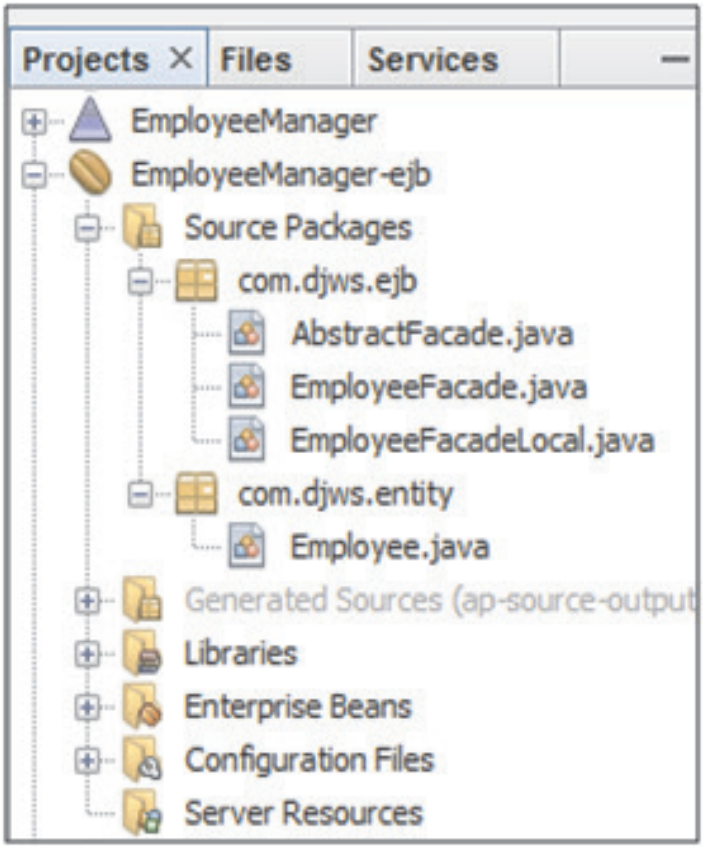


Figure 5.15: Java Classes

To create a Web Service from the sessions beans, right-click EmployeeManager-war, and then click New → Other. Then, in the New File wizard, select Web Services in the Categories list and Web Service in the File Types list, as shown in figure 5.16.

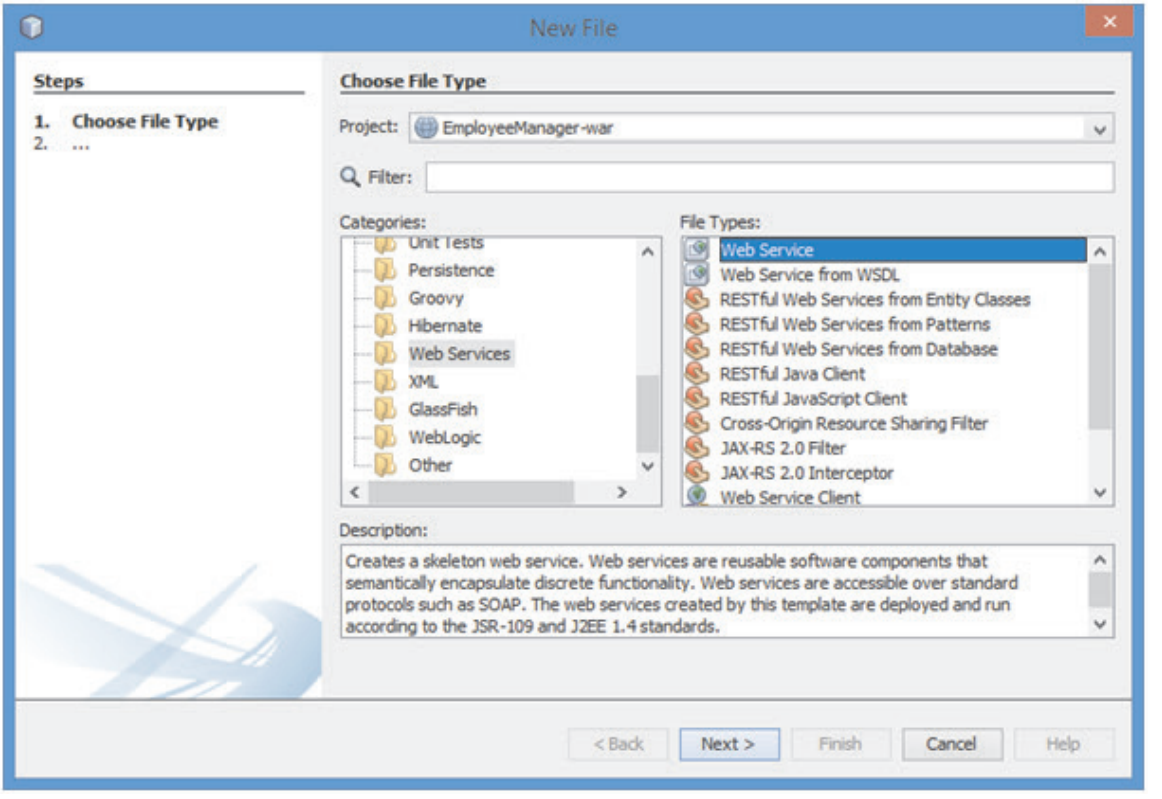


Figure 5.16: Creating a Web Service

Specify a name for the Web service, a package name for the Web service, select the Create Web Service from Existing Session Bean option, and then, click Browse. In the Browse Enterprise Bean dialog box, select the bean to be used, as shown in figure 5.17.

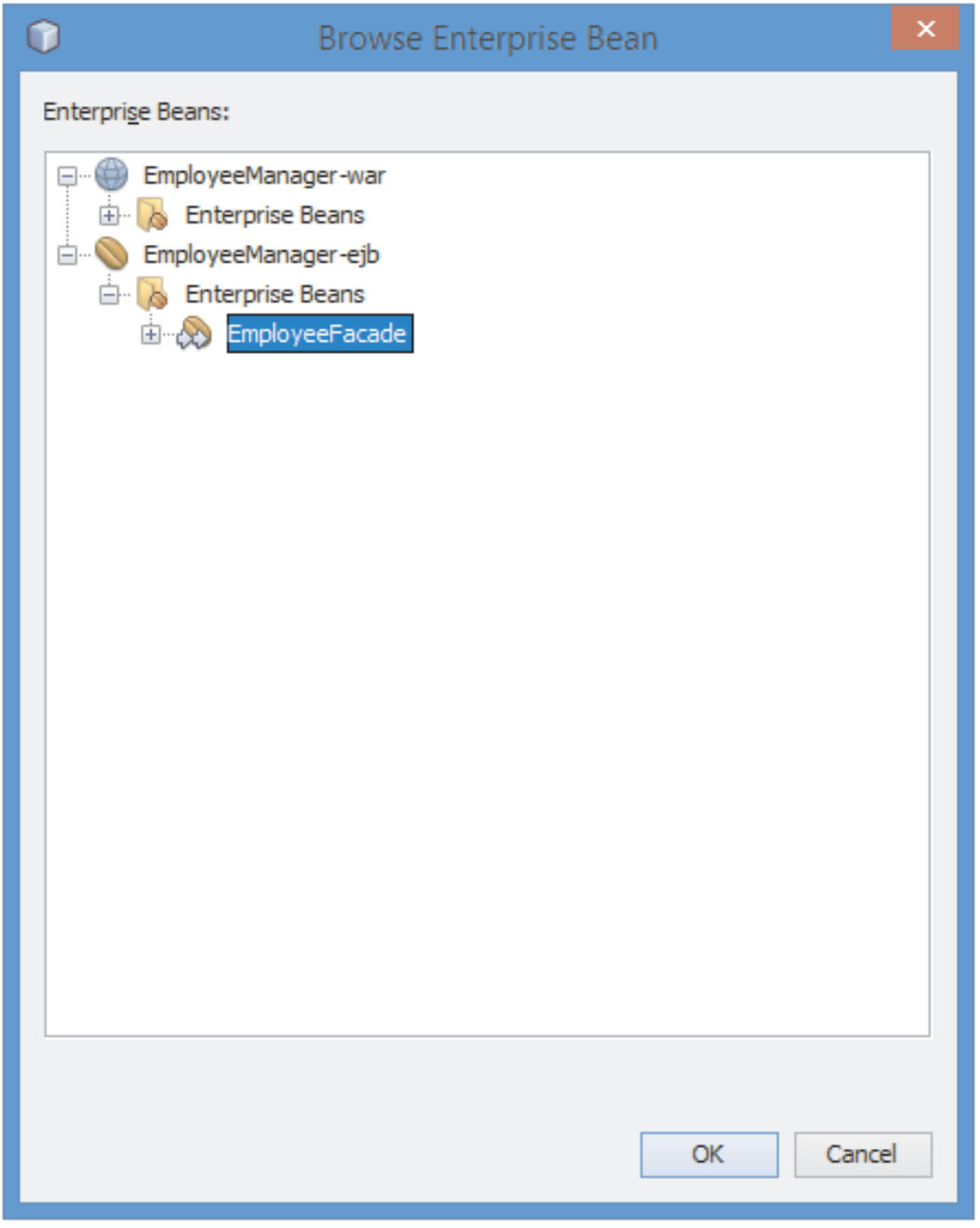


Figure 5.17: Selecting Session Bean for Creating Web Service

Select the Implement Web Service as Stateless Session Bean check box, as shown in figure 5.18.

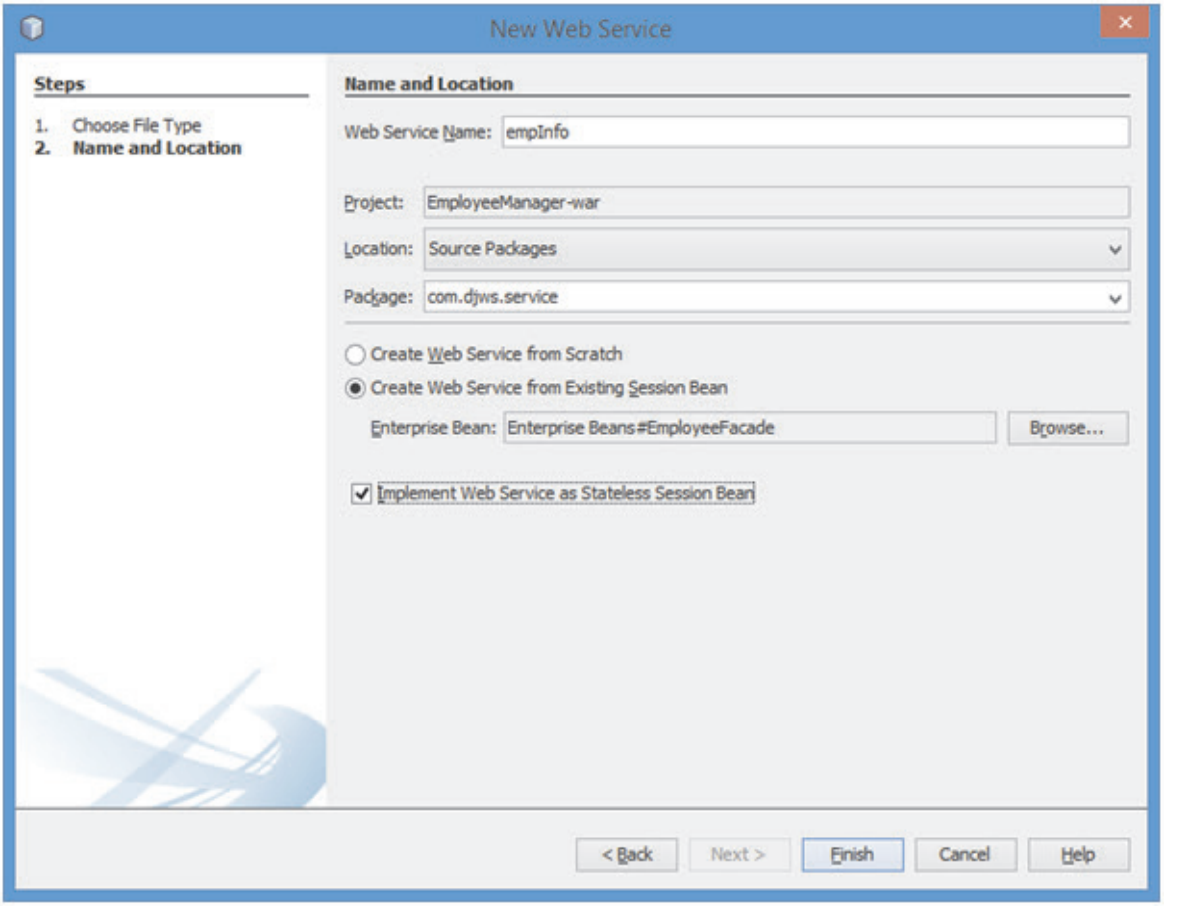


Figure 5.18: Selecting the Required Option

Alternatively, you can create a session bean from scratch. By default, the Web service is implemented as a stateful session bean. To create a stateless session bean, you need specify that explicitly by selecting the check box provided. The .java file for the Web service is created.

By default, this Web service contains code for several operations, such as create, edit, remove, find, findAll, findRange, and count, which can be viewed by clicking the Design tab as shown in figure 5.19.

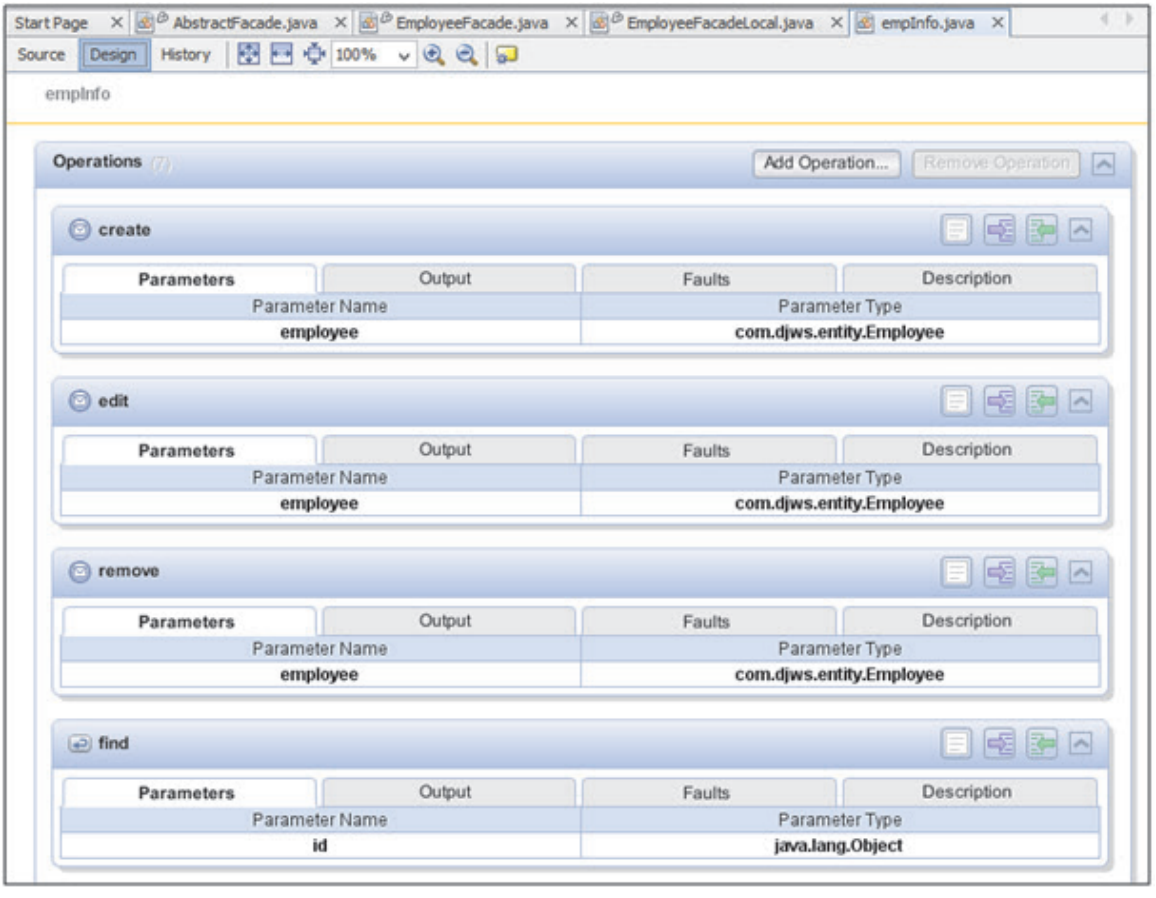


Figure 5.19: Web Service Operations

To remove some of these default operations, in the Design tab, click the operation name bar to select the operation and click Remove Operation button. Retain the count, find, and findAll operations as shown in figure 5.20.

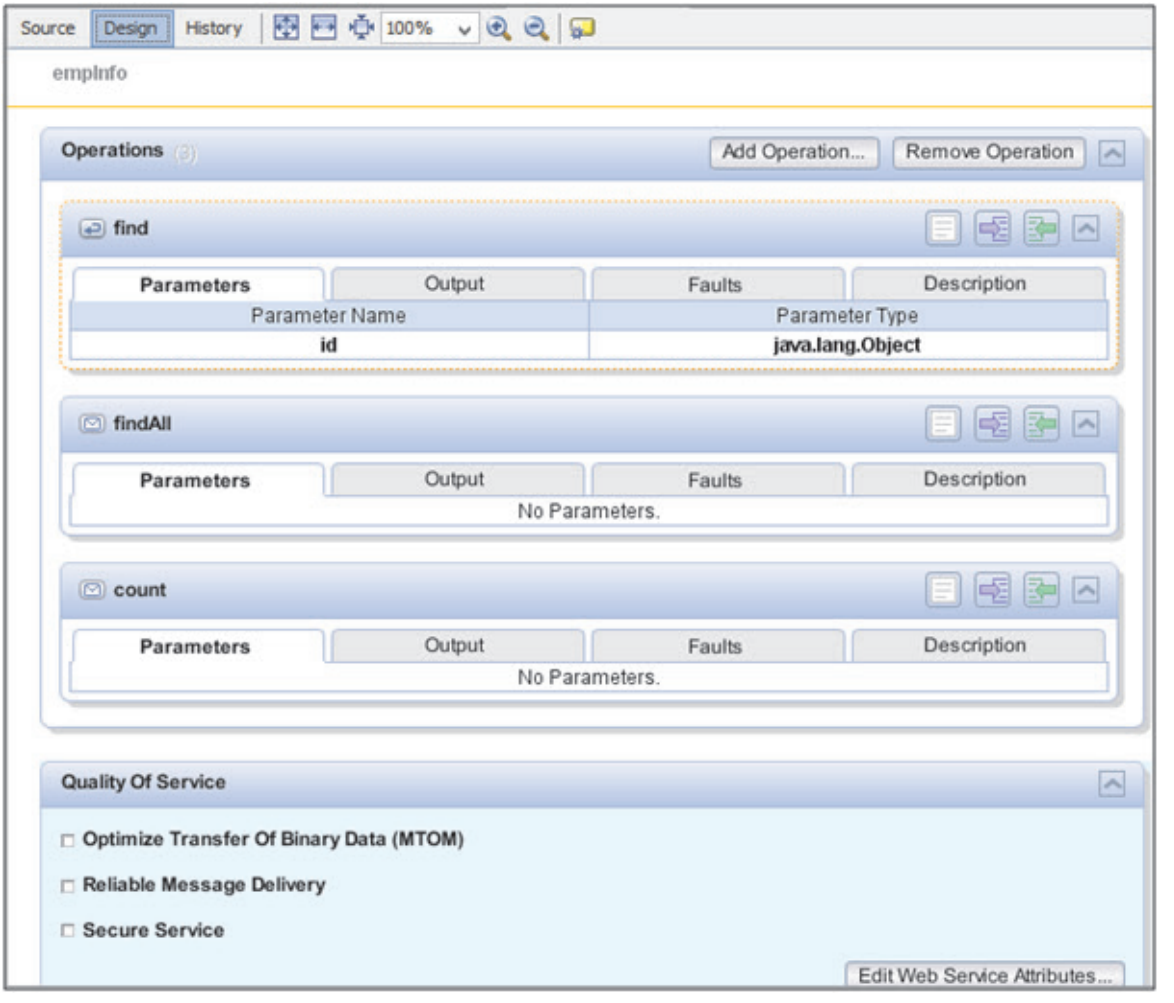


Figure 5.20: Retained Web Service Operations

5.6.2 Build, Package, and Deploy the Web Service

The generation and deployment of the Web service is done in NetBeans IDE. The steps to do are as follows:

1. Select the project in NetBeans IDE.
2. Right-click to view the context menu.
3. Select Deploy from the context menu.

The command Deploy builds and packages the Web application into the WAR file. Then, it deploys the file on the application server, that is, GlassFish Server. To test the Web Service, right-click the Web Service and then, in the context menu, click Test Web Service as shown in figure 5.21.

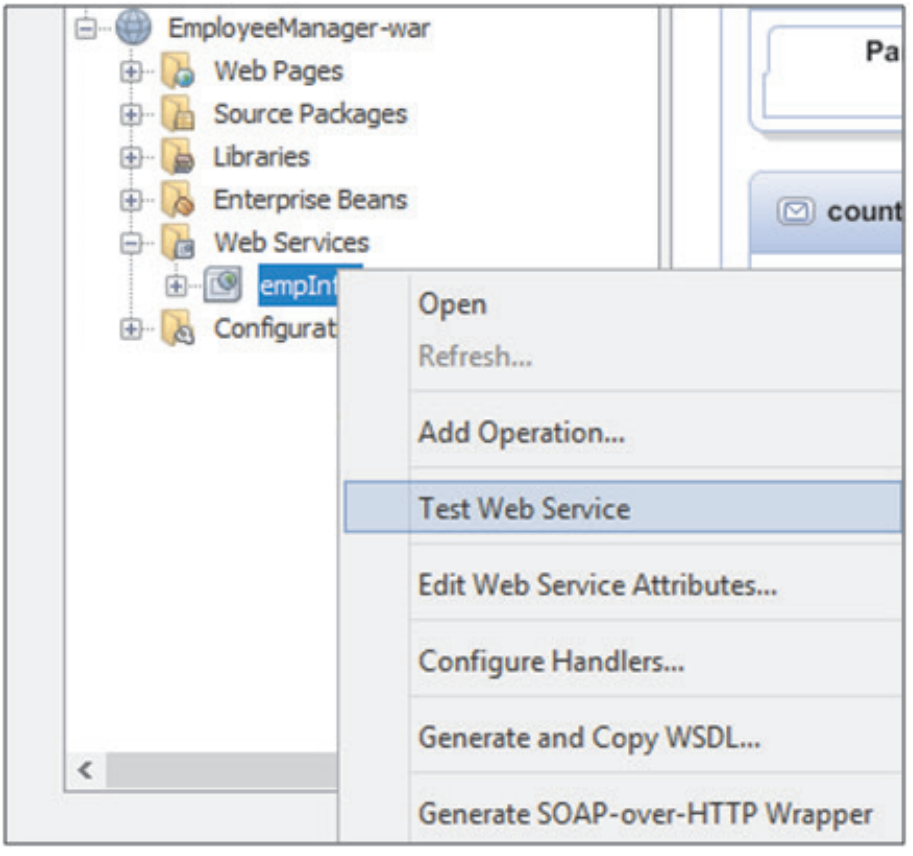


Figure 5.21: Testing the Web Service

The tester Web page appears as shown in figure 5.22.

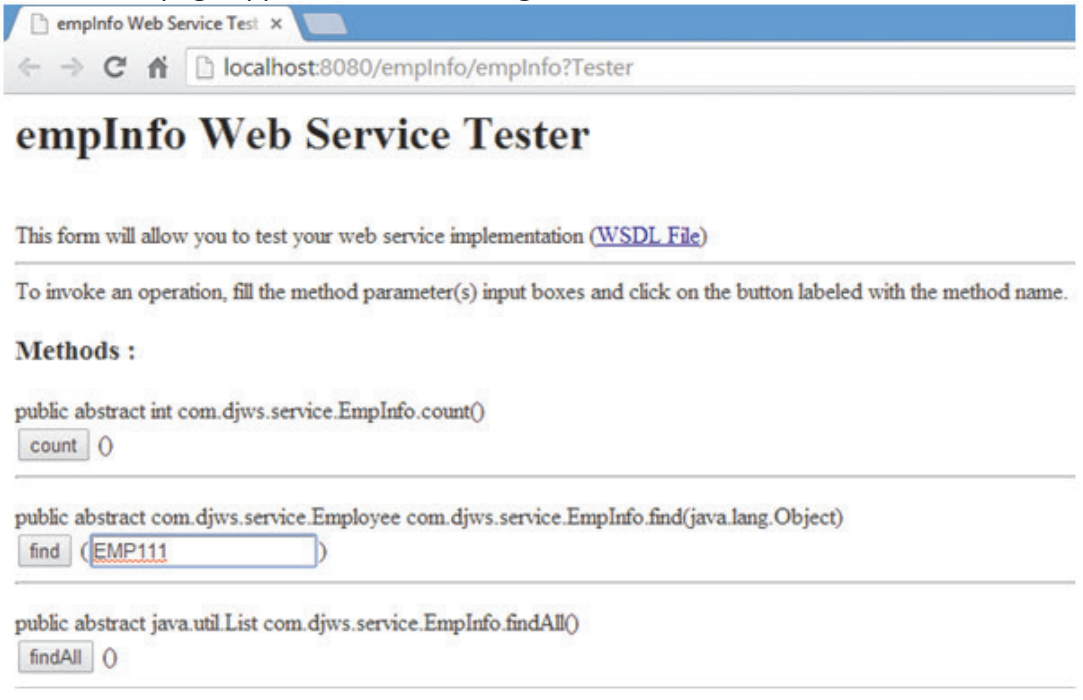


Figure 5.22: Web Service Tester

The Web Service Tester displays the operations that were retained—count, find, and findAll. The count operation will display the number of records in the Employee table. The find operation will display the details of the employee whose employee ID has been specified in the box. The findAll operation will display the details of all employees stored in the Employee table. Figure 5.23 shows the output of the findAll operation.

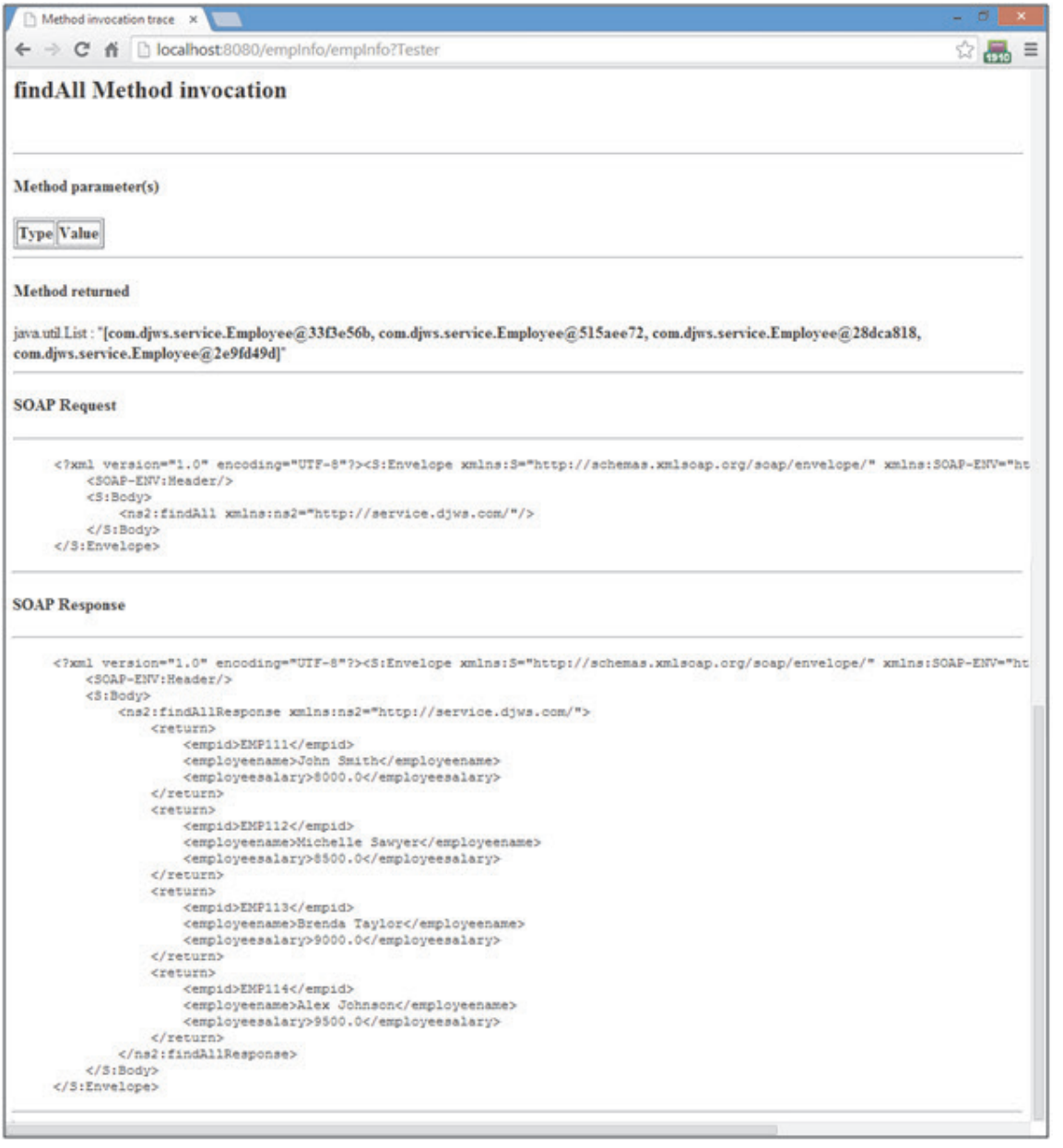


Figure 5.23: Output of findAll Operation

5.6.3 Create a Web Client

A Web client can be created using JSP, Servlet, and so on. The WSDL file is generated based on the Web service. Code Snippet 1 shows the WSDL generated by the Web service.

Code Snippet 1:

```
This XML file does not appear to have any style information associated
with it. The document tree is as follows:

<!--

Published by JAX-WS RI at http://jax-ws.dev.java.net. RI's version
is

Metro/2.3 (tags/2.3-7528; 2013-04-29T19:34:10+0000) JAXWS-RI/2.2.8

JAXWS/2.2 svn-revision#unknown.

-->

<!--

Generated by JAX-WS RI at http://jax-ws.dev.java.net. RI's version
is  Metro/2.3  (tags/2.3-7528;  2013-04-29T19:34:10+0000)  JAXWS-
RI/2.2.8 JAXWS/2.2 svn-revision#unknown.

-->

<definitions xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-utility-1.0.xsd"

xmlns:wsp="http://www.w3.org/ns/ws-policy"

xmlns:wsp1_2="http://schemas.xmlsoap.org/ws/2004/09/policy"

xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"

xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"

xmlns:tns="http://service.djws.com/"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns="http://schemas.xmlsoap.org/wsdl/"

targetNamespace="http://service.djws.com/" name="empInfo">

<types>
```

```
<xsd:schema>
<xsd:import namespace="http://service.djws.com/"
schemaLocation="http://localhost:8080/empInfo/
empInfo?xsd=1"/>
</xsd:schema>
</types>
<message name="count">
<part name="parameters" element="tns:count"/>
</message>
<message name="countResponse">
<part name="parameters" element="tns:countResponse"/>
</message>
<message name="find">
<part name="parameters" element="tns:find"/>
</message>
<message name="findResponse">
<part name="parameters" element="tns:findResponse"/>
</message>
<message name="findAll">
<part name="parameters" element="tns:findAll"/>
</message>
<message name="findAllResponse">
<part name="parameters" element="tns:findAllResponse"/>
</message>
<portType name="empInfo">
<operation name="count">
<input wsam:Action="http://service.djws.com/empInfo/countRequest"
message="tns:count"/>
<output wsam:Action="http://service.djws.com/empInfo/
countResponse" message="tns:countResponse"/>
```

```
</operation>
<operation name="find">
<input   wsam:Action="http://service.djws.com/empInfo/findRequest"
message="tns:find"/>
<output wsam:Action="http://service.djws.com/empInfo/findResponse"
message="tns:findResponse"/>
</operation>
<operation name="findAll">
<input wsam:Action="http://service.djws.com/empInfo/
findAllRequest" message="tns:findAll"/>
<output wsam:Action="http://service.djws.com/empInfo/
findResponse" message="tns:findResponse"/>
</operation>
<operation name="findAll">
<input wsam:Action="http://service.djws.com/empInfo/
findAllRequest" message="tns:findAll"/>
<output wsam:Action="http://service.djws.com/
empInfo/findAllResponse"
message="tns:findAllResponse"/>
</operation>
</portType>
<binding name="empInfoPortBinding" type="tns:empInfo">
<soap:binding transport="http://schemas.xmlsoap.org/soap/http"
style="document"/>
<operation name="count">
<soap:operation soapAction=""/>
<input>
<soap:body use="literal"/>
</input>
<output>
```

```
<soap:body use=”literal”/>
</output>
</operation>
<operation name=”find”>
<soap:operation soapAction=””/>
<input>
<soap:body use=”literal”/>
</input>
<output>
<soap:body use=”literal”/>
</output>
</operation>
<operation name=”findAll”>
<soap:operation soapAction=””/>
<input><soap:body use=”literal”/></input>
<output>
<soap:body use=”literal”/>
</output>
</operation>
</binding>
<service name=”empInfo”>
<port name=”empInfoPort” binding=”tns:empInfoPortBinding”>
<soap:address location=”http://localhost:8080/empInfo/empInfo”/>
</port>
</service>
</definitions>
```

The WSDL can also be copied to the project. To do this, right-click the Web service and then click Generate and Copy WSDL as shown in figure 5.24.

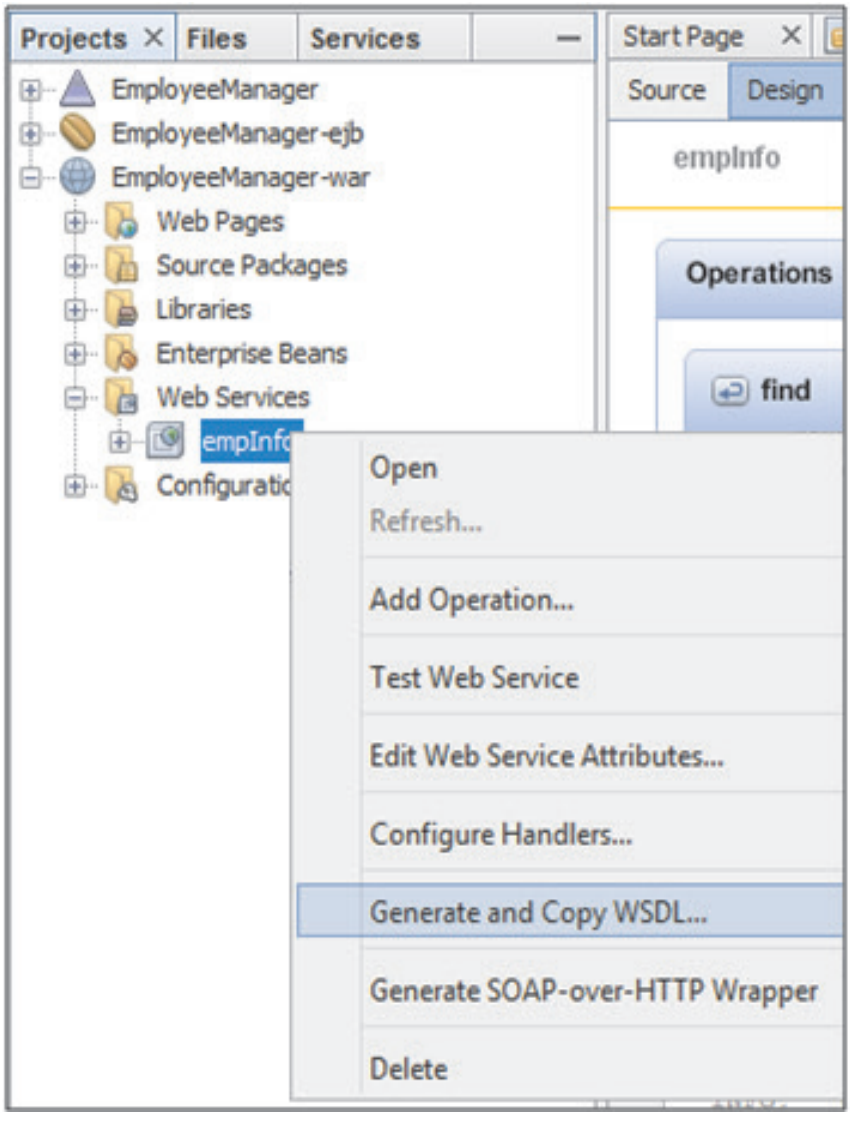


Figure 5.24: Generating and Copying WSDL to Project

A new folder is created in the project named Generated Sources (jax-ws). To view the WSDL file, expand Generated Sources (jax-ws) and then expand resources as shown in figure 5.25.

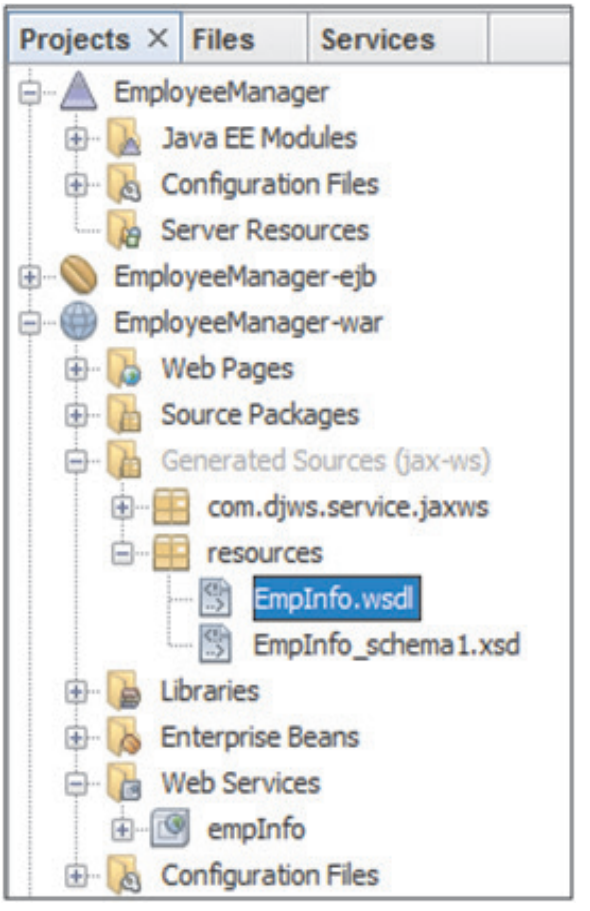


Figure 5.25: WSDL File Location

The WSDL file is used to generate the Web service client. Perform the following steps to generate the Web service client:

1. Create a new Web Application project in NetBeans IDE.
2. Ensure that the Glassfish Server is selected and the Context Path is set.
3. Right-click the project and then click New → Others and then in the New File Wizard, select Web Services in the Categories list and Web Service Client in the File Types list.
4. To specify the project where the WSDL file is located, click Browse next to the Project box.
5. Browse to the Web service for which the Web service client is being created, as shown in figure 5.26.

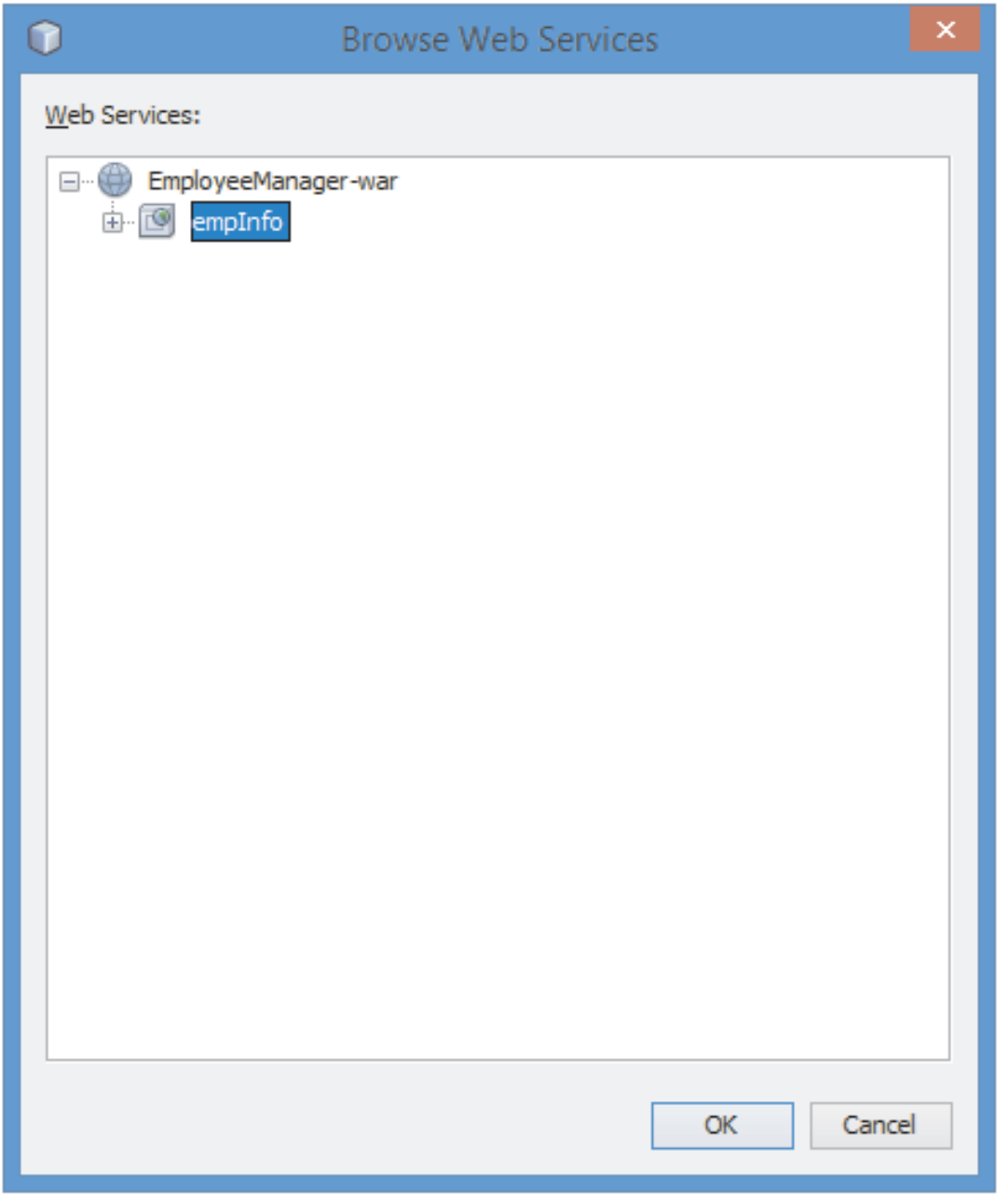


Figure 5.26: Selecting the Web Service

A Web Service References folder is created in the project. This folder contains the method exposed by the Web service, as shown in figure 5.27.

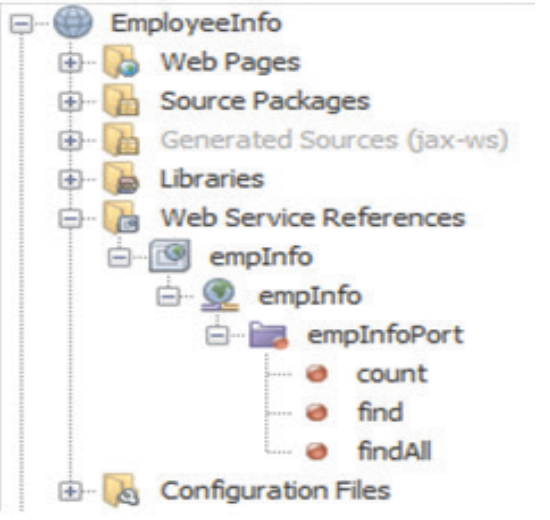


Figure 5.27: Method Exposed by Web Service

6. Add a new JSP file by right-clicking the project and then clicking New → JSP.

7. Name the JSP file as index.

Drag the find() method from the Web Service Reference folder, as shown in figure 5.25, to the index.jsp file and place it after the h1 heading. The updated index.jsp file appears as shown in figure 5.28.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>

    </head>
    <body>
        <h1>Hello World</h1>    <%-- start web service invocation --%><hr/>
    <%
    try {
        com.djws.service.EmpInfo_Service service = new com.djws.service.EmpInfo_Service();
        com.djws.service.EmpInfo port = service.getEmpInfoPort();
         // TODO initialize WS operation arguments here
                java.lang.Object id = null;
        // TODO process result here
        com.djws.service.Employee result = port.find(id);
    } catch (Exception ex) {
        // TODO handle custom exceptions here
    }
    %>
    <%-- end web service invocation --%><hr/>

    </body>
</html>
```

Figure 5.28: Updated index.jsp File

Update the title of the JSP page and h1 heading, add an input text box for entering the employee ID and a button, add the code to use the Web service to retrieve and display the relevant employee details and add the exception handling code in the catch block as shown in Code Snippet 2.

Code Snippet 2:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
        <title>JSP Page</title>
     <link href="style.css" rel="stylesheet" type="text/css"/>
    </head>
    <body>
        <h1>Employee Information</h1>
<%-- start Web service invocation --%><hr/>
        <form action="index.jsp">
        Enter Employee ID:
        <input type="text" id="txtbox" name="empid" size="6"
maxlength="6">
        <br></br>
        <input type="submit" value="Get Details">
        <br></br>
<%
    try {
      com.djws.service.EmpInfo_Service service = new
com.djws.service.EmpInfo_Service();
      com.djws.service.EmpInfo port = service.getEmpInfoPort();
// TODO initialize WS operation arguments here
      if(request.getParameter("empid")!=null){
```

```
java.lang.String id =request.getParameter("empid");
// TODO process result here
      com.djws.service.Employee emp = port.find(id);
      out.println("The details of the employee are as follows:");
       out.println("<br>");
       out.println("<b>ID: </b>" + emp.getEmpid());
       out.println("<br>");
       out.println("<b>Name: </b>" + emp.getEmployeename());
       out.println("<br>");
       out.println("<b>Salary: </b>" + emp.getEmployeesalary());
             }
        }
catch (Exception ex)
        {
      out.println("Exception :" + ex);
        }
    %>
    <%-- end Web service invocation --%><hr/>
        </form>
    </body>
</html>
```

The code shows the JSP client that will consume the Web service. In the code, an input text box and a button named Get Details has been added. The text box is provided to specify the employee id of the employee whose details need to be viewed. The code adds an if statement to check whether an employee ID has been specified. If an employee ID has been specified, the code calls the find() method of the Web service and passes the employee ID to this method. The find() method provides an object of the Employee class. The id, name, and salary of the employee is retrieved using the relevant getter methods of the Employee object and the details are on the Web page.

The style.css file specified in the code is used to format the JSP page.

5.6.4 Build, Package, and Deploy the JSP Client

To build, package, and deploy the client application, right-click the project name and select Run from the context menu. The output of the client application is as shown in figure 5.29.

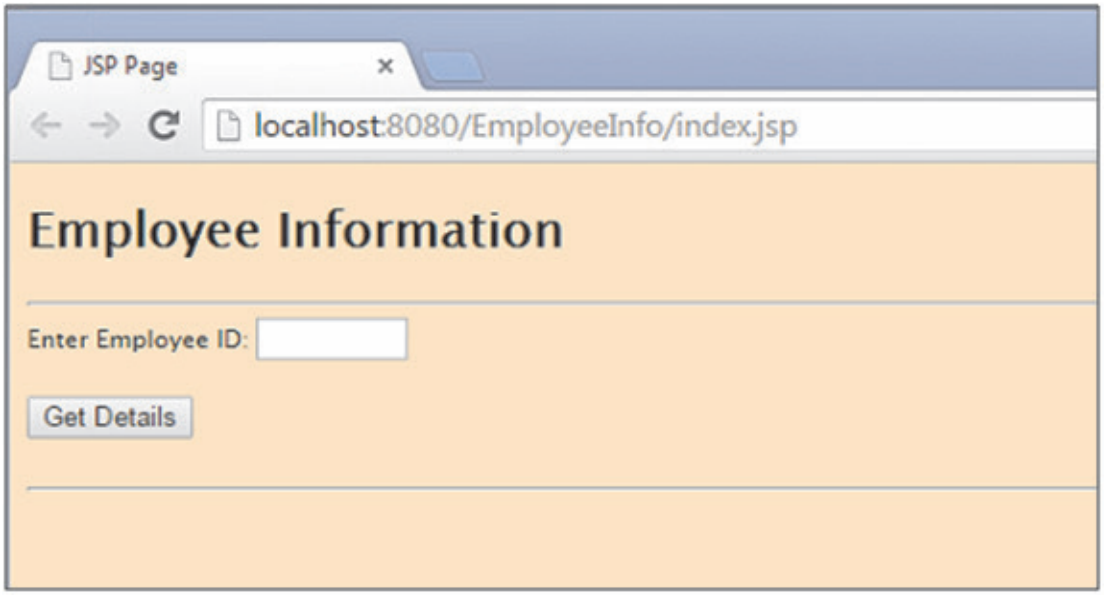


Figure 5.29: Output of JSP Client

When an employee ID is specified and the Get Details button is clicked, the name and salary of the employee whose ID has been specified is displayed on the Web page as shown in figure 5.30.

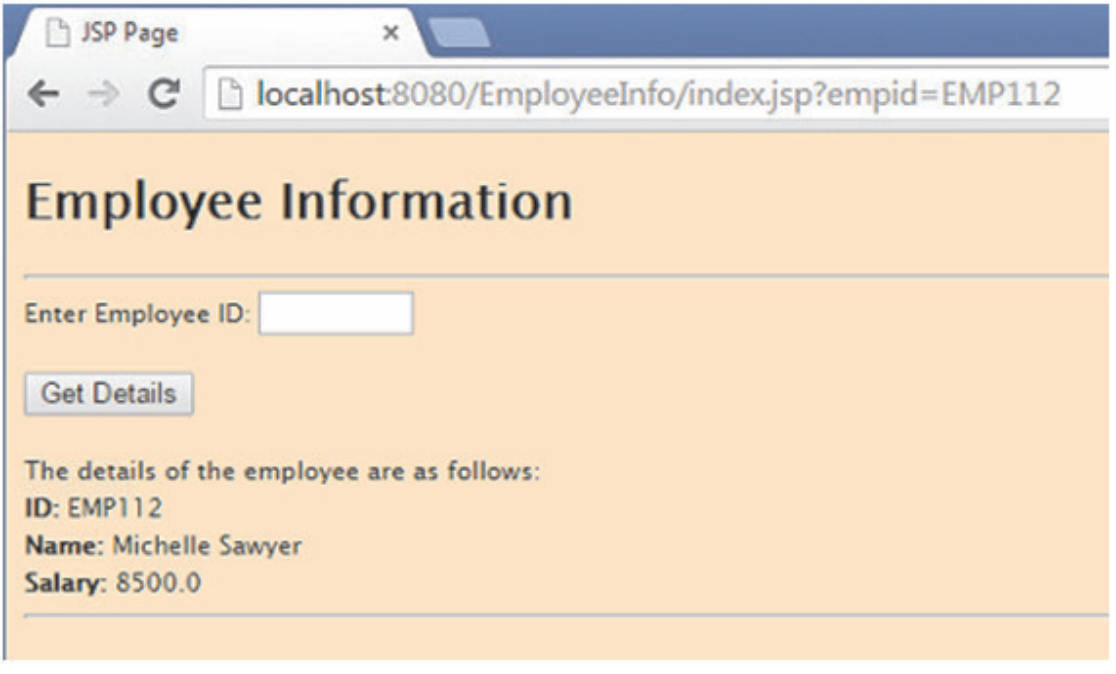


Figure 5.30: Employee Details Retrieved and Displayed on Web Page

5.6.5 Create an Application Client

You can also create a Java application client for the Web service. To do this, perform the following steps:

1. Create a new Java Application project in NetBeans IDE by clicking File → New Project and then, in the New Project wizard, select Java in the Categories list and Java Application in the Projects list.

2. Specify a name and location for the project and ensure that the Create Main Class check box is selected as shown in figure 5.31.

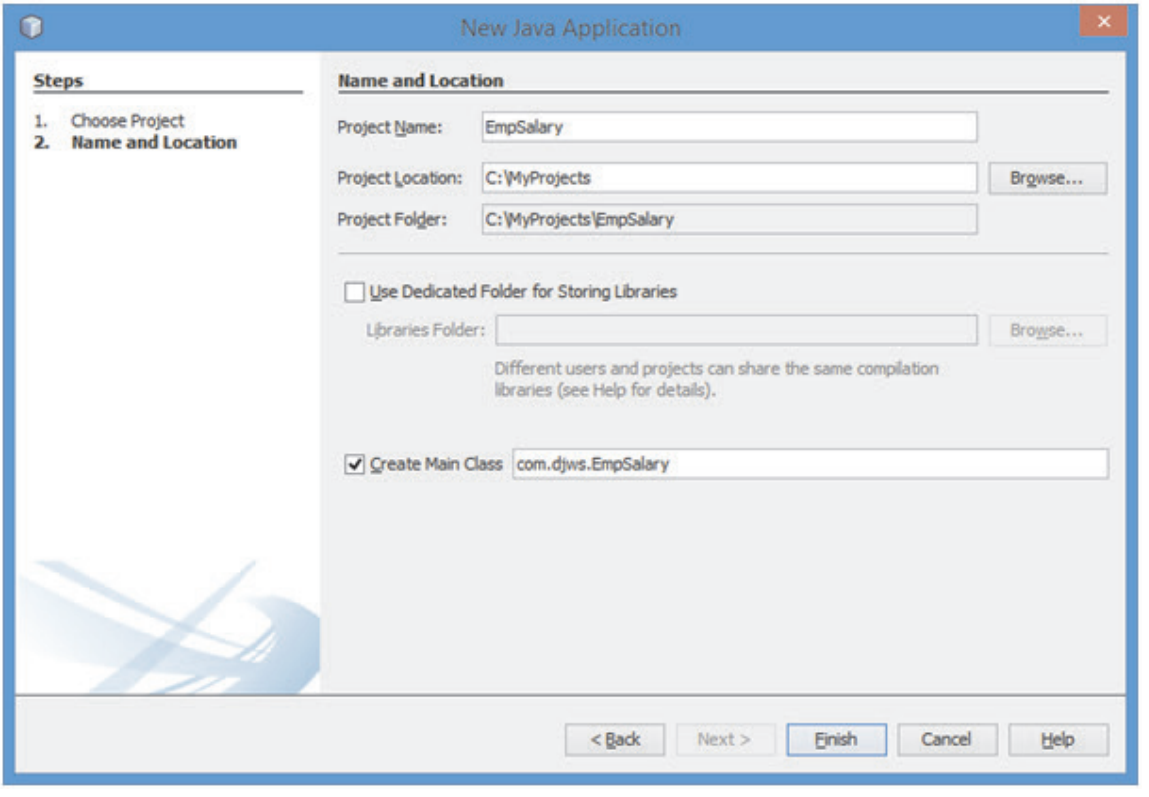


Figure 5.31: New Java Application

3. To create a Web service client, right-click the Java Application project and then click New → Others. Then, in the New File wizard, select Web Services in the Categories list and Web Service Client in the File Types list, as shown in figure 5.32.

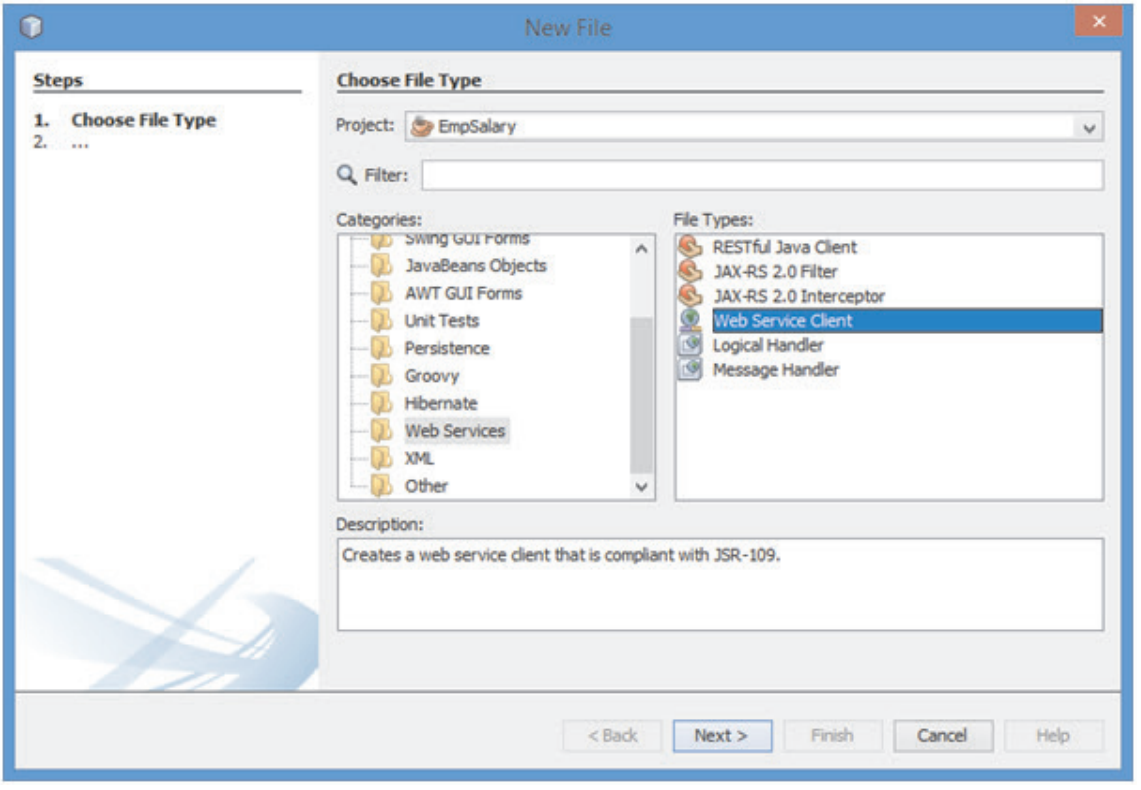


Figure 5.32: Creating a Web Service Client

4. To specify the project that contains the Web service, in the New Web Service Client wizard, click Browse next to the Project box, and then in the Browse, Web Service dialog box, select the Web service for which the client is being created, as shown in figure 5.33.

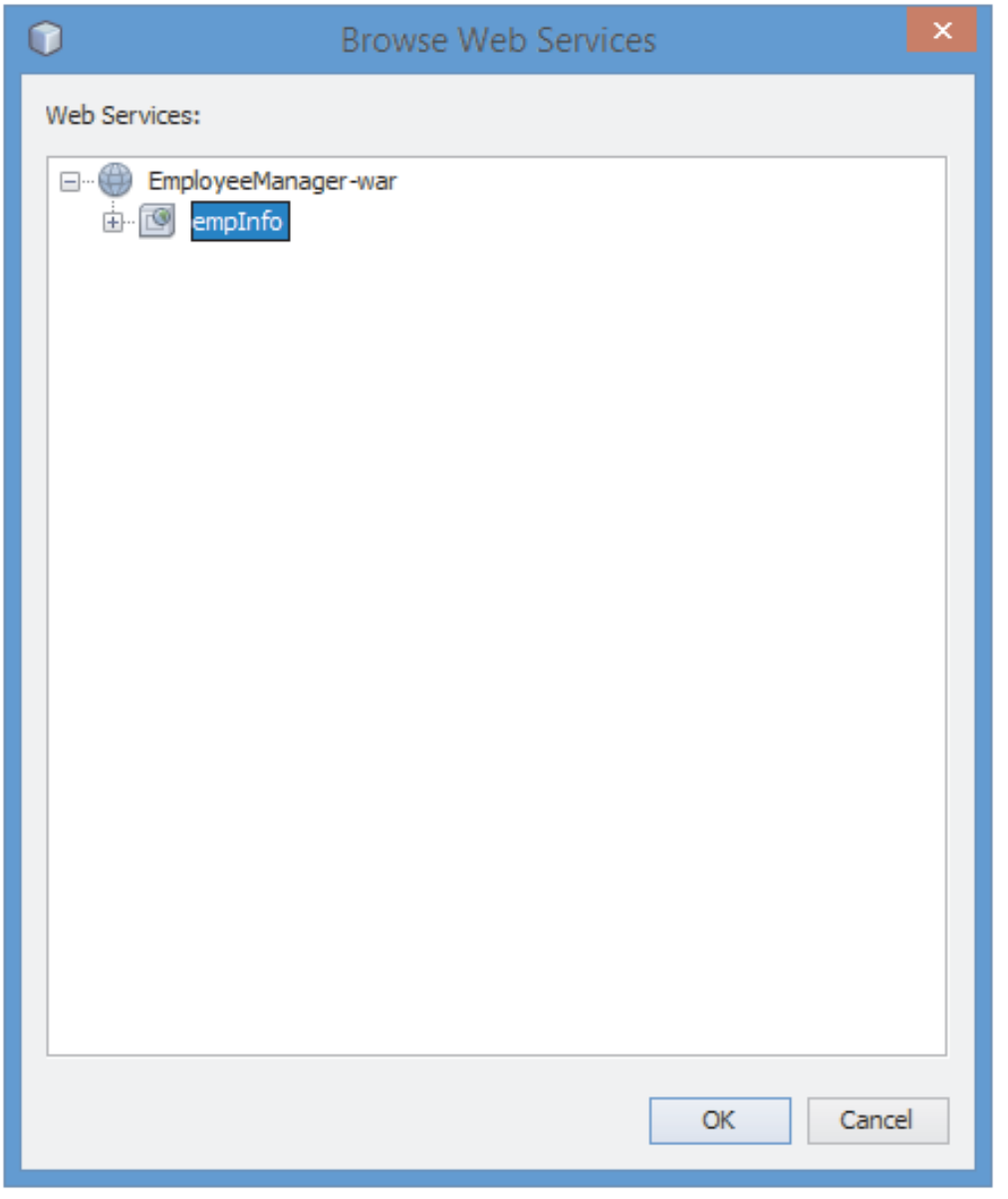


Figure 5.33: Selecting the Web Service

5. The Web Service code will be generated. Build the Web service client project to compile all the generated resources.

6. Open the EmpSalary.java file, right-click inside the main() method, and then click Insert Code as shown in figure 5.34.

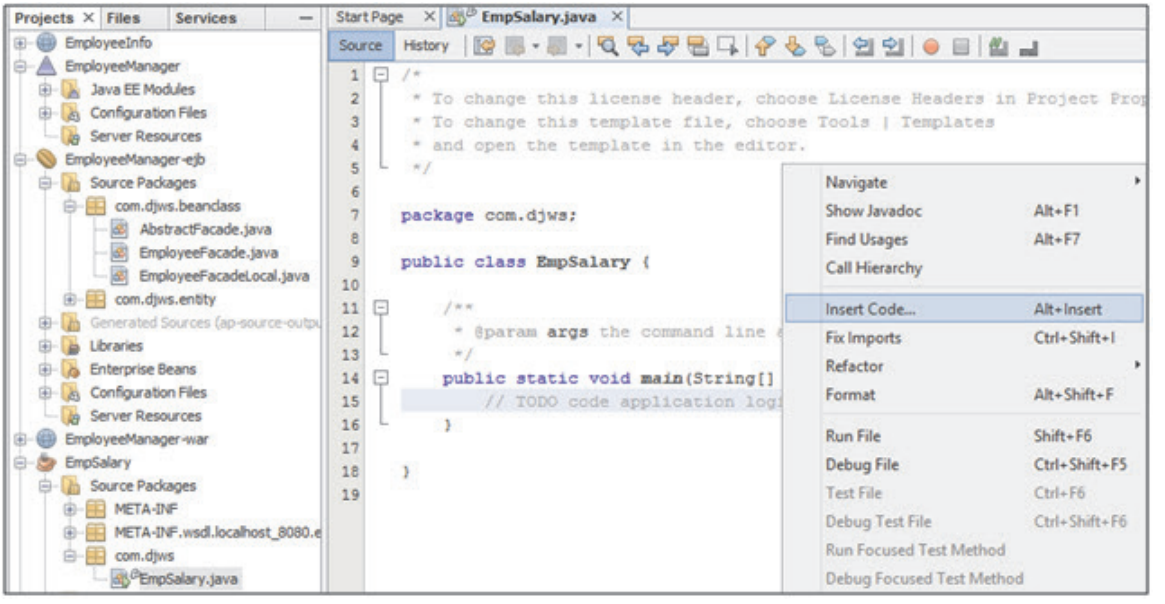


Figure 5.34: Inserting Code

7. In the context menu that appears, click Call Web Service Operation, as shown in figure 5.35.

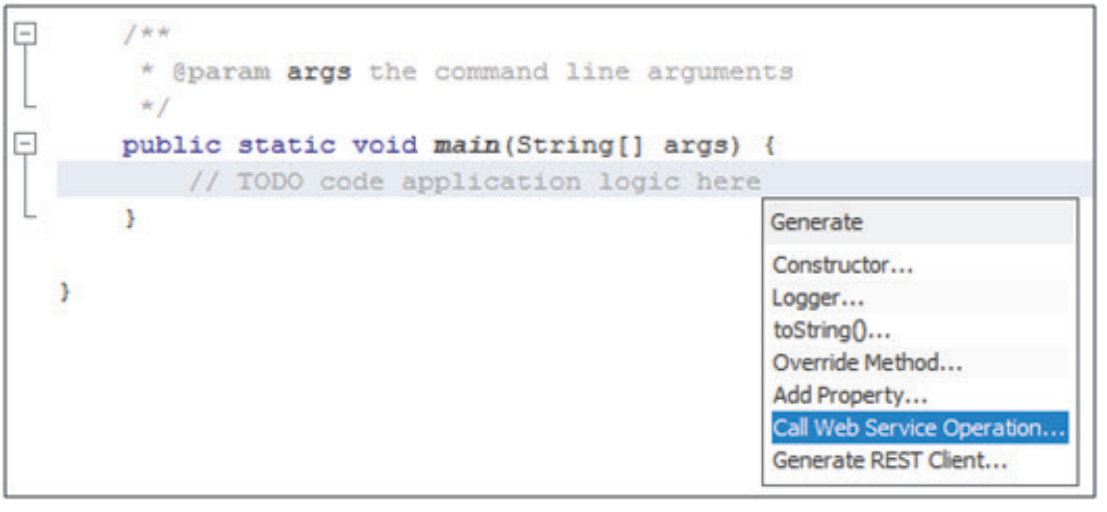


Figure 5.35: Calling Web Service Operation

8. In the Select Operation to Invoke dialog box, select the find() method, as shown in figure 5.36.

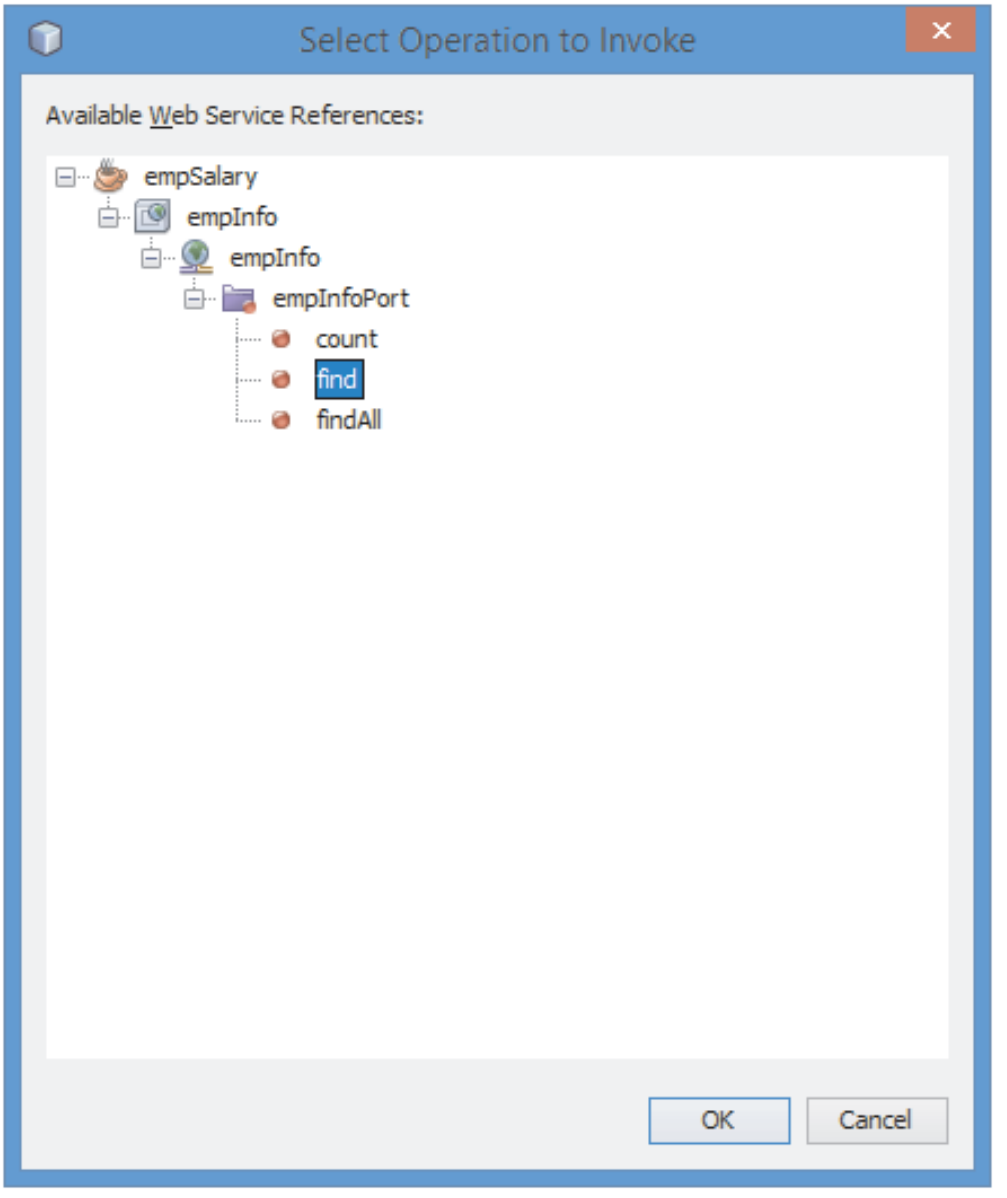


Figure 5.36: Selecting Web Service Operation to Invoke

The IDE adds the code required to invoke the find() method, as shown in figure 5.37.

```
11
12  public class EmpSalary {
13
14      /**
15       * @param args the command line arguments
16       */
17      public static void main(String[] args) {
18          // TODO code application logic here
19      }
20
21      private static Employee find(java.lang.Object id) {
22          com.djws.service.EmpInfo_Service service = new com.djws.service.EmpInfo_Ser
23          com.djws.service.EmpInfo port = service.getEmpInfoPort();
24          return port.find(id);
25      }
26
27  }
```

Figure 5.37: Code to Invoke the Selected Web Service Method

9. To accept user input and display the output, add the code given as shown in Code Snippet 3, in the main() method of the EmpSalary class.

Code Snippet 3:

```
...
import java.util.scanner;
...{
public static void main(String[] args) {
    String empId = null;
    Scanner in = new Scanner(System.in);
    System.out.println("Enter Employee Id :");
    empId = in.nextLine();
    Employee emp = new Employee();
    emp = find (empId);
    System.out.println("Employee ID: " + emp.getEmpid());
        System.out.println("Employee Name: " + emp.
getEmployeename());
    System.out.println("Employee Salary: " +
                        emp.getEmployeesalary());
   }
```

```
    private static Employee find(java.lang.String id) {
     com.djws.EmpInfo_Service service = new
          com.djws.EmpInfo_Service();
     com.djws.EmpInfo port = service.getEmpInfoPort();
     return port.find(id);
}
...
}
```

10. Build and run the EmpSalary project. The user will be prompted to enter the employee ID as shown in figure 5.38.

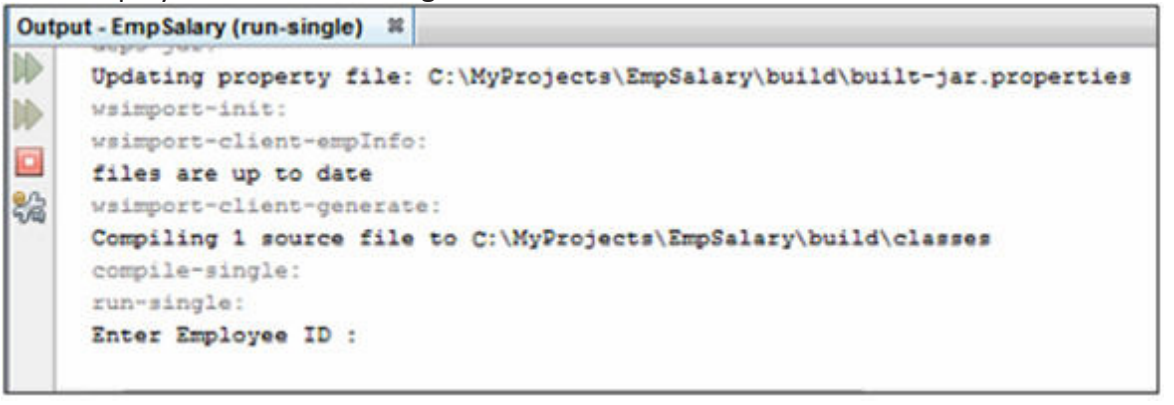


Figure 5.38: Prompt for Entering Employee ID

11. Specify employee ID EMP114 and then press ENTER. The employee name and employee salary is displayed as shown in figure 5.39.

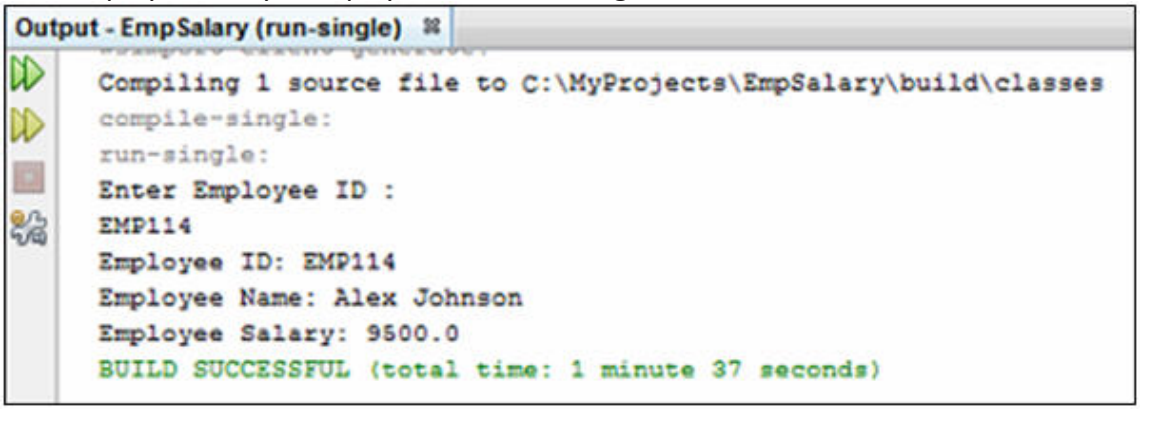


Figure 5.39: Output of the Web Service

Check Your Progress

1. Which Java API supports creation of Web services based on REST architecture?

(A)	JAX-WS	(C)	JAX-RPC
(B)	JAX-RS	(D)	JAXB

2. What is the standard XML-based protocol used for communication over HTTP on the Web?

(A)	SOAP	(C)	XSD
(B)	FTP	(D)	XTP

3. Which annotation specifies whether the service provider accesses the entire message or just the message body?

(A)	@WebServiceProvider	(C)	@ServiceMode
(B)	@WebService	(D)	@WebServiceRef

4. Which annotation generates the service endpoint interface on the service implementation class?

(A)	@WebMethod annotation	(C)	@WebService annotation
(B)	@WebParam annotation	(D)	@WebResult annotation

5. What does the 'use' attribute of javax.jws.soap.SOAPBinding annotation define?

(A)	The message encoding style used in Web services.	(C)	The method parameters placed in the message body.
(B)	The message formatting style used in Web services.	(D)	The XML namespace for return value.

Answers

1.	B
2.	A
3.	C
4.	C
5.	B

Summary

- Web services can be developed using annotations and Plain Old Java Objects (POJOs). They are available on JEE platform, which provides the right environment for easy development and deployment of Web services.
- There are two types of Web services. They are 'Big' Web services and RESTful Web services.
- JAX-WS supports annotations that make development of Web service applications simple and easy.
- JAX-WS data binding and parsing are based on the JAXB and StAX.
- JAX-WS architecture is based on the generation of dynamic proxy class instance, which is responsible to send/receive SOAP request/response on the client/server.
- The two types of service endpoint implementations supported by JAX-WS are the standard JavaBeans service endpoint interface and a new Provider interface.
- The different types of clients in Web service client programming model supported by JAX-WS are Dynamic proxy client and Dispatch client. They support synchronous and asynchronous invocations on the client-side.

Onlinevarsity

your e-way to learning

Visit the

Frequently Asked Questions

section @

www.onlinevarsity.com

06

RESTful Web Services

Welcome to the Session, **RESTful Web Services**.

This session begins with introduction to RESTful Web Services. Further, it describes the JAX-RS API. It also explains how to build RESTful Web Service with JAX-RS API.

In this Session, you will learn to:

- Explain RESTful Web Services
- Describe JAX-RS API
- Explain how to build a RESTful Web Service with JAX-RS API

Session

06

RESTful Web Services

6.1 Introduction to RESTful Web Services

Web application development is based on HTTP protocol. HTTP is a network protocol used for data communication on the World Wide Web. It implements a request-response pattern in the client-server architecture.

Besides HTTP, the other technologies that are used in Web application development are HTML, JavaScript, XML, and Asynchronous JavaScript and XML (AJAX).

Web services are Web applications based on XML standards and transport protocol, HTTP. The development of Web services can be classified based on the technologies and the architectures available on the Web.

When Web services are classified on the basis of technologies, the different technologies that they are dependent on are SOAP, WSDL, XML, security, Uniform Resource Locator (URL), and JAX-RPC, collectively known as 'WS-*'. Thus, it can be defined that 'WS-*' is a set of specifications associated with Web services. Web services are also based on the Web architectures such as client-server architecture, in which a server transfers the information requested by the client. This information is then presented to the client.

The two approaches used for designing the Web services are as follows:

- **Standards-based approach:** This approach is based on SOAP protocol, which is the standard for exchanging XML-based messages.
- **REST-based approach:** This approach is much simpler compared to heavy SOAP-based standards and emphasizes on simple point-to-point communication over HTTP. The concept of Representational State Transfer (REST) defines an architecture style in which Web services are viewed as resources. These resources are identified by Uniform Resource Identifiers (URIs) that are accessible over HTTP protocol.

6.2 REST

REST is a set of guidelines or principles applied to the architectures available in a network system. REST is neither a protocol nor a standard; it is an architecture-style on which systems are designed consisting of protocols, data components, hyperlinks, and clients.

The World Wide Web, a system of interlinked hypertext document is an example of the REST architecture-style. The Web architecture consists of a stateless HTTP connection protocol, server-side documents (HTML and XML) also referred as resources containing hyperlinks associated with URIs, and the client-side interface or application (browser) to read the resources.

The REST architecture guidelines specifies on how the data is transferred between client and server. The different constraints applied to the architectures based on the REST style are as follows:

- **Client-Server:** Architectures must be based on client-server system. The exchange of information between client and server is done using standard interfaces and protocols.

- **Stateless:** The client-server communication should be stateless. Each request from any client should contain all the required information to service the request.
- **Cache:** In the applications based on REST architecture style, responses received, can be cached at the client side for a particular request. The data within the response received are either implicitly or explicitly labeled as cacheable or non-cacheable. Clients can reuse the response data for further requests in case of cacheable response thus, improving the application performance and scalability.
- **Layered components:** A client request can be forwarded to actual server or to some intermediate servers added between client and server to access the data. The intermediate servers such as proxy servers or cache servers added between client and server supports load balancing and also help to enforce security policies.
- **Uniform interface:** Each resource accessed by the client must have a unique address and is accessed using a generic interface such as HTTP GET, POST, PUT, and DELETE.
- **Code on demand:** At runtime, a client can download components such as Java Applets, which are transferred by the server as a response object.

The REST architectural style constraints set the bounds for the architectures based on it.

6.3 RESTful Web Services

RESTful Web services are Web services based on REST architecture and are accessed using HTTP protocol on the Web. The characteristics of the RESTful Web services are similar to SOAP-based Web services, that is, both are platform and language independent.

SOAP-based Web services use SOAP protocol over HTTP to send data to the service. On the other hand, RESTful Web services use Uniform Resource Identifier (URI) to send data directly to the service over the underlying protocol, HTTP.

As RESTful Web services are accessed through URIs and are lightweight compared to SOAP-based Web services. The requirements for developing RESTful Web services based on the REST architecture style are as follows:

- Resources
- Representation of a Resource
- URI
- HTTP methods

6.3.1 Resources

Resource can be defined as any kind of information exposed over the Web. Information can be a document, an image, or an entity such as a book, or a flight detail from a database. Typically, on the Web, a resource is transferred by the server and accessed by the client through an URI over HTTP protocol.

RESTful Web services are a collection of resources and are expressed as Nouns. A unique identifier on the server identifies every resource.

Examples of some RESTful Web services resources are as follows:

- Courses offered by a university
- A CNN news story board
- A search result for any article in the search engine
- Flight information stored in the flight database

6.3.2 Representation of a Resource

Representation can be defined as the format in which the resource is sent and received during client and server interaction on the Web. The format of the resource representation, returned as a response to the client represents the temporary state of the resource.

A resource can be represented in different data formats such as a text, HTML, XML, image, or JavaScript Object Notation (JSON). Different clients can access the different representations for the same resource stored on the server. For example, a client can access either the HTML representation or XML representation of the resource depending on the client requirement.

Figure 6.1 shows the handling of request and response of a RESTful Web service.

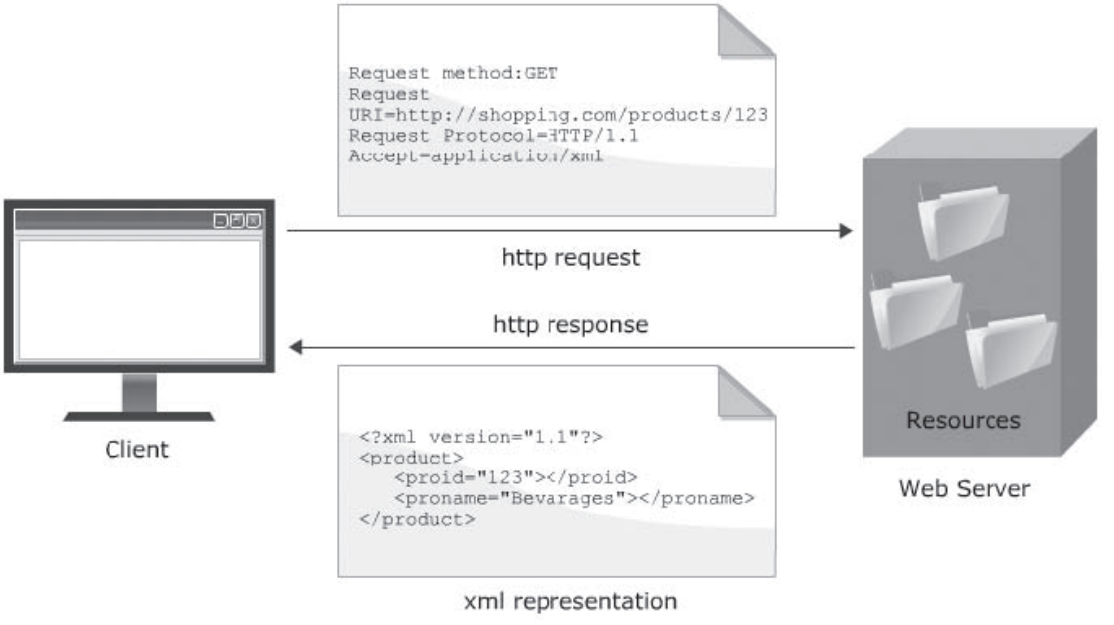


Figure 6.1: Handling of Request and Response in the RESTful Web Service

The given figure shows a client application requesting for a resource on the URI http://shopping.com/products/123 and receiving an XML document containing the details of the product in an XML format.

The RESTful Web services containing multiple resources stored on the server. The format in which the resource is returned as a response is dependent on client requirement.

6.3.3 URI

In the RESTful Web service, a resource is identified by a URI. A URI is a combination of name and address of the resource, which helps the client and server to exchange data formats during interaction.

6.3.4 HTTP

HTTP is a stateless transport protocol used to exchange data over the Web. Web browsers support HTTP methods to send and receive data from the server. Most commonly used HTTP methods for transferring data to the server in a request object are GET and POST. RESTful Web services support these HTTP methods for sending and receiving the data represented as a resource on the Web.

Apart from HTTP methods such as GET and POST, RESTful Web services also support other HTTP methods such as PUT, HEAD, and DELETE.

HTTP methods are also known as **verbs**. Every HTTP method requested by the client corresponds to an action performed on the resource stored on the server. The different types of actions are create, read, update, and delete, also known as **CRUD** actions.

Table 6.1 lists the mapping of HTTP methods with their matching CRUD actions to be performed on the resource by the Web service.

HTTP Method	Action
POST	Create a new resource from the incoming data in the request
GET	Retrieve a resource
PUT	Update a resource from the incoming data in the request
DELETE	Delete a resource

Table 6.1: Mapping of HTTP Methods with their Matching CRUD Actions

6.4 JAX-RS API

RESTful Web services are developed on the Java platform using Java API for RESTful Web services (JAX-RS). JAX-RS specification is an official API on Java EE platform for creating Web services based on REST architecture. JAX-RS is a Java programming API, designed to simplify the development process of RESTful Web service using annotations. The development process of RESTful Web services includes an annotated Java Web service class, known as the resource class. JAX-RS API allows creating two types of resources, which are as follows:

- **Root resource class:** This is a Java class, which acts as an entry point to all the resources present within the class. The class is annotated using the @Path annotation to work like a root resource class.

The @path annotation is specified in the JAX-RS API and defines the base URI on which the root resource is accessible by the client.

- **Sub resources:** These are the methods present in the root resource class. The methods are also annotated using the @Path annotation. The @Path annotation applied on the methods defines the URI which is relative to the base URI defined on the root resource class.

6.4.1 JAX-RS API Annotations

JAX-RS API annotations simplify the development process of RESTful Web services. These annotations help to identify the resources and the actions to be performed on the identified resources. Typically, JAX-RS annotations are runtime annotations. At runtime, the Annotation Processing Tool (APT) processes these annotations and generates the helper classes and necessary artifacts for the resource class.

Annotations available in the java.ws.rs package are categorized as follows:

- @Path annotation
- HTTP method annotations
- Annotations for injecting data from request URI
- Data representation type annotations

The detailed description of these annotations is as follows:

- **@Path annotation:** The @Path annotation specifies the URI of the Web service class, hosted as a resource on the server. The @Path annotation has a single attribute which accepts a String value. The String value specifies the URI to access the resource deployed on the server and is also called as **URI template**. The @Path annotation can be used in the Web service class at the following levels:

  - **Class level:** In this, @Path annotation is used before the Web service class. The URI template specified with @Path annotation represents the base URI to access any method within the Web service class.
  - **Method level:** In this, the @Path annotation is specified on the methods present in the Web service class. The URI template specified in @Path annotation is relative to the root URI applied on the Web service class.

  Code Snippet 1 demonstrates the use of @Path annotation with the URI template applied at class level to access the Web service class deployed on the Web server.

Code Snippet 1:

```
import javax.ws.rs.Path;

import javax.ws.rs.Produces;

import javax.ws.rs.GET;
```

```
//Sets the path to base URL + /Welcome
@Path("/welcome")
public class Welcome{
// This method is called if TEXT_PLAIN is requested
@GET
@Produces(MediaType.TEXT_PLAIN)
public String sayPlainTextWelcome(){
return "Welcome to the Learning Portal";
}
}
```

This code displays a plain text response of "Welcome to the Learning Portal" when the method sayPlainTextWelcome() is called from the Web Service client. In the code, the @GET annotation helps to map the HTTP GET method to the Web service method that retrieves the data. The @Produces annotation specifies that type of output or response that the Web service method will provide. In this case, a call to the sayPlainTextWelcome() method will produce a plain text response as a string.

The URI template can also be embedded with the variables for accessing the resources. The variables embedded in the URI template are used when specific sub resources have to be accessed from the Web service class.

The URI template containing the embedded variables specified with the @Path annotation can be written as follows:

```
@Path("resourcePath/{param1}/…/{paramN}")
```

where, {param1}…{paramN} in the curly braces are the variables. The values of these variables are substituted at runtime from the requested URI invoked by the client.

Code Snippet 2 demonstrates the URI template with the embedded variables to access the sub resource from the Web Service class on the Web server.

Code Snippet 2:

```
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.GET;
```

```
//Sets the path to base URL + /Welcome
@Path("/welcome/{username}")
public class Welcome
{
// This method is called if TEXT_PLAIN is requested
@GET
@Produces(MediaType.TEXT_PLAIN)
public String sayPlainTextWelcome(@PathParam("username")
String userName)
{
return "Welcome to the Learning Portal " + userName;
}
...
@GET
@Path("students/{id}")
@Produces({"application/xml", "application/json"})
public Student find(@PathParam("id") String id)
{
return super.find(id);
}
}
```

Code Snippet 2 displays a plain text response of 'Welcome to the Learning Portal' along with the username when the method sayPlainTextWelcome(String UserName) is called from the Web Service client.

In the given code, the find() method is a sub resource annotated using the @Path annotation. The URI template, students/{id}, given with the @Path annotation is the URI to access the find() method.

This URI is relative to the base URI, /students, of the Welcome class. The {id} represents a variable embedded in the URI. The value for the variable, {id}, is substituted with the value sent by the client through the request URI. Thus, the URI to invoke find() method from client-side application is, http://localhost:8080/Webapp/resources/students/STU001.

The value STU001 will substitute the value in the variable, {id}, when the request is received by the Web service class.

- **HTTP Method Annotations:** These annotations are applied on the methods of the resource class and correspond to the named HTTP methods such as GET, POST, PUT, and DELETE. These annotations are also known as request method designators. Request method designators help to identify the methods responsible for handling HTTP request sent from the client. The annotations in the javax.ws.rs packages corresponding to the equivalent HTTP methods such as GET, POST, PUT, and DELETE are as follows:
  - @GET annotation maps to the HTTP GET method
  - @POST annotation maps to the HTTP POST method
  - @PUT annotation maps to the HTTP PUT method
  - @DELETE annotation maps to the HTTP DELETE method
- **Injecting Data from Request URI Annotations:** These annotations are used to extract the data from the request URI into the resource class.
  - **@PathParam Annotation:** The @PathParam annotation is applied on the parameters of the methods in the Web service class. It extracts the value of the variable embedded in the URI template from the incoming request sent by the client and initializes the parameters of the method.
    The @PathParam annotation has only one attribute named Value. This attribute defines the name of the URI variable whose value will initialize the parameter of the method annotated by @PathParam annotation.
    Code Snippet 3 demonstrates the use of the @PathParam annotation on the parameter of a method of the Web service class present on the Web server.

Code Snippet 3:

```
@PUT
    @Path("{id}")
    @Consumes({"application/xml", "application/json"})
    public void edit(@PathParam("id") String id, Student
entity)
{
        super.edit(entity);
 }
```

In the given code, the edit() method has been annotated with @Path annotation with a URI template, {id}, where {id} represents an embedded variable in the URI template.

The value of {id} is substituted with the value sent in the request URI by the client. The @PathParam annotation is applied on the method parameter of the method, edit(). The annotation extracts this value from the request URI and initializes the parameter, id defined in the method.

- **@QueryParam Annotation:** This annotation is used to extract the query parameter from the request URI sent by the client as a name-value pair. The query parameter is appended with a ? at the end of the request URI.
  For example, in the HTTP request URI sent by the client, http://localhost:8080/Webapp/resources/courses?name="JAVA", the query parameter, name="JAVA" is appended with '?' at the end of the URI.
  The @QueryParam annotation has only one attribute named Value.
  This attribute identifies the name of the query parameter from the HTTP request URI. The JAX-RS runtime extracts and injects the value of the query parameter into the method parameter.
  Code Snippet 4 demonstrates the use of @QueryParam annotation with method parameters in the Web service class.

Code Snippet 4:

```
@Path("/mess")

public     String     getMessage(@QueryParam("str")String
studentName){

return "Welcome " + studentName + " ....";

}

...
```

In the given code, the @QueryParam annotation is used to extract the name of the student from the HTTP request. The name of the student is then used to return a welcome message.

- **Data Representation Type Annotations:** The data representation type annotations specify the data type to be produced or consumed by the Web Service class.

  - **@Produces Annotation:** This annotation is applied on the methods of the Web service class along with @GET, @POST, and @PUT HTTP method annotations. It specifies the data format to be produced by the method of the Web service class to be returned for a client request. The different types of data representation that can be sent back to the client are text/plain, text/xml, or text/json.
    Code Snippet 5 demonstrates the use of @Produces annotation applied to a method of the Web service class.

Code Snippet 5:

```
@GET
    @Override
    @Produces({"application/xml", "application/json"})
    public List<Student> findAll(){
            return super.findAll();
    }
```

In the given code, the method, findAll() is annotated by the @GET and @ Produces annotations. The @GET annotation specifies that the method implement the HTTP GET method, to return the data to the client. The type of data to be returned from the findAll() method is specified by the @Produces annotation. The @Produces annotation has an attribute, application/xml, which is mapped to the HTTP header value, sent along with the HTTP request from the client-side. Since, the HTTP header is set as application/xml the findAll() method produces the contents of type XML to be sent back to the client.

- **@Consumes Annotation:** This annotation specifies the different types of data representations that can be sent by the client to the methods of the Web service class. The different type of data representations consumed by the Web service methods are text/plain, text/xml, or application/x-www-form-urlencoded. Code Snippet 6 demonstrates the use of @Consumes annotation applied at the method level of the Web service class.

Code Snippet 6:

```
@POST
    @Override
    @Consumes({"application/xml", "application/json"})
    public void create(Student entity)
   {
          super.create(entity);
    }
```

In the given code, the create() method reads the Student entity in XML format from the client request and creates a new row in the entity table.

- **@Provider Annotation:** This annotation specifies the different types of data representations that can be sent by the client to the methods of the Web service class. The different type of data representations consumed by the Web service methods are text/plain, text/xml, or application/x-www-form-urlencoded.

Code Snippet 7 demonstrates the use of @Provider annotation applied at the method level of the Web service class.

Code Snippet 7:

```
//Use of @Provider annotation in MessageBodyReader


@Consumes("application/x-www-form-urlencoded")
@Provider
public class FormReader implements MessageBodyReader<Nam
eValuePair> {
.....
....
.....
}
//Use of @Provider annotation in MessageBodyWriter
@Produces("text/html")
@Provider
public class FormWriter implements
     MessageBodyWriter<Student<String studentName, String
studentGrade>>
{
....
....
}
```

Here, the @Provider annotation is used to map MessageBodyReader and MessageBodyWriter to the HTTP request and HTTP response, respectively, using JAX-RS runtime. For HTTP requests, the MessageBodyReader is used to map an HTTP request entity body to method parameters. On the response side, a return value is mapped to an HTTP response entity body by using MessageBodyWriter. If the application needs to supply additional metadata, such as HTTP headers or a different status code, a method can return a Response that wraps the entity and that can be built using Response. ResponseBuilder.

6.5 Building RESTful Web Service with JAX-RS API

The steps to design and build a RESTful Web service using JAX-RS API are as follows:

- Code the implementation class which will act as a root resource class
- Define the methods within the main implementation class that act as sub resources
- Annotate the class with JAX-RS annotations
- Build, package, and deploy the files on the application server
- Access the resource in the client browser application

In order to demonstrate the RESTful Web service, a StudentFacadeREST class is designed which provides information about the Student in an XML format.

6.5.1 RESTful Web Implementation Class

The RESTful implementation class is a resource which can be accessed by the client through a URI. Following criteria must be followed by a class to behave as a root resource class:

- The RESTful implementation class must be annotated with @Path, which specifies it as a base URI of the resource implemented by the service.
- The implementation class must have a public constructor so that it can be invoked by the JAX-RS runtime.
- The HTTP methods accepts different types of request such as GET, POST, PUT, or DELETE for handling various types of requests.

Code Snippet 8 demonstrates the implementation of the StudentFacadeREST Web service class.

Code Snippet 8:

```
package com.syskan.studentdb.entities.service;

import com.syskan.studentdb.entities.Student;

import java.util.List;

import javax.ejb.Stateless;

import javax.persistence.EntityManager;

import javax.persistence.PersistenceContext;

import javax.ws.rs.Consumes;

import javax.ws.rs.DELETE;

import javax.ws.rs.GET;

import javax.ws.rs.POST;

import javax.ws.rs.PUT;
```

```
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
@Stateless
@Path("com.syskan.studentdb.entities.student")
public class StudentFacadeREST extends AbstractFacade<Student>
{
@PersistenceContext(unitName = "com.syskan_StudentDB_war_1.0-
SNAPSHOTPU")
private EntityManager em;

public StudentFacadeREST()
{
super(Student.class);
}
@POST
@Override
@Consumes({"application/xml", "application/json"})
public void create(Student entity)
{
super.create(entity);
}
@PUT
@Path("{id}")
@Consumes({"application/xml", "application/json"})
public void edit(@PathParam("id") Integer id, Student entity)
{
super.edit(entity);
}
```

```
@DELETE
@Path("{id}")
public void remove(@PathParam("id") Integer id)
{
super.remove(super.find(id));
}
@GET
@Path("{id}")
@Produces({"application/xml", "application/json"})
public Student find(@PathParam("id") Integer id)
{
return super.find(id);
}
@GET
@Override
@Produces({"application/xml", "application/json"})
public List<Student> findAll()
{
return super.findAll();
}
@GET
@Path("{from}/{to}")
@Produces({"application/xml", "application/json"})
public List<Student> findRange(@PathParam("from") Integer from, @
PathParam("to") Integer to)
{
return super.findRange(new int[]{from, to});
}
```

```
@GET
@Path("count")
@Produces("text/plain")
public String countREST()
{
return String.valueOf(super.count());
}
@Override
protected EntityManager getEntityManager()
{
return em;
}
}
```

The given code shows the implementation of StudentsFacadeREST Web service. This code uses the Web service to retrieve a list of the student names based on count, by id, or by using the functions findAll(), find(), and findRange() with proper parameters.

6.5.2 Build, Package, and Deploy the RESTful Web Service

Consider a STUDENTS table as shown in figure 6.2.

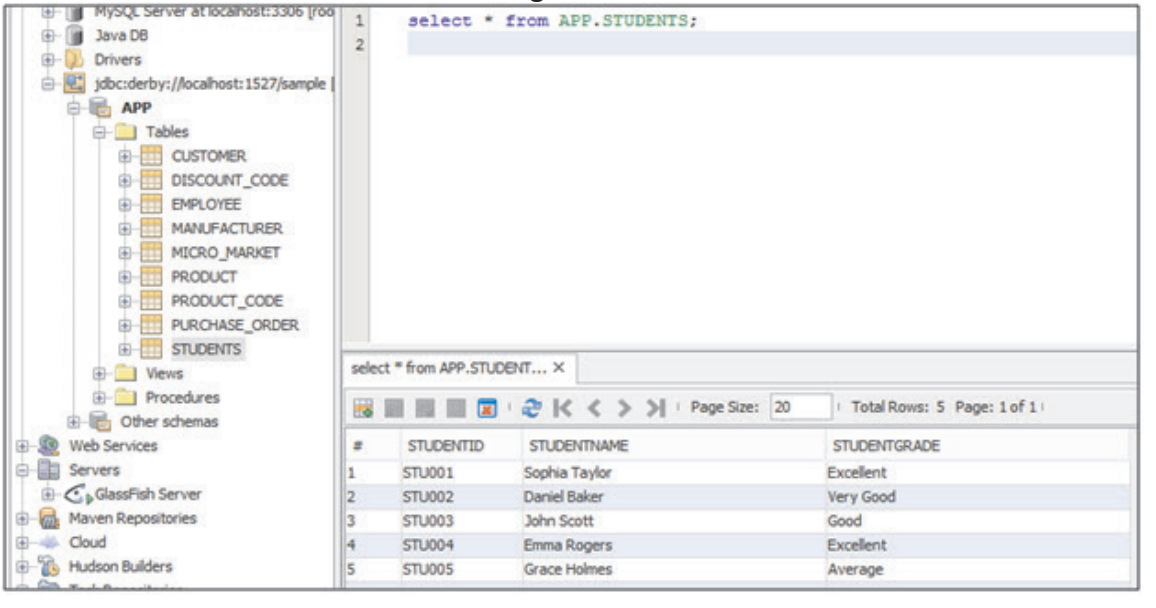


Figure 6.2: Students Database

To create a RESTful Web service based on this database table, perform the following steps:

1. In the NetBeans IDE, create a new Web application project. To do this, click File → New and then in the New Project wizard, select Java Web in the Categories list and Web Application in the Projects list.
2. Specify a name and location for the project.
3. Ensure that GlassFish Server is selected and the Context Path is set.
4. To create a RESTful Web service in the Web application project named StudentManager.
5. Right-click the project node and then click New → Other.
6. In the New File wizard, select Web Services in the Categories list and RESTful Web Services from Database in the File Types list, as shown in figure 6.3.

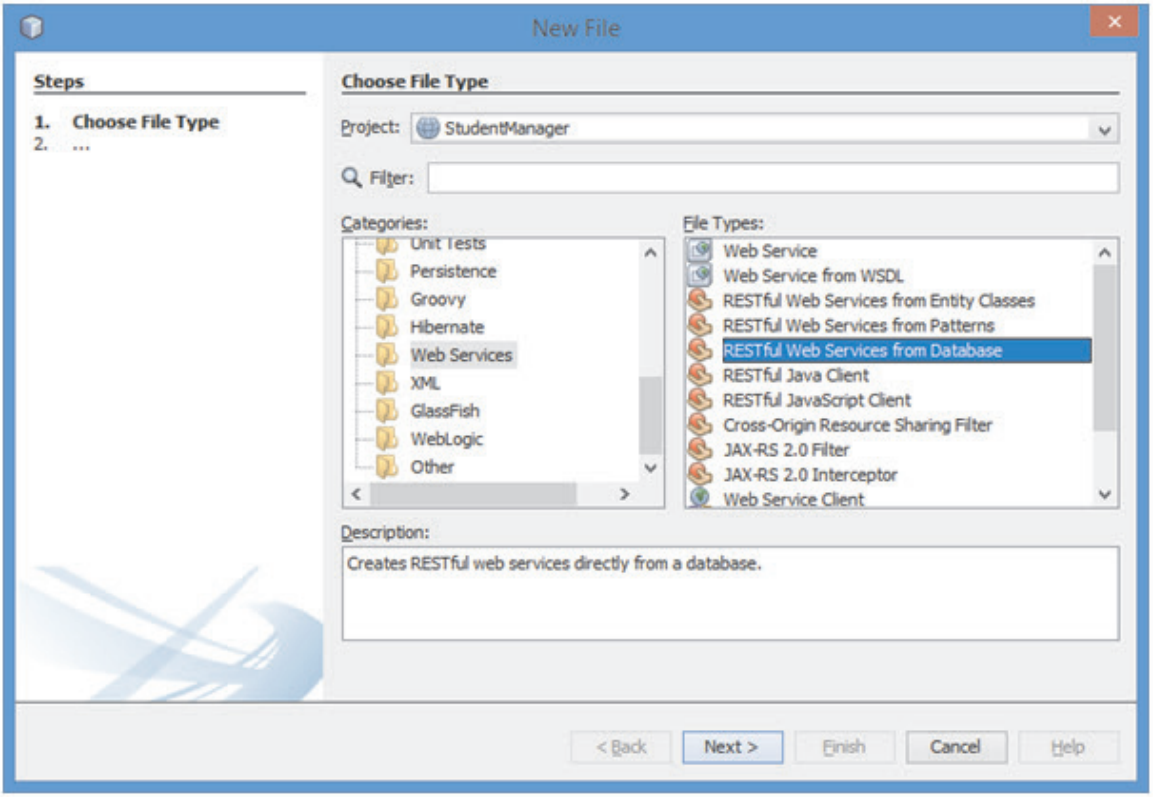


Figure 6.3: Creating RESTful Web Service from Database

7. In the New RESTful Web Service from Database dialog box, in the Data Source drop-down list, select New Data Source as shown in figure 6.4.

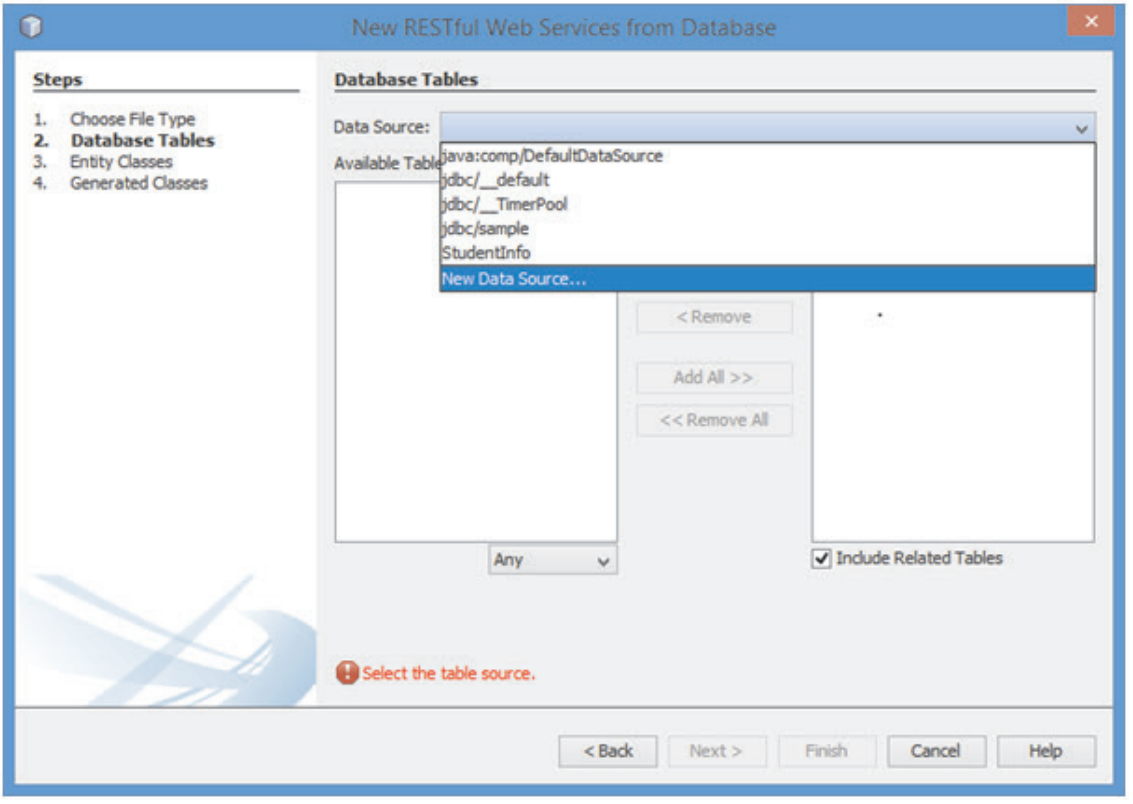


Figure 6.4: Creating a New Data Source

8. In the Create Data Source dialog box, in the Database Connection drop-down list, select jdbc:derby://localhost:1527/sample and specify stuInfo as the JNDI name for the data source as shown in figure 6.5.

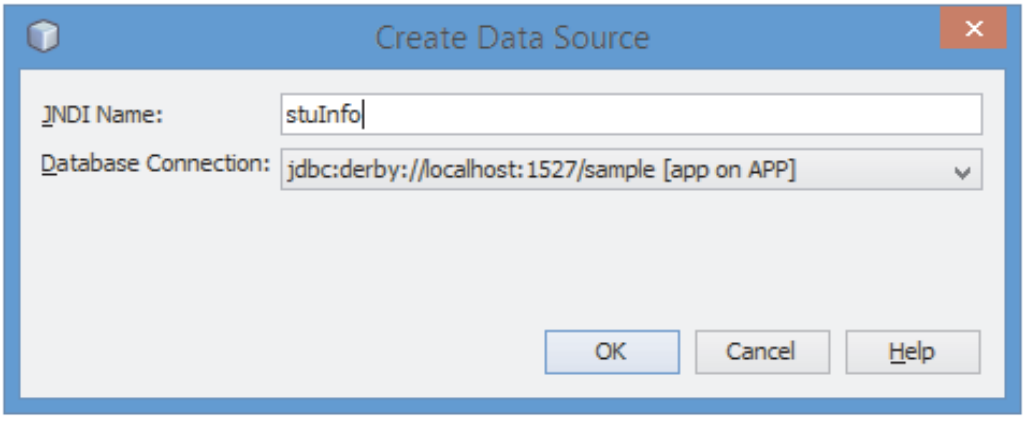


Figure 6.5: Naming the New Data Source

A list of the existing tables are listed in the Available Tables list box.

9. Select the STUDENTS table and click the Add button. The STUDENTS table is added to the Selected Tables list, as shown in figure 6.6.

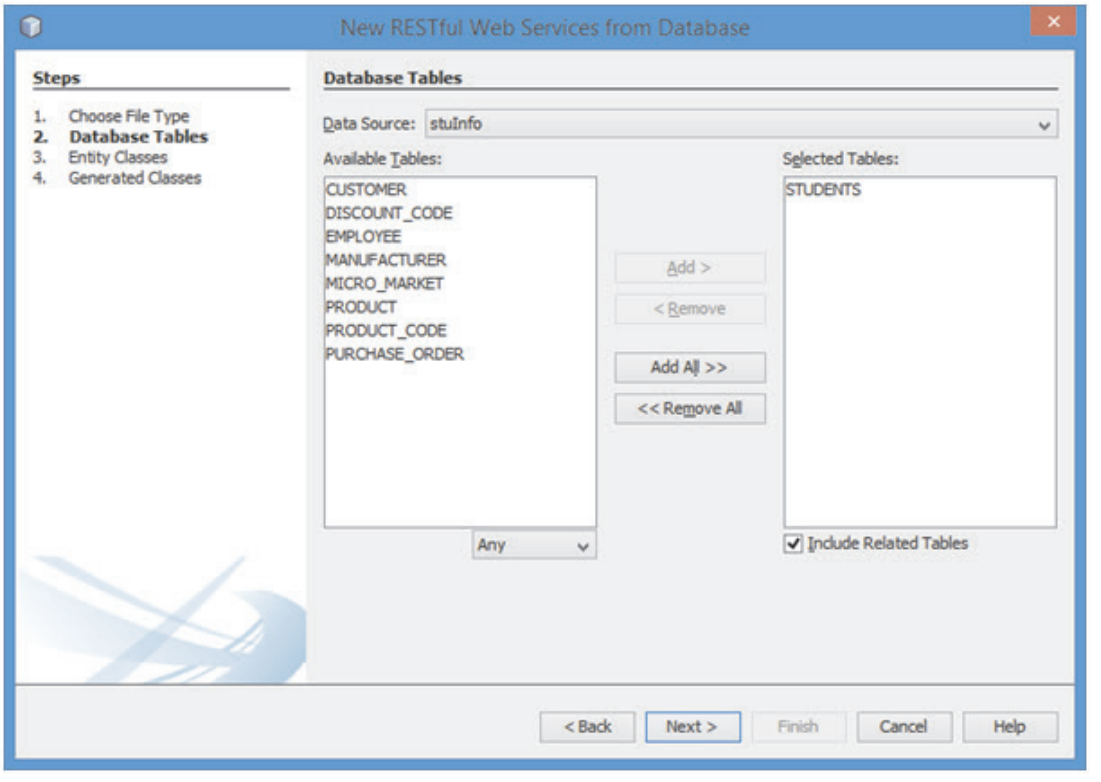


Figure 6.6: Selecting the Database Table

10. Specify a package name for the Web service, as shown in figure 6.7.

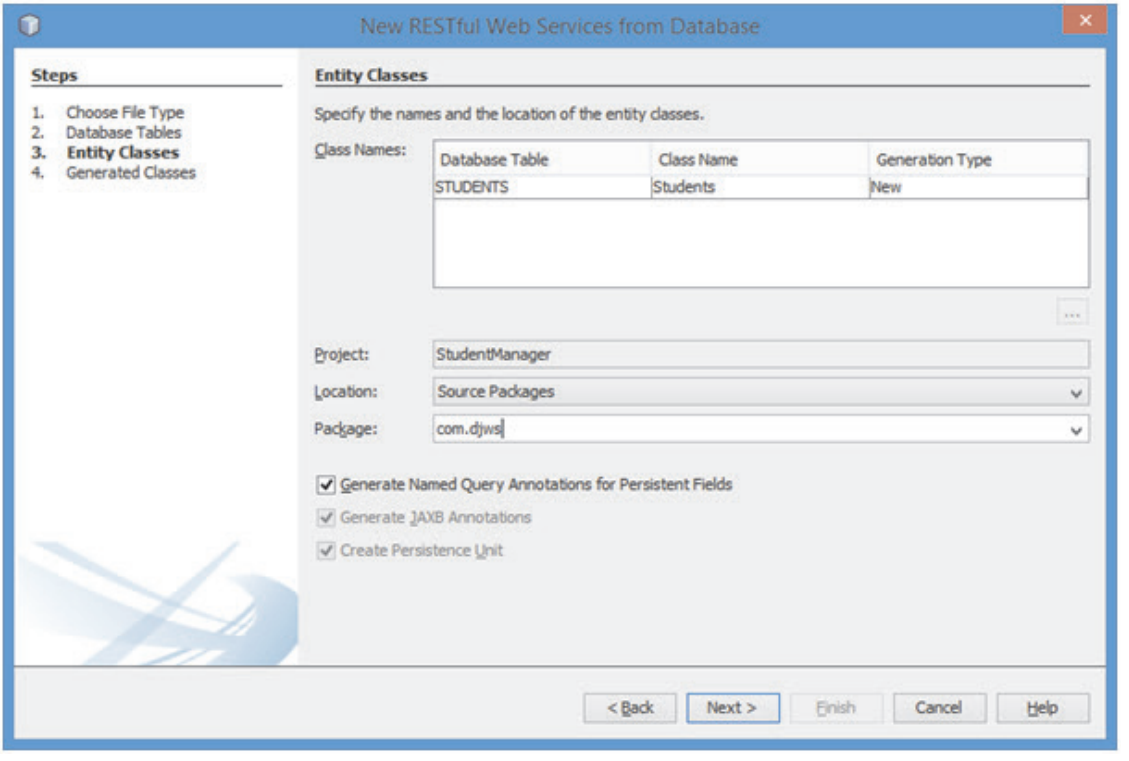


Figure 6.7: Specifying Package Name

11. On the next page of the wizard, accept the defaults and click Finish. The IDE generates the RESTful Web service. The generated entity classes are listed in the com.djws package and the services are listed in the com.djws.service package, as shown in figure 6.8.

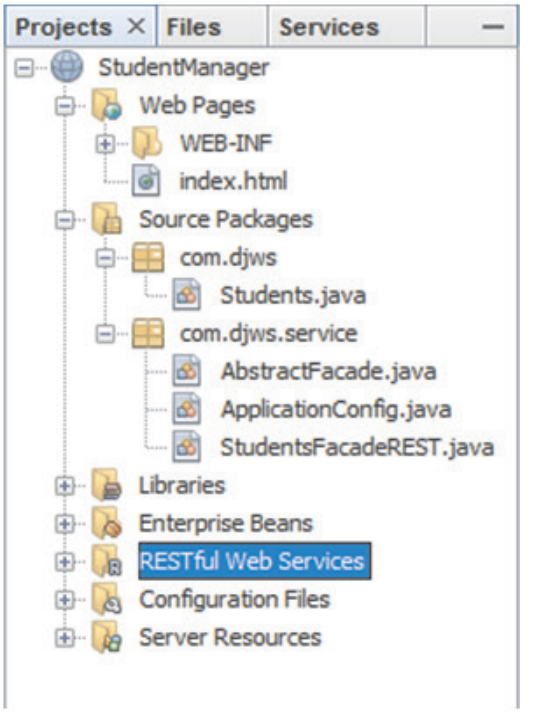


Figure 6.8: Entity Classes and Services

12. To test the Web service, right-click StudentManager project, and then click Test RESTful Web Services, as shown in figure 6.9.

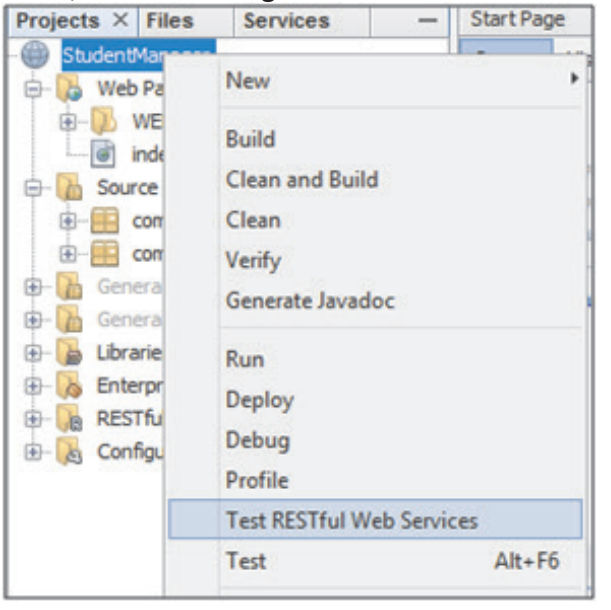


Figure 6.9: Testing RESTful Web Services

The Web service opens in the browser as shown in figure 6.10.

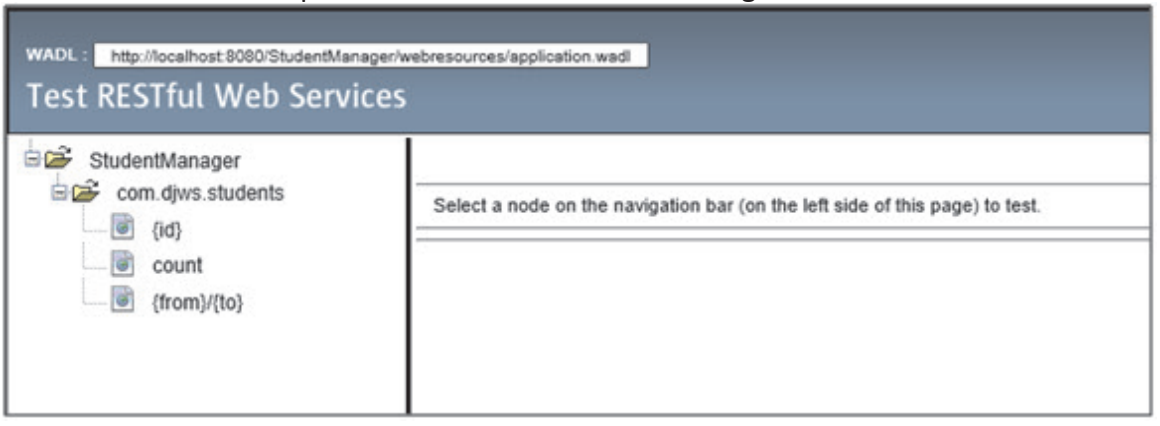


Figure 6.10: RESTful Web Service Tester

13. To find the details of a specific student, in the left pane, click {id}.

14. In the right pane, in the id box, enter the ID of the student STU003 and then click Test. The details of the student whose ID was specified are displayed. This is shown in figure 6.11.

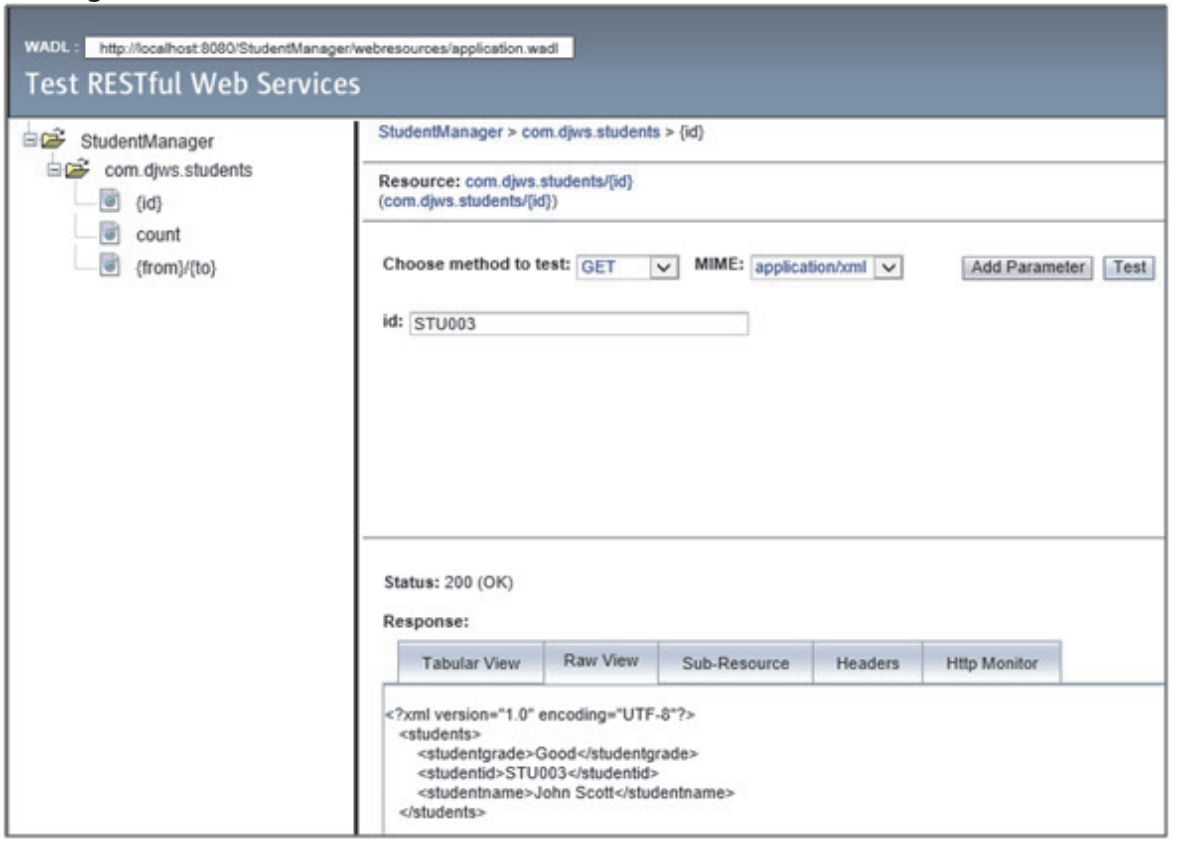


Figure 6.11: Output of the RESTful Web Service

6.5.3 Implementing a Web Client

To create a Web client for the RESTful Web service, perform the following steps:

1. To create a Web client for the RESTful Web service, right-click the StudentManager project node and then click New → Other. Then, in the New File wizard, select Web Services in the Categories list and RESTful Java Client in the File Types list, as shown in figure 6.12.

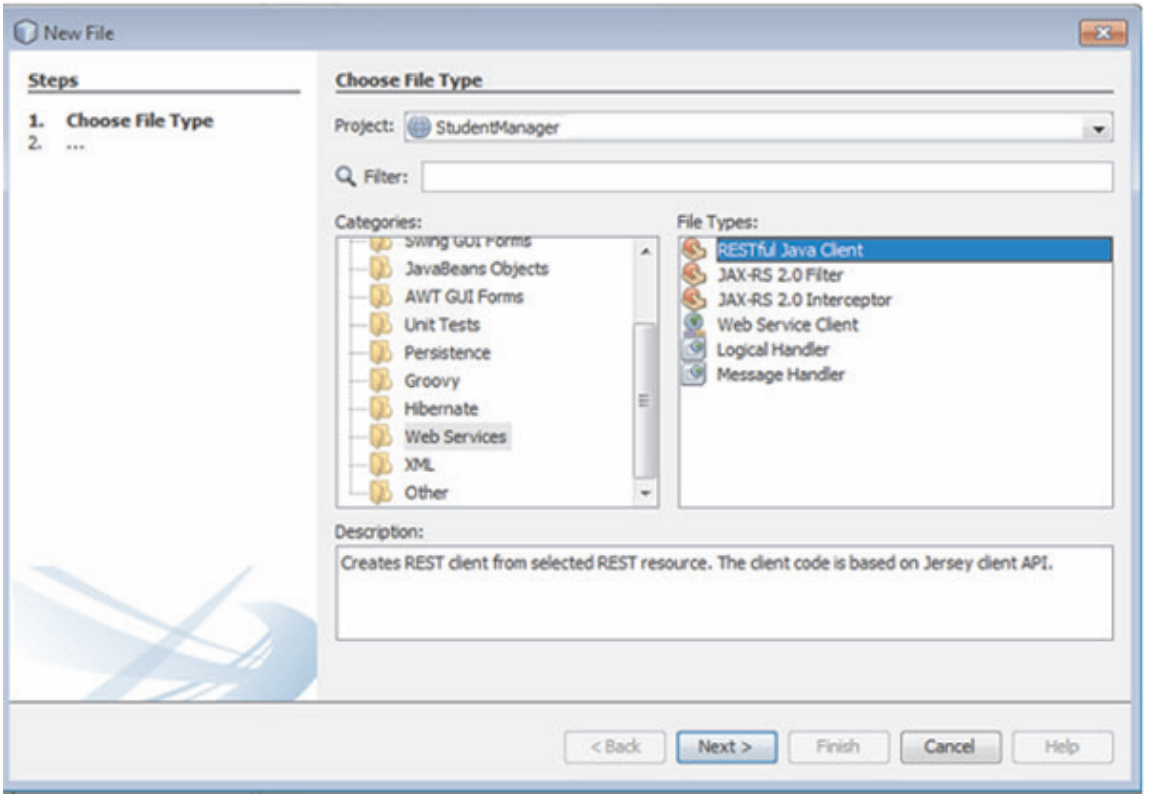


Figure 6.12: Creating a RESTful Java Client

2. Specify a name for the client.

3. To select the REST resource to be used for the client, under Select REST Resource, ensure that the From Project option is selected and then click Browse.

4. In the Available REST Resources dialog box, select the appropriate REST Web service, as shown in figure 6.13.

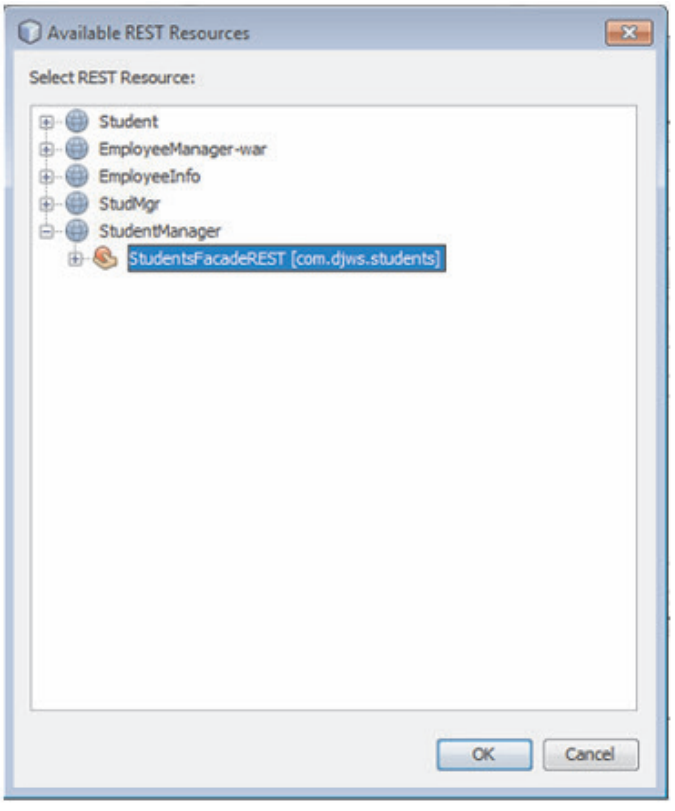


Figure 6.13: Selecting the RESTful Web Service

5. Select the package for the client, as shown in figure 6.14.

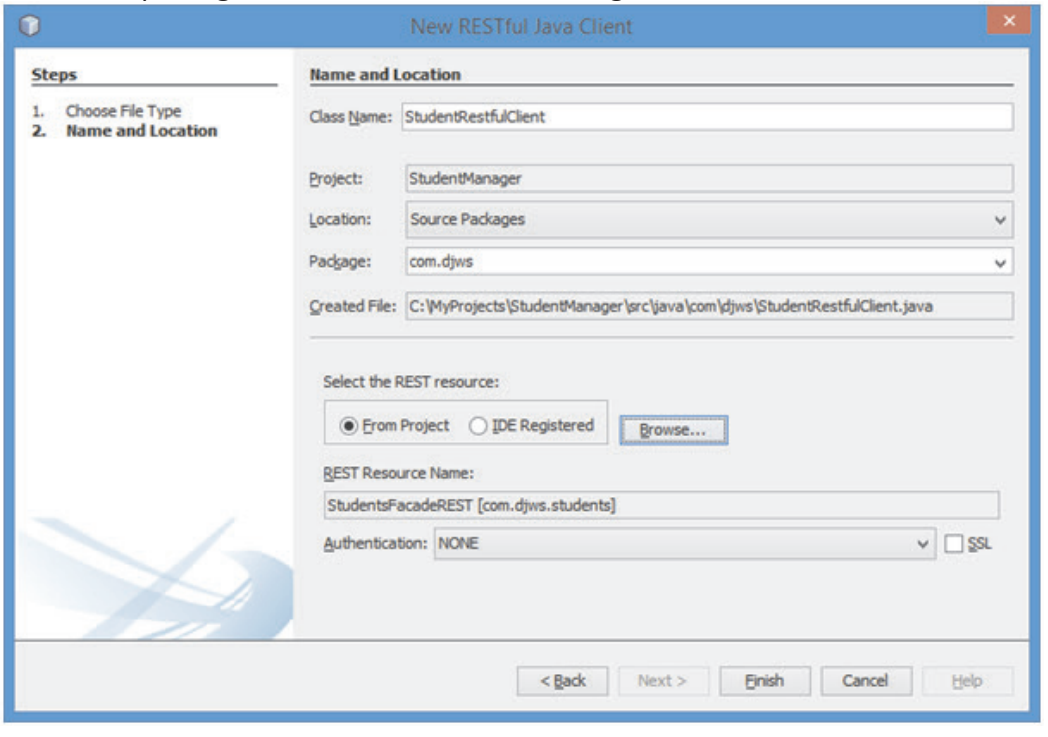


Figure 6.14: Specifying Package for Java Client

6. To create a JSP file for the client, right-click the StudentManager project and click New → Other. Then, in the New File wizard, select Web in the Categories list and JSP in the File Types list, as shown in figure 6.15.

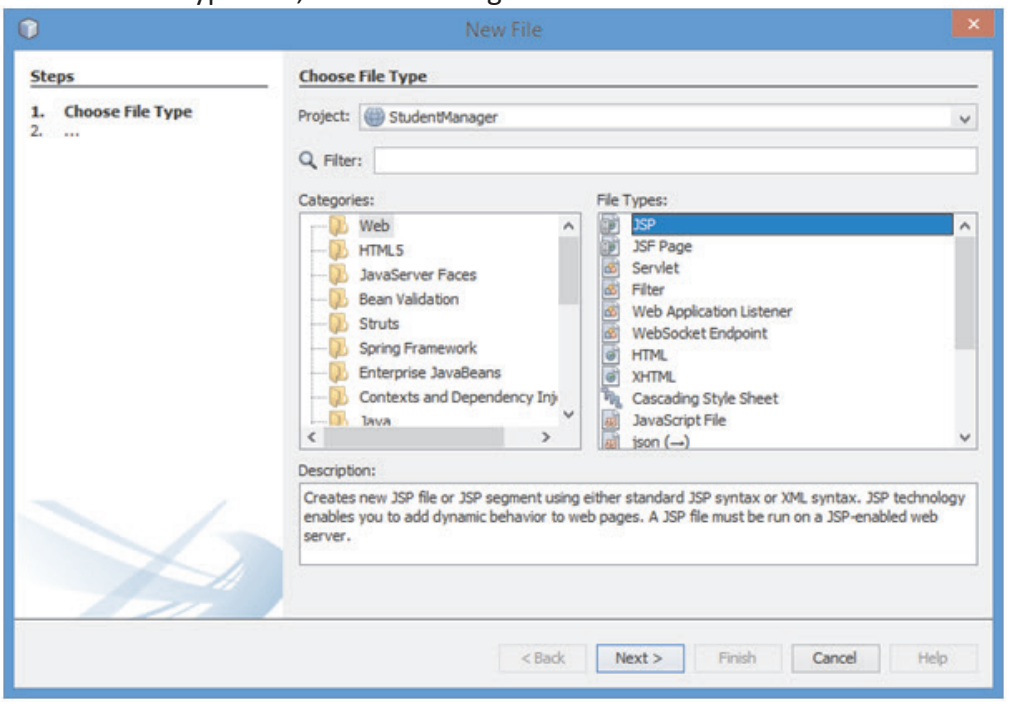


Figure 6.15: Creating a JSP File

7. Specify a name for the JSP file as shown in figure 6.16.

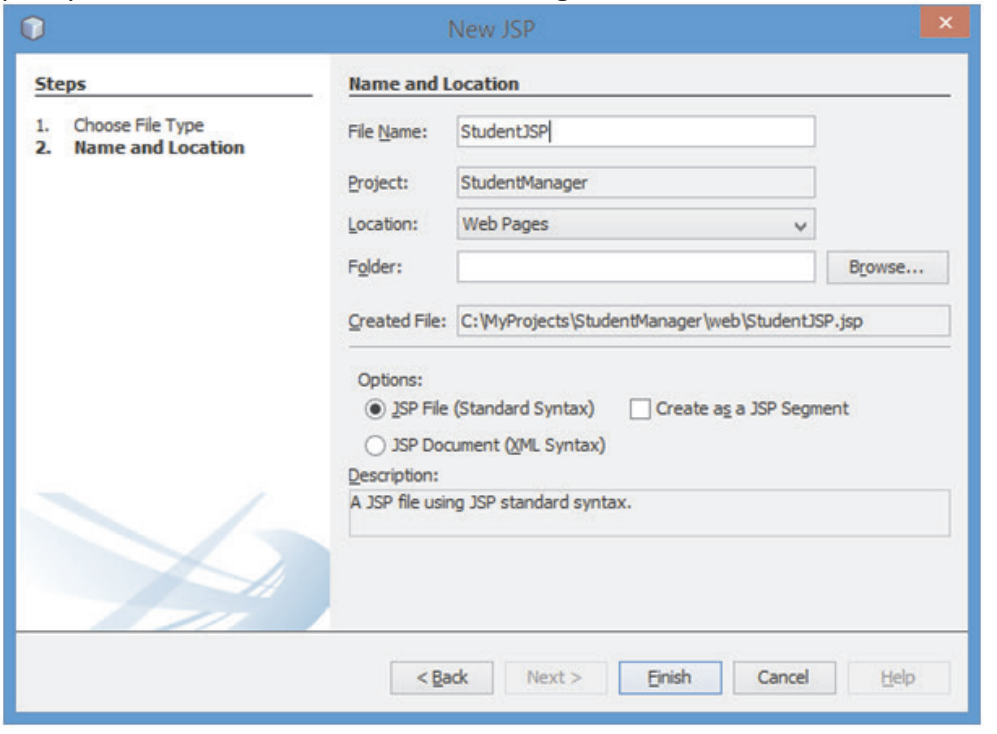


Figure 6.16: Specifying Name for JSP File

8. To create a Servlet for the client, right-click the StudentManager project and click New → Other. Then, in the New File wizard, select Web in the Categories list and Servlet in the File Types list, as shown in figure 6.17.

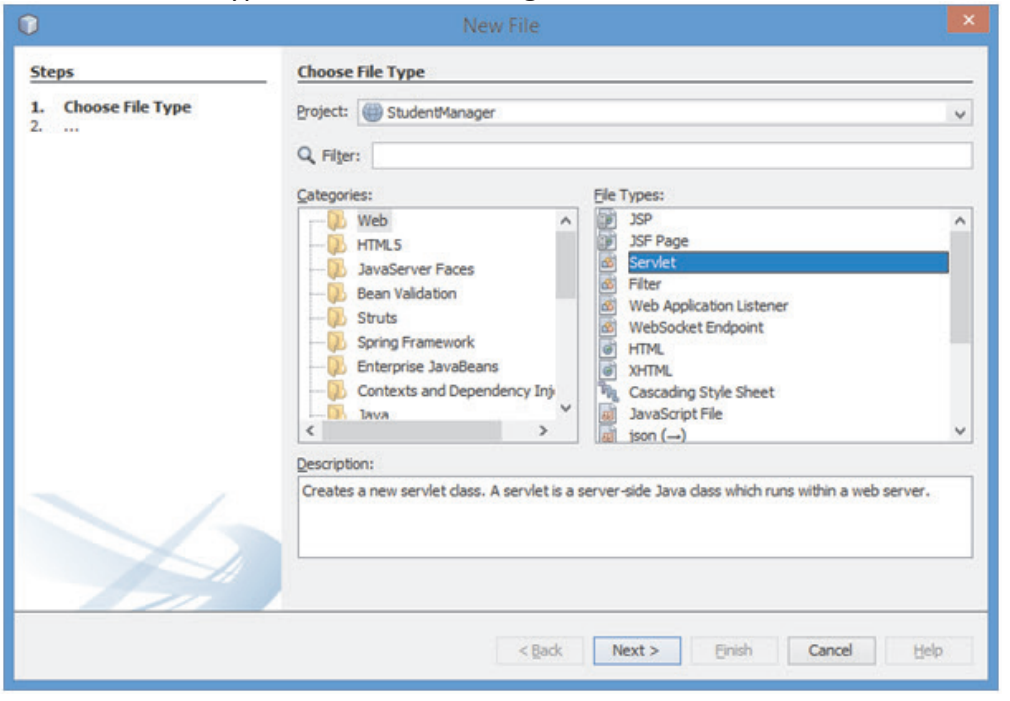


Figure 6.17: Creating a Servlet

9. Specify the name and package for the Servlet as shown in figure 6.18.

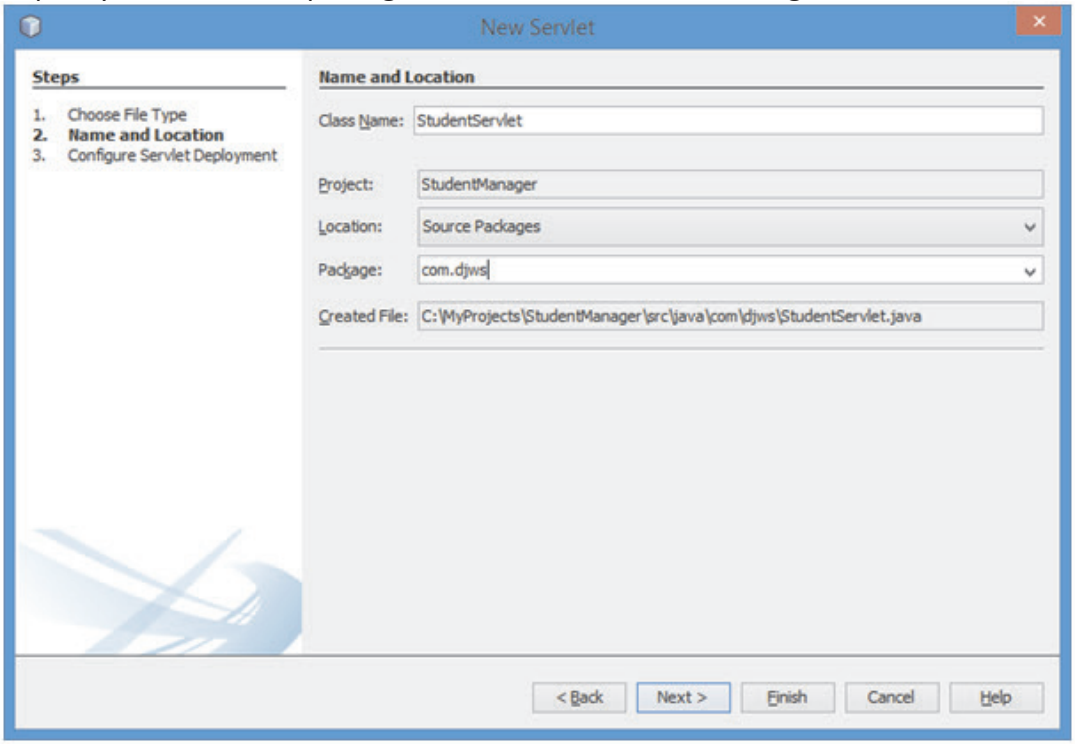


Figure 6.18: Specifying Name for Servlet

10. To add the servlet information to the web.xml page, select the Add information to deployment descriptor (web.xml) check box as shown in figure 6.19.

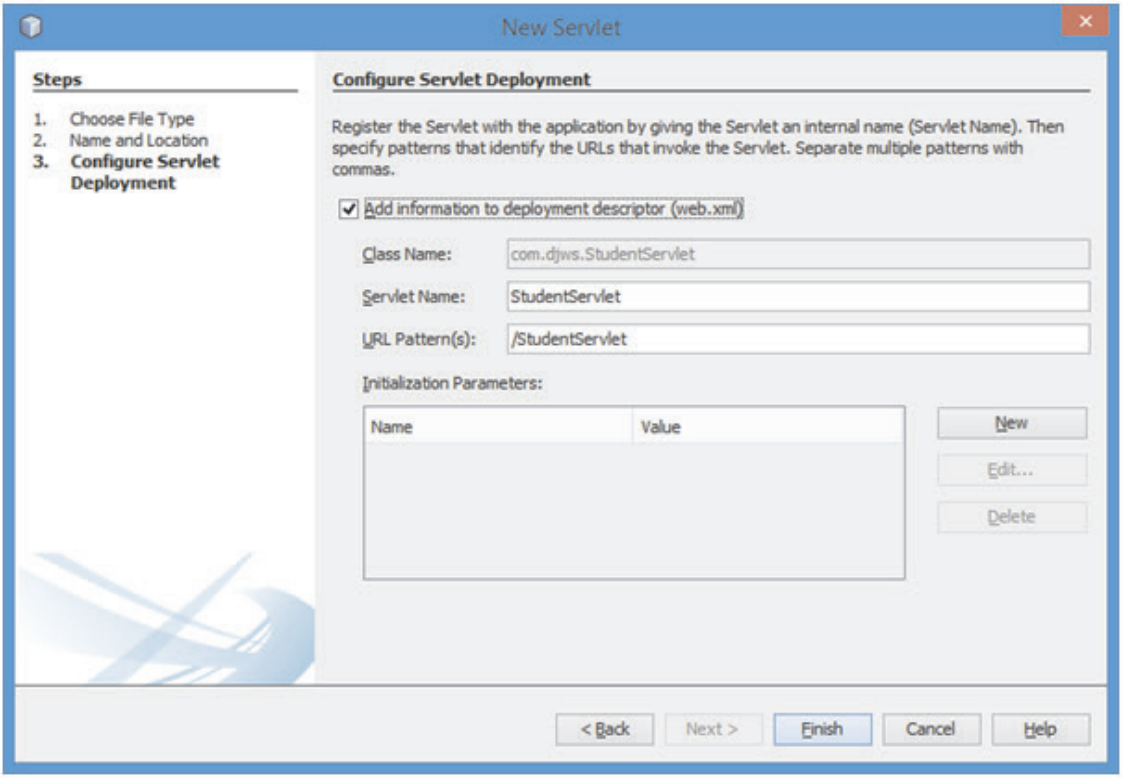


Figure 6.19: Adding Servlet Information to web.xml

11. To add a text box and a submit button to the StudentJSP file, update the code as given in Code Snippet 9.

Code Snippet 9:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>

<!DOCTYPE html>

<html>

  <head>

   <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">

    <title>JSP Page</title>

  </head>
```

```
<body>
  <form action =”StudentServlet”>
    <h1>Student Details</h1>
    Enter Student ID: <input type=”text” name=”txtID”
value=”” />
    <input type=”submit” value=”Submit” name=”Submit” />
   </form>
  </body>
</html>
```

12. To use the Web service to retrieve the student information from the database and display it on the Web page, update the code in the StudentServlet file as shown in Code Snippet 10.

Code Snippet 10:

```
...
protected void processRequest(HttpServletRequest request,
HttpServletResponse response)throws ServletException,
IOException
{
   response.setContentType(“text/html;charset=UTF-8”);
   try (PrintWriter out = response.getWriter())
   {
    /* TODO output your page here. You may use following sample
code. */
   out.println(“<!DOCTYPE html>”);
   out.println(“<html>”);
   out.println(“<head>”);
   out.println(“<title>Servlet StudentServlet</title>”);
   out.println(“</head>”);
   out.println(“<body>”);
   String id = request.getParameter(“txtID”);
```

```
   StudentRestfulClient stuInfo = new StudentRestfulClient();
Students stu = stuInfo.find_XML(Students.class, id);
out.println("Student ID: " + stu.getStudentid()+ "<br/>");
out.println("Student Name: " + stu.getStudentname()+ "<br/>");
out.println("Student Grade: " + stu.getStudentgrade());
stuInfo.close();
   out.println("</body>");
   out.println("</html>");
  }
}
```

13. To specify the startup file for the project, open web.xml and then click the Pages tab, as shown in figure 6.20.

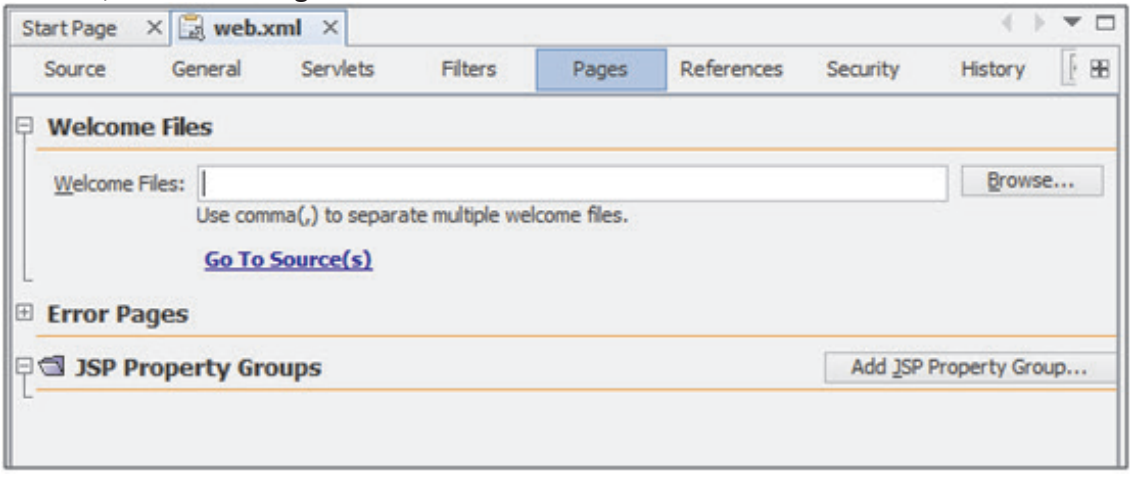


Figure 6.20: Pages Tab of web.xml

14. Click the Browse button and then in the Browse Files dialog box, select the StudentJSP file.

15. Click Select File, as shown in figure 6.21.

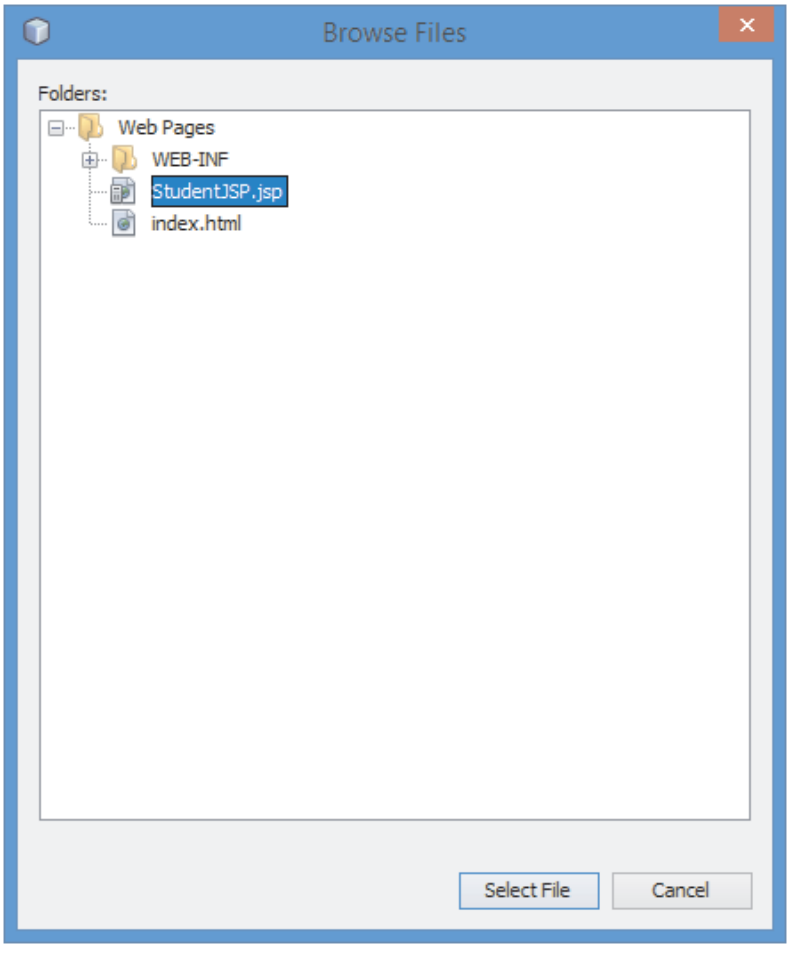


Figure 6.21: Selecting the JSP File

16. Save, build, and run the application. The JSP page opens displaying a text box to enter the student ID and a Submit button, as shown in figure 6.22.

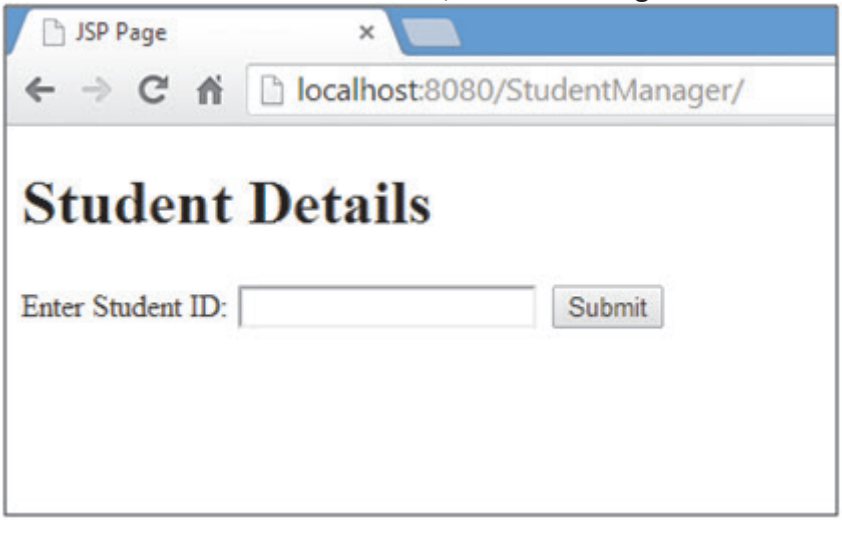


Figure 6.22: JSP Page for Entering Student ID

17. Enter a student ID, as shown in figure 6.23.

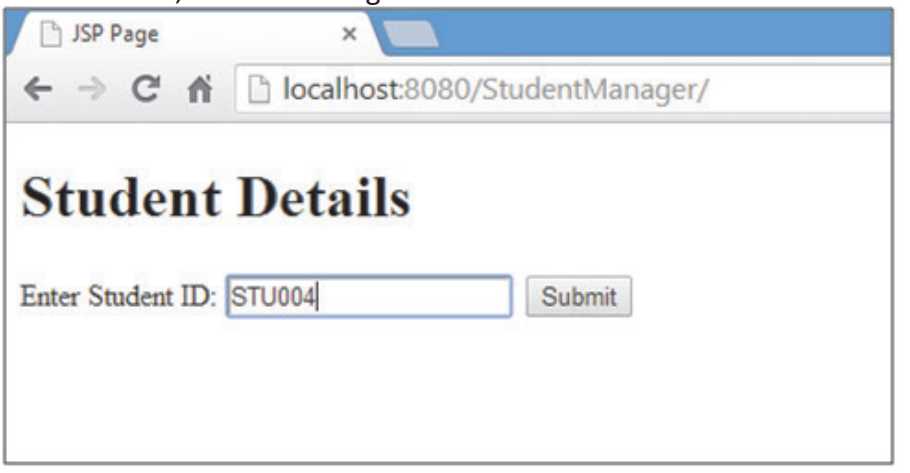


Figure 6.23: Entering Student ID

18. Click the Submit button. The student's details are displayed. This is shown in figure 6.24.

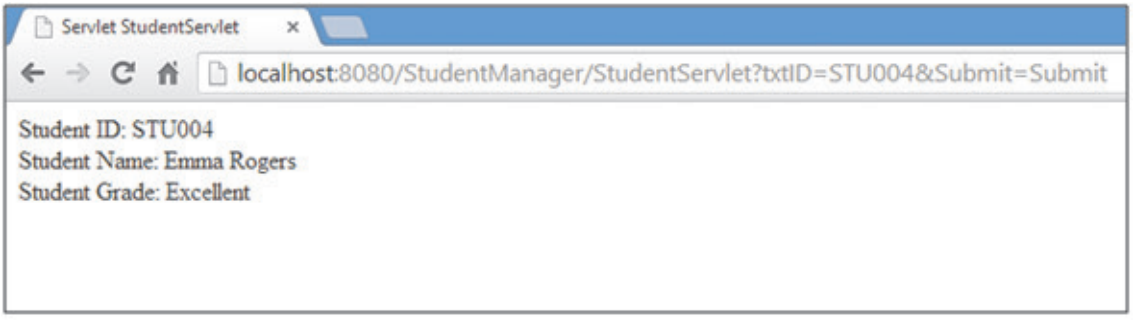


Figure 6.24: Student Details Displayed

Check Your Progress

1. _____ is any kind of information exposed on Web.

(A)	Resource	(C)	Data
(B)	Pages	(D)	Images

2. _____ is the set of guidelines for developing the Web Services.

(A)	Web	(C)	HTTP
(B)	REST	(D)	SOAP

3. Which annotation in JAX-RS specifies the URL path for the Web service on which the resources are hosted?

(A)	@Path annotation	(C)	@WebService notation
(B)	@GET annotation	(D)	@WebParam annotation

4. _____ annotations applied on the methods of the resource class corresponds to the named HTTP methods in the resource class.

(A)	SOAP method annotation	(C)	Data Injecting annotations
(B)	URL path method annotations	(D)	HTTP method annotations

5. _____ is used to extract the query parameter from the request URI sent by the client as a name-value pair.

(A)	@GET	(C)	@QueryParam
(B)	@Path	(D)	@Consumes

Answers

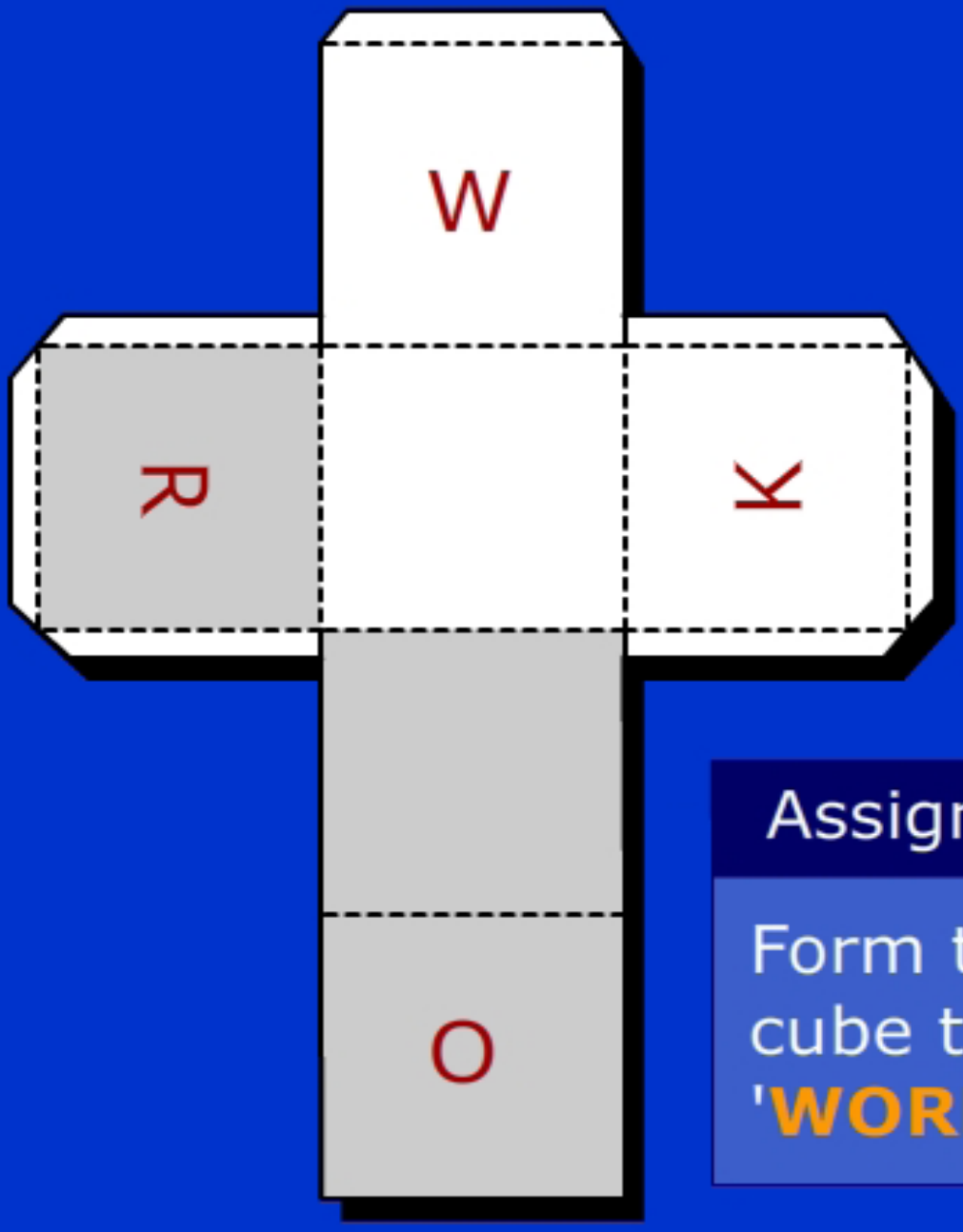
1.	A
2.	B
3.	A
4.	D
5.	C

Summary

- Web services are Web applications based on XML standards and transport protocol, HTTP.
- Web services can be developed either using SOAP based approach or REST based approach.
- REST is an architecture style used for developing Web services. It is a set of guidelines or principles for developing a network-system.
- RESTful Web services are Web services based on REST principles and accessed over HTTP protocol on the Web.
- The requirements for developing RESTful Web services based on the REST architecture style are resources, data representation, URI, and HTTP methods.
- JAX-RS is a Java programming API designed to simplify the development process of RESTful Web service.
- JAX-RS API is based on annotations such as @Path, @GET, @POST, @PUT, @DELETE, @Produce, @Consumes, and so on which are present in javax.ws.rs package.

WORK ASSIGNMENT

Onlinevarsity
your e-way to learning



Assignment

Form the cube to read '**WORK**'.

"Practice does not make perfect. Only perfect practice makes perfect."
- Vince Lombardi

For perfection, solve the assignments @

www.onlinevarsity.com