WEB SERVICES

Computer Science & Engineering

M.Tech - 1st Year 1st Semester

www.btechsmartclass.com

LAB MANUALS

Author : Rajinikanth B | Regulation: R13 | Year 2016

WEB SERVICES

LAB MANUAL

OBJECTIVE

- To implement the technologies like WSDL, UDDI.
- To learn how to implement and deploy web service client and server

LIST OF PROGRAMS

1. Write a program to implement WSDL Service (HelloService.WSDL File)
2. Write a program the service provider can be implement a single get price (), static bind () and get product operation.
3. Write a program to implement the operation can receive request and will return a response in two ways.
   a) One-Way operation
   b) Request - Response
4. Write a program to implement to create a simple web service that converts the temperature from Fahrenheit to Celsius (using HTTP Post Protocol)
5. Write a program to implement business UDDI Registry entry
6. Write a program to implement
   a) Web based service consumer
   b) Windows application based web service consumer

1. Write a program to implement WSDL Service (HelloService.WSDL File)

- **Creating a HelloService Web Service**

1. Open Netbeans 7.4
2. Open New Project → Java Web → Web Application
3. Type Project Name **HelloWorld**
4. Right Click on Source Packages and add new web service **HelloService** with package **org.me.Hello**.
5. Open **HelloService.java** which is in Source packages → **org.me.Hello** and create web service as follows:

```
package org.me.Hello;

import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebParam;

@WebService(serviceName = "HelloService")
public class HelloService {

@WebMethod(operationName = "hello")
   public String hello(@WebParam(name = "username") String user) {
      return "Hello " + user + " !";
    }
}
```

6. Now Right Click on Project Name HelloWorld, Build and then Deploy.
7. Test the web service
   Expand web services and Right click HelloService and Select Test Web Service
   Or use the following url in browser.
   http://localhost:8080/HelloWorld/HelloService?Tester

8. To View WSDL file use the following url in browser.
   http://localhost:8080/HelloWorld/HelloService?WSDL

HelloService.WSDL Contents

```
<definitions targetNamespace=http://Hello.me.org/ name="HelloService">
      <types>
          <xsd:schema>
          <xsd:import namespace=http://Hello.me.org/
                        schemaLocation="http://localhost:8080/HelloWorld/HelloService?xsd=1"/>
          </xsd:schema>
     </types>
     <message name="hello">
         <part name="parameters" element="tns:hello"/>
     </message>
     <message name="helloResponse">
        <part name="parameters" element="tns:helloResponse"/>
    </message>
    <portType name="HelloService">
          <operation name="hello">
                <input wsam:Action="http://Hello.me.org/HelloService/helloRequest"
                              message="tns:hello"/>
                <output wsam:Action="http://Hello.me.org/HelloService/helloResponse"
                                        message="tns:helloResponse"/>
          </operation>
     </portType>

    <binding name="HelloServicePortBinding" type="tns:HelloService">
         <soap:binding transport=http://schemas.xmlsoap.org/soap/http style="document"/>
            <operation name="hello">
                <soap:operation soapAction=""/>
                        <input><soap:body use="literal"/></input>
                        <output><soap:body use="literal"/></output>
            </operation>
    </binding>
    <service name="HelloService">
         <port name="HelloServicePort" binding="tns:HelloServicePortBinding">
             <soap:address location="http://localhost:8080/HelloWorld/HelloService"/>
         </port>
    </service>

</definitions>
```

Definition: HelloService

- Type: Using built-in data types and they are defined in XMLSchema.
- Message : HelloRequest : firstName parameter Helloresponse: greeting return value
- Port Type: HelloService operation that consists of a request and response service.
- Binding: Direction to use the SOAP HTTP transport protocol.
- Service: Service available at http://localhost:8080/HelloWorld/HelloService

2. Write a program the service provider can be implement a single get price () and get product () operation

getprice()

```
package com.ecerami.soap.examples;

import java.util.Hashtable;
/**
 * A Sample SOAP Service
 * Provides Current Price for requested Stockkeeping Unit (SKU)
 */
public class PriceService {
 protected Hashtable products;
 /**
 * Zero Argument Constructor
 * Load product database with two sample products
 */
 public PriceService ( ) {
 products = new Hashtable( );
 // Red Hat Linux
 products.put("A358185", new Double (54.99));
 // McAfee PGP Personal Privacy
 products.put("A358565", new Double (19.99));
 }
  /**
 * Provides Current Price for requested SKU
 * In a real-setup, this method would connect to
 * a price database. If SKU is not found, method
 * will throw a PriceException.
 */
 public double getPrice (String sku)
 throws ProductNotFoundException {
 Double price = (Double) products.get(sku);
 if (price == null) {
 throw new ProductNotFoundException ("SKU: "+sku+" not found");
 }
 return price.doubleValue( );
 }
}
```

To generate a WSDL file for this class, run the following command:

java2wsdl com.ecerami.soap.examples.PriceService -s -e http://localhost:
8080/soap/servlet/rpcrouter -n urn:examples:priceservice

The -s option directs GLUE to create a SOAP binding; the -e option specifies the address of our service; and the -n option specifies the namespace URN for the service. GLUE will generate a PriceService.wsdl file.

Getproduct ()

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="ProductService"
 targetNamespace="http://www.ecerami.com/wsdl/ProductService.wsdl"
 xmlns="http://schemas.xmlsoap.org/wsdl/"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:tns="http://www.ecerami.com/wsdl/ProductService.wsdl"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsd1="http://www.ecerami.com/schema">
```



```
 <types>
 <xsd:schema
 targetNamespace="http://www.ecerami.com/schema"
 xmlns="http://www.w3.org/2001/XMLSchema">
 <xsd:complexType name="product">
 <xsd:sequence>
 <xsd:element name="name" type="xsd:string"/>
 <xsd:element name="description" type="xsd:string"/>
 <xsd:element name="price" type="xsd:double"/>
 <xsd:element name="SKU" type="xsd:string"/>
 </xsd:sequence>
 </xsd:complexType>
 </xsd:schema>
 </types>

 <message name="getProductRequest">  <part name="sku" type="xsd:string"/>
 </message>

 <message name="getProductResponse">
 <part name="product" type="xsd1:product"/>
 </message>
```

```
<portType name="Product_PortType">
<operation name="getProduct">
<input message="tns:getProductRequest"/>
<output message="tns:getProductResponse"/>
</operation>
</portType>

<binding name="Product_Binding" type="tns:Product_PortType">
<soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
<operation name="getProduct">
<soap:operation soapAction="urn:examples:productservice"/>
<input>
<soap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:examples:productservice"
use="encoded"/>
</input>
<output>
<soap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:examples:productservice" use="encoded"/>
</output>
</operation>
</binding>

<service name="Product_Service">
<port name="Product_Port" binding="tns:Product_Binding">
<soap:address
location="http://localhost:8080/soap/servlet/rpcrouter"/>
</port>
</service>
</definitions>
```

3. **Write a program to implement the operation can receive request and will return a response in two ways.**
   a) One - Way operation
   b) Request –Response

The **<portType>** element is the most important WSDL element.

WSDL - The <portType> Element

The <portType> element defines **a web service**, the **operations** that can be performed, and the **messages** that are involved.

<portType> defines the connection point to a web service. It can be compared to a function library (or a module, or a class) in a traditional programming language. Each operation can be compared to a function in a traditional programming language.

Operation Types

The request-response type is the most common operation type, but WSDL defines four types:

Type	Definition
One-way	The operation can receive a message but will not return a response
Request-response	The operation can receive a request and will return a response
Solicit-response	The operation can send a request and will wait for a response
Notification	The operation can send a message but will not wait for a response

One-Way Operation

A one-way operation example:

```
<message name="newTermValues">
        <part name="term" type="xs:string"/>
        <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
     <operation name="setTerm">
              <input name="newTerm" message="newTermValues"/>
    </operation>
</portType >
```

In the example above, the portType "glossaryTerms" defines a one-way operation called "setTerm".
The "setTerm" operation allows input of new glossary terms messages using a "newTermValues" message with the input parameters "term" and "value". However, no output is defined for the operation.

Request-Response Operation

A request-response operation example:

```
<message name="getTermRequest">
        <part name="term" type="xs:string"/>
</message>

<message name="getTermResponse">
        <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
        <operation name="getTerm">
                <input message="getTermRequest"/>
                <output message="getTermResponse"/>
         </operation>
</portType>
```

In the example above, the portType "glossaryTerms" defines a request-response operation called "getTerm".
The "getTerm" operation requires an input message called "getTermRequest" with a parameter called "term", and will return an output message called "getTermResponse" with a parameter called "value".

4. Write a program to implement to create a simple web service that converts the temperature from Fahrenheit to Celsius (using HTTP Post Protocol)

- Creating a TemperatureApp Web Service
  1. Open Netbeans 7.4
  2. Open New Project → Java Web → Web Application
  3. Type Project Name **TemperatureApp**
  4. Right Click on Source Packages and add new web service **TempConvertWS** with package **org.me.temperature**.
  5. Open **TempConvertWS.java** which is Source packages → org.me.temperature and create web service as follows:

```
package org.me.temperature;

import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebParam;

@WebService(serviceName = "TempConvertWS")
public class TempConvertWS {

    @WebMethod(operationName = "FtoC")
    public float FtoC(@WebParam(name = "parameter") float faren) {
        float cel = (float)5/9*(faren-32);
        return cel;
    }
}
```

  6. Now Right Click on Project Name TemperatureApp, Build and then Deploy.
  7. Test the web service
     Expand web services and Right click TempConvertWS and Select Test Web Service
     Or use the following url in browser.
     http://localhost:8080/Temperature/TempConvertWS?Tester

     Now This Web Service TempConvertWS can be consumed as Web Based Service or Windows application based (Stand alone) service.

5. Write a program to implement business UDDI Registry entry

UDDI stands for Universal Description, Discovery, and Integration. The UDDI Project is an industry initiative aims to enable businesses to quickly, easily, and dynamically find and carry out transactions with one another.

A populated UDDI registry contains cataloged information about businesses; the services that they offer; and communication standards and interfaces they use to conduct transactions.

Built on the Simple Object Access Protocol (SOAP) data communication standard, UDDI creates a global, platform-independent, open architecture space that will benefit businesses.

The UDDI registry can be broadly divided into two categories:

UDDI and Web Services

UDDI and Business Registry

For details about the UDDI data structure, see UDDI Data Structure.

UDDI and Web Services

The owners of Web services publish them to the UDDI registry. Once published, the UDDI registry maintains pointers to the Web service description and to the service.

The UDDI allows clients to search this registry, find the intended service, and retrieve its details. These details include the service invocation point as well as other information to help identify the service and its functionality.

Web service capabilities are exposed through a programming interface, and usually explained through Web services Description Language (WSDL). In a typical publish-and-inquire scenario, the provider publishes its business; registers a service under it; and defines a binding template with technical information on its Web service. The binding template also holds reference to one or several tModels, which represent abstract interfaces implemented by the Web service. The tModels might have been uniquely published by the provider, with information on the interfaces and URL references to the WSDL document.

A typical client inquiry may have one of two objectives:

- To find an implementation of a known interface. In other words, the client has a tModel ID and seeks binding templates referencing that tModel.

- To find the updated value of the invocation point (that is., access point) of a known binding template ID

UDDI and Business Registry

As a Business Registry solution, UDDI enables companies to advertise the business products and services they provide, as well as how they conduct business transactions on the Web. This use of UDDI complements business-to-business (B2B) electronic commerce.

The minimum required information to publish a business is a single business name. Once completed, a full description of a business entity may contain a wealth of information, all of which helps to advertise the business entity and its products and services in a precise and accessible manner.

A Business Registry can contain:

- Business Identification - Multiple names and descriptions of the business, comprehensive contact information, and standard business identifiers such as a tax identifier.
- Categories - Standard categorization information (for example a D-U-N-S business category number).
- Service Description - Multiple names and descriptions of a service. As a container for service information, companies can advertise numerous services, while clearly displaying the ownership of services. The bindingTemplate information describes how to access the service.
- Standards Compliance - In some cases it is important to specify compliance with standards. These standards might display detailed technical requirements on how to use the service.
- Custom Categories - It is possible to publish proprietary specifications (tModels) that identify or categorize businesses or services.

UDDI Data Structure

The data structure within UDDI consists of four constructions: a businessEntity structure, a businessService structure, a bindingTemplate structure and a tModelstructure.

```
import java.io.*;
importjava.util.*;
 public class UDDISoapClient
{
   // Default values used if no command line parameters are set
private static final String DEFAULT_HOST_URL =
                        "http://localhost:8080/wasp/uddi/inquiry/";
private static final String DEFAULT_DATA_FILENAME   = "./Default.xml";

   // In the SOAP chapter, we used "urn:oreilly:jaws:samples",
   // but Systinet UDDI requires this to be blank.
private static final String URI                  = "";
private String m_hostURL;
private String m_dataFileName;

publicUDDISoapClient(String hostURL, String dataFileName) throws Exception
{
m_hostURL = hostURL;
m_dataFileName    = dataFileName;

System.out.println(  );
System.out.println("_____");
System.out.println("Starting UDDISoapClient:");
System.out.println("    host url       = " + m_hostURL);
System.out.println("    data file      = " + m_dataFileName);
System.out.println("_____");
System.out.println(  );
   }

public void sendSOAPMessage(  ) {
try {
         // Get soap body to include in the SOAP envelope from FILE
FileReaderfr = new FileReader (m_dataFileName);
javax.xml.parsers.DocumentBuilderxdb =
org.apache.soap.util.xml.XMLParserUtils.getXMLDocBuilder(  );
org.w3c.dom.Document doc =
xdb.parse (new org.xml.sax.InputSource (fr));
```

```
if (doc == null) {
throw new org.apache.soap.SOAPException
                (org.apache.soap.Constants.FAULT_CODE_CLIENT, "parsing error");
        }

        // Create a vector for collecting the body elements
        Vector bodyElements = new Vector(  );

        // Parse XML element as soap body element
bodyElements.add(doc.getDocumentElement (  ));
        // Create the SOAP envelope
org.apache.soap.Envelope envelope = new org.apache.soap.Envelope(  );
envelope.declareNamespace("idoox", "http://idoox.com/uddiface");
envelope.declareNamespace("ua", "http://idoox.com/uddiface/account");
envelope.declareNamespace("config",
          "http://idoox.com/uddiface/config");
envelope.declareNamespace("attr", "http://idoox.com/uddiface/attr");
envelope.declareNamespace("fxml", "http://idoox.com/uddiface/formxml");
envelope.declareNamespace("inner", "http://idoox.com/uddiface/inner");
envelope.declareNamespace("", "http://idoox.com/uddiface/inner");
envelope.declareNamespace("uddi", "urn:uddi-org:api_v2");
        //
        // NO SOAP HEADER ELEMENT AS SYSTINET WASP DOES NOT REQUIRE IT
        //
        // Create the SOAP body element
org.apache.soap.Body body = new org.apache.soap.Body(  );
body.setBodyEntries(bodyElements);
envelope.setBody(body);
        // Build and send the Message.
org.apache.soap.messaging.Messagemsg =
neworg.apache.soap.messaging.Message(  );
msg.send (new java.net.URL(m_hostURL), URI, envelope);
System.out.println("Sent SOAP Message with Apache HTTP SOAP Client.");
        // Receive response from the transport and dump it to the screen
System.out.println("Waiting for response....");
org.apache.soap.transport.SOAPTransportst = msg.getSOAPTransport (  );
BufferedReaderbr = st.receive (  );

if(line == null) {
System.out.println("HTTP POST was unsuccessful. \n");
```

```
          } else {
while (line != null) {
System.out.println (line);
line = br.readLine(  );
              }
           }
        /////
        // Version in examples has XML pretty printing logic here.
        ////
        } catch(Exception e) {
e.printStackTrace(  );
        }
    }
    //
    // NOTE: the remainder of this deals with reading arguments
    //
    /** Main program entry point. */
public static void main(String args[]) {
     // Not Relevant
    }
}
```

Output:

```
Starting UDDISoapClient:
hosturl      = http://localhost:8080/wasp/uddi/inquiry/
data file     = Ch6_FindBusiness.xml
Sent SOAP Message with Apache HTTP SOAP Client.
Waiting for response....
 <?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelopexmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
     <SOAP-ENV:Body>
          <businessListxmlns="urn:uddi-org:api_v2" generic="2.0" operator="SYSTINET">
                    <businessInfos>
                       <businessInfobusinessKey="892ac280-c16b-11d5-85ad-801eef208714">
                            <name xml:lang="en">
                                              Demi Credit
                            </name>
                          <description xml:lang="en">
                              A smaller demo credit agency used for illustrating UDDI inquiry.
                         </description>
                       <serviceInfos>
```

```
                    <serviceInfo serviceKey="860eca90-c16d-11d5-85ad-801eef208714"
                        businessKey="9a26b6e0-c15f-11d5-85a3-801eef208714">
                         <name xml:lang="en">
                              DCAmail
                         </name>
                    </serviceInfo>
               </serviceInfos>
          </businessInfo>
     </businessInfos>
</businessList>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

6. Write a program to implement

a) Web based service consumer

b) Windows application based web service consumer

- Creating a Calculator Web Service
  1. Open Netbeans 7.4
  2. Open New Project → Java Web → Web Application
  3. Type Project Name **CalculatorApp**
  4. Right Click on Source Packages and add new web service **CalculatorWS** with package **org.me.calculator**.
  5. Open CalculatorWS.java which is Source packages → org.me.calculator and create web service as follows:

```
package org.me.calculator;
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebParam;
@WebService(serviceName = "CalculatorWS")
public class CalculatorWS {
    @WebMethod(operationName = "add")
    public int add(@WebParam(name = "a") int a,@WebParam(name = "b") int b) {
        return a+b;
    }
}
```

  6. Now Right Click on Project Name CalculatorApp, Build and Deploy.
  7. Test the web service
     Expand web services and Right click CalculatorWS and Select Test Web Service
     Or use the following url in browser.
     http://localhost:8080/CalculatorApp/CalculatorWS?Tester
- Now This Web Service CalculatorWS can Consumed as Web Based Service or Windows application based (Stand-alone) service.

a) Web based service consumer

`CalcClientServlet` is a servlet that, like the Java client, calls the `add` method of the web service. Like the application client, it makes this call through a port.

1. Open New Project → Java Web → Web Application
2. Project Name `CalculatorClientApp`
3. Expand Web Pages → Open index.html and add a form in body tag

```
<body>
    <form name = "submit" action ="CalcClientServlet" >

     Number 1:<input type ="text" name ="num1" value = "5">
     <br/>
     Number 2:<input type ="text" name ="num2" value = "6">
     <br/>
     <input type="submit" value = "Add" >
  </form>
</body>
```

4. Now Right Click on project CalculatorClientApp and add new Web Service Client
5. Provide WSDL information with provided options.
   Option 1 Project → browse and Select and Select CalculatorApp → CalculatorWS
   Also Provide package name `org.me.calculator.client`
6. Right Click on Source packages and add new Servlet `CalcClientServlet` and code as below.

```
package org.me.calculator.client;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.xml.ws.WebServiceRef;

@WebServlet(name = "CalcClientServlet", urlPatterns = {"/CalcClientServlet"})
public class CalcClientServlet extends HttpServlet {
    @WebServiceRef(wsdlLocation =
                  "http://localhost:8080/CalculatorApp1/CalculatorWS?wsdl")
    public org.me.calculator.client.CalculatorWS_Service service;

protected void processRequest(HttpServletRequest request,
      HttpServletResponse response)throws ServletException, IOException {

        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {

            CalculatorWS port = service.getCalculatorWSPort();

            int i = Integer.parseInt(request.getParameter("num1"));
            int j = Integer.parseInt(request.getParameter("num2"));
            int res = port.add(i, j);

            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet CalcClientServlet</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Servlet CalcClientServlet at " +
                          request.getContextPath() + "</h1>");
            out.println("<br/> Result <br/>" + i + " + " + j + " = " + res );
            out.println("</body>");
            out.println("</html>");
        } // end of try block
    } // end of processRequest method
} // end of CalcClientServlet
```

7. Now Build and then Deploy
8. Finally Run the CalculatorClientApp, which opens index.html in default web browser.

b) Windows application based web service consumer

1. Open New Project → Java → Java Application
2. Project Name CalcclientApp

3. Now Right Click on project CalcClientApp and add new Web Service Client
4. Provide WSDL information with provided options.
   Option 1 Project → browse and Select and Select CalculatorApp → CalculatorWS
   Also provide package name org.me.calcclient
5. Open Source packages and add new Servlet `CalcClientApp.java` and code as below.

```
package calcclient;
public class CalcClientApp {
  public static void main(String[] args) {
        try {
              System.out.println("Result = " + add(7,5));
          }catch(Exception ex) {
                  System.out.println(ex);
            }
   }
  private static int add(int i, int j) {
   org.me.calculator.CalculatorWSService service = new
                                        org.me.calculator.CalculatorWSService();
    org.me.calculator.CalculatorWS port = service.getCalculatorWSPort();
    return port.add(i, j);
  }
}
```

6. Now Build and then Deploy
7. Finally Run the CalclientApp.

Getting Started with JAX-WS Web Services

Java API for XML Web Services (JAX-WS), JSR 224, is an important part of the Java EE platform. A follow-up to the release of Java API for XML-based RPC 1.1(JAX-RPC), JAX-WS simplifies the task of developing web services using Java technology. It addresses some of the issues in JAX-RPC 1.1 by providing support for multiple protocols such as SOAP 1.1, SOAP 1.2, XML, and by providing a facility for supporting additional protocols along with HTTP. JAX-WS uses JAXB 2.0 for data binding and supports customizations to control generated service endpoint interfaces. With its support for annotations, JAX-WS simplifies web service development and reduces the size of runtime JAR files.
This document demonstrates the basics of using the IDE to develop a JAX-WS web service. After you create the web service, you write three different web service clients that use the web service over a network, which is called "consuming" a web service. The three clients are a Java class in a Java SE application, a servlet, and a JSP page in a web application. A more advanced tutorial focusing on clients is Developing JAX-WS Web Service Clients.
We use the following software and resources.

Software or Resource	Version Required
NetBeans IDE	Java EE download bundle
Java Development Kit (JDK)	JDK 7 or JDK 8
Java EE-compliant web or application server	GlassFish Server Open Source Edition Oracle WebLogic Server

Note: The GlassFish server can be installed with the Java EE distribution of NetBeans IDE. Alternatively, you can visit the the GlassFish server downloads page or theApache Tomcat downloads page.
Important: Java EE projects require GlassFish Server or Oracle WebLogic Server 12c.

Enabling Access to External Schema

You need to enable the IDE and the GlassFish Server to access external schema to parse the WSDL file of the web service. To enable access you need to modify the configuration files of the IDE and the GlassFish Server. For more details, see the FAQ How to enable parsing of WSDL with an external schema?

Configuring the IDE

To generate a web service client in the IDE from a web service or WSDL file you need to modify the IDE configuration file (netbeans.conf) to add the following switch tonetbeans_default_options.

```
-J-Djavax.xml.accessExternalSchema=all
```

For more about locating and modifying the netbeans.conf configuration file, see Netbeans Conf FAQ.

Configuring the GlassFish Server

If you are deploying to the GlassFish Server you need to modify the configuration file of the GlassFish Server (domain.xml) to enable the server to access external schemas to parse the wsdl file and generate the test client. To enable access to external schemas, open the GlassFish configuration file (***GLASSFISH_INSTALL***/glassfish/domains/domain1/config/domain.xml) and add the following JVM option element (in **bold**). You will need to restart the server for the change to take effect.

```
</java-config>
  ...
  <jvm-options>-Djavax.xml.accessExternalSchema=all</jvm-options>
</java-config>
```

Creating a Web Service Using NetBeans 7.4

The goal of this exercise is to create a project appropriate to the deployment container that you decide to use. Once you have a project, you will create a web service in it.

Choosing a Container

You can either deploy your web service in a web container or in an EJB container. This depends on your choice of implementation. If you are creating a Java EE application, use a web container in any case, because you can put EJBs directly in a web application. For example, if you plan to deploy to the Tomcat Web Server, which only has a web container, create a web application, not an EJB module.

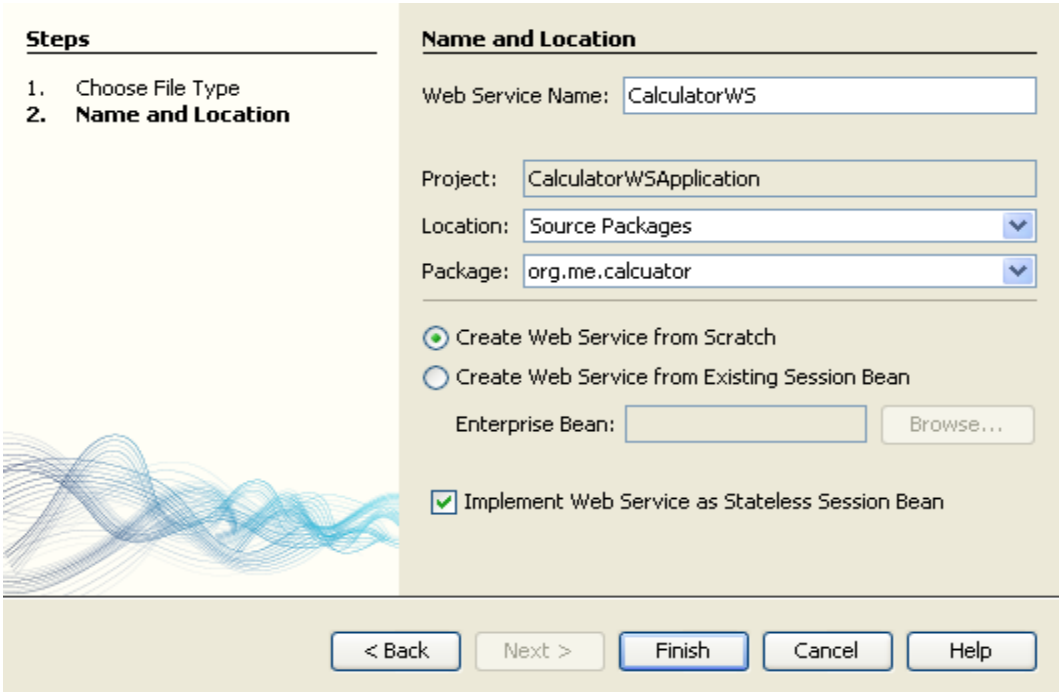
1. Choose File > New Project (Ctrl-Shift-N on Linux and Windows, ⌘-Shift-N on MacOS). Select Web Application from the Java Web category or EJB Module from the Java EE category.

   Note. You can create a JAX-WS web service in a Maven project. Choose File > New Project (Ctrl-Shift-N on Linux and Windows, ⌘-Shift-N on MacOS) and then Maven Web Application or Maven EJB module from the Maven category. If you haven't used Maven with NetBeans before, see Maven Best Practices.
2. Name the project CalculatorWSApplication. Select a location for the project. Click Next.
3. Select your server and Java EE version and click Finish.

   Note. To use the Oracle WebLogic server, register the server with the IDE. Also, if you are using the WebLogic server, watch the screencast on Deploying a Web Application to Oracle WebLogic.

Creating a Web Service from a Java Class

1. Right-click the CalculatorWSApplication node and choose New > Web Service.
2. Name the web service CalculatorWS and type org.me.calculator in Package. Leave Create Web Service from Scratch selected.
3. If you are creating a Java EE project on GlassFish or WebLogic, select Implement Web Service as a Stateless Session Bean.



4. Click Finish. The Projects window displays the structure of the new web service and the source code is shown in the editor area.

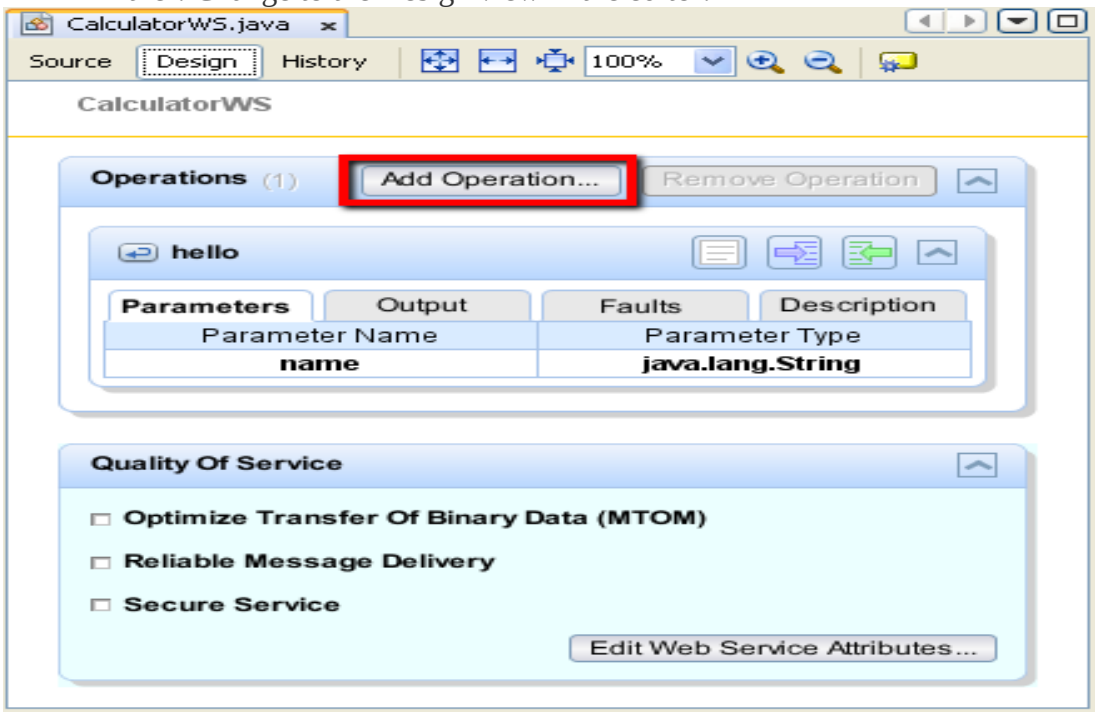
Adding an Operation to the Web Service

The goal of this exercise is to add to the web service an operation that adds two numbers received from a client. The NetBeans IDE provides a dialog for adding an operation to a web service. You can open this dialog either in the web service visual designer or in the web service context menu.

Warning: The visual designer is not available in Maven projects.

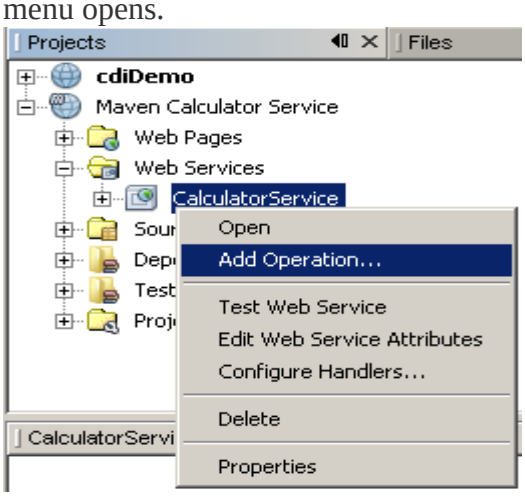
To add an operation to the web service:

- Either: Change to the Design view in the editor.



Or:

Find the web service's node in the Projects window. Right-click that node. A context menu opens.



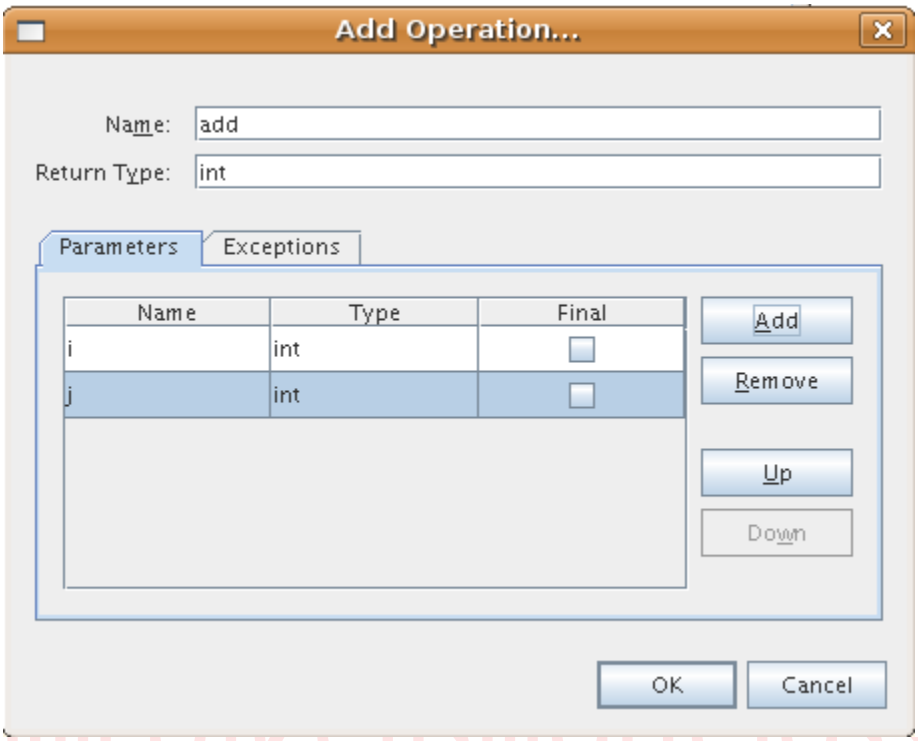
Click Add Operation in either the visual designer or the context menu. The Add Operation dialog opens.

In the upper part of the Add Operation dialog box, type add in Name and type int in the Return Type drop-down list.

In the lower part of the Add Operation dialog box, click Add and create a parameter of type int named i.

Click Add again and create a parameter of type int called j.

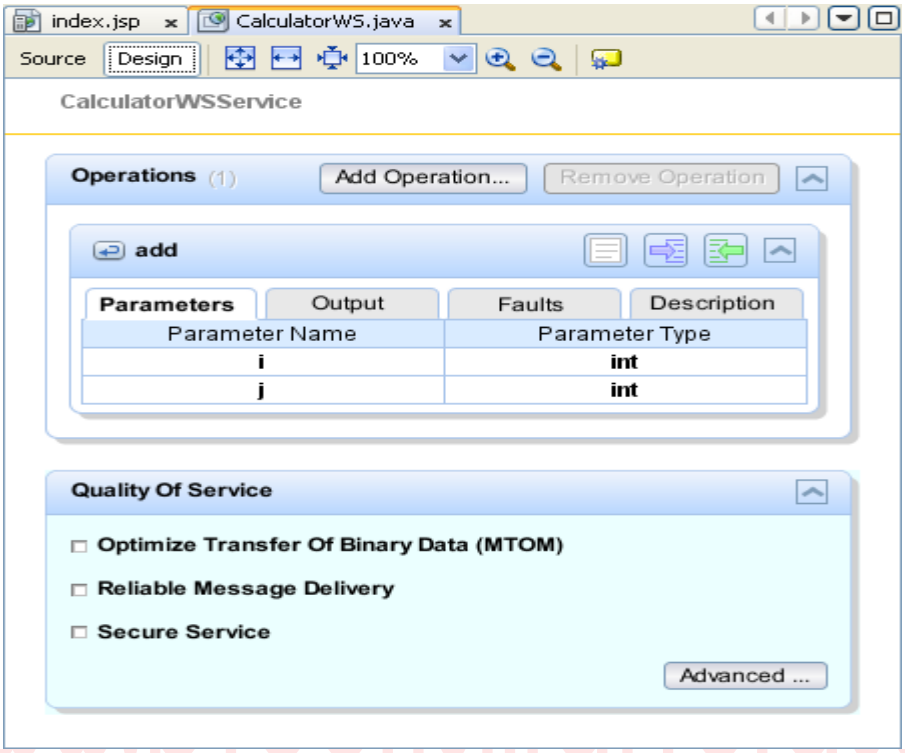
You now see the following:



Click OK at the bottom of the Add Operation dialog box. You return to the editor.

Remove the default hello operation, either by deleting the hello() method in the source code or by selecting the hellooperation in the visual designer and clicking Remove Operation.

The visual designer now displays the following:



Click Source and view the code that you generated in the previous steps. It differs whether you created the service as an Java EE stateless bean or not. Can you see the difference in the screenshots below? (A Java EE 6 or Java EE 7 service that is not implemented as a stateless bean resembles a Java EE 5 service.)

```
package org.me.calculator;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;

/**
 *
 * @author gw152771
 */
@WebService()
public class CalculatorWS {

/**
     * Web service operation
     */
    @WebMethod(operationName = "add")
    public int add(@WebParam(name = "i")
    int i, @WebParam(name = "j")
    int j) {
//TODO write your implementation code here:
        return 0;
    }

}
```

```
package org.me.calcuator;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
import javax.ejb.Stateless;

/**
 *
 * @author jeff
 */
@WebService()
@Stateless()
public class CalculatorWS {

    /**
     * Web service operation
     */
    @WebMethod(operationName = "add")
    public int add(@WebParam(name = "i")
    int i, @WebParam(name = "j")
    int j) {
        //TODO write your implementation code
        return 0;
    }
}
```

Note. In NetBeans IDE 7.3 and 7.4 you will notice that in the generated @WebService annotation the service name is specified explicitly:
`@WebService(serviceName = "CalculatorWS").`

In the editor, extend the skeleton `add` operation to the following (changes are in bold):

```
@WebMethod
public int add(@WebParam(name = "i") int i, @WebParam(name = "j")int j)
{
        int k = i + j;
        return k;
 }
```

As you can see from the preceding code, the web service simply receives two numbers and then returns their sum. In the next section, you use the IDE to test the web service.

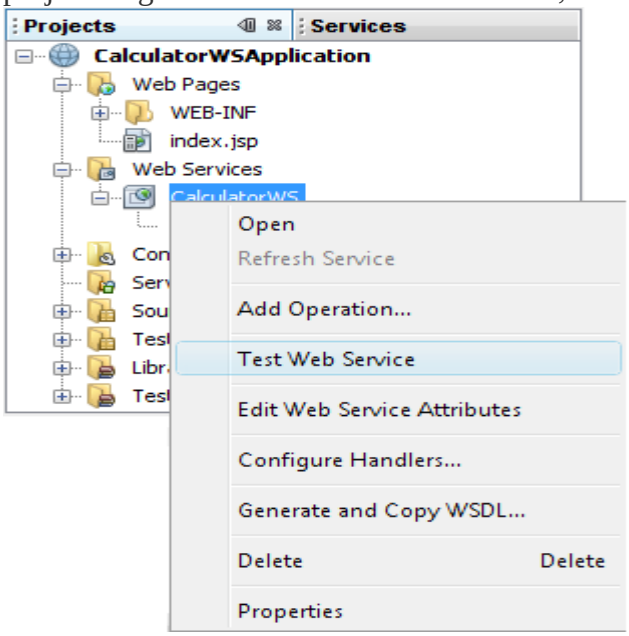
Deploying and Testing the Web Service

After you deploy a web service to a server, you can use the IDE to open the server's test client, if the server has a test client. The GlassFish and WebLogic servers provide test clients.
If you are using the Tomcat Web Server, there is no test client. You can only run the project and see if the Tomcat Web Services page opens. In this case, before you run the project, you need to make the web service the entry point to your application. To make the web service the entry point to your application, right-click the CalculatorWSApplication project node and choose Properties. Open the Run properties and type /CalculatorWS in the Relative URL field. Click OK. To run the project, right-click the project node again and select Run.

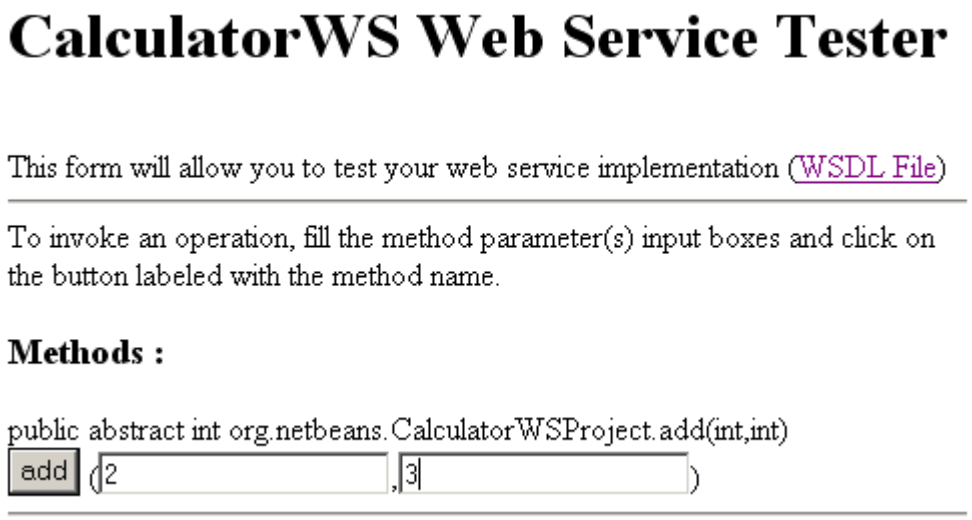
To test successful deployment to a GlassFish or WebLogic server:

1. Right-click the project and choose Deploy. The IDE starts the application server, builds the application, and deploys the application to the server. You can follow the progress of these operations in the CalculatorWSApplication (run-deploy) and the GlassFish server or Tomcat tabs in the Output view.

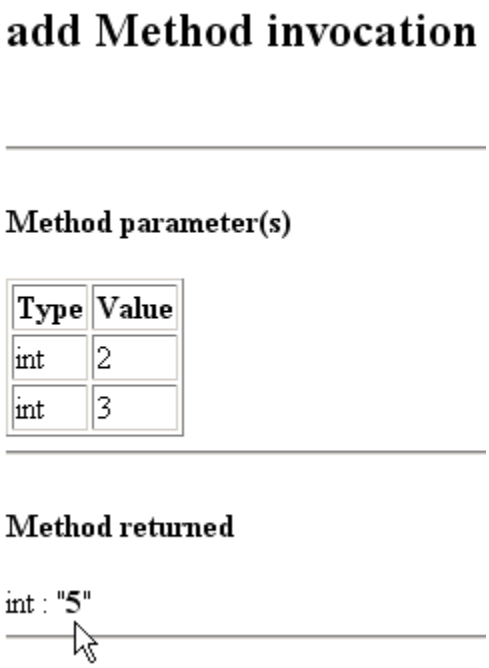
2. In the IDE's Projects tab, expand the Web Services node of the CalculatorWSApplication project. Right-click the CalculatorWS node, and choose Test Web Service.



The IDE opens the tester page in your browser, if you deployed a web application to the GlassFish server. For the Tomcat Web Server and deployment of EJB modules, the situation is different:
If you deployed to the GlassFish server, type two numbers in the tester page, as shown below:



The sum of the two numbers is displayed:



Samples

You can open a complete Java EE stateless bean version of the Calculator service by choosing File > New Project (Ctrl-Shift-N on Linux and Windows, ⌘-Shift-N on MacOS) and navigating to Samples > Web Services > Calculator (EE6).
A Maven Calculator Service and a Maven Calculator Client are available in Samples > Maven.

Consuming the Web Service

Now that you have deployed the web service, you need to create a client to make use of the web service's add method. Here, you create three clients— a Java class in a Java SE application, a servlet, and a JSP page in a web application.
Note: A more advanced tutorial focusing on clients is Developing JAX-WS Web Service Clients.

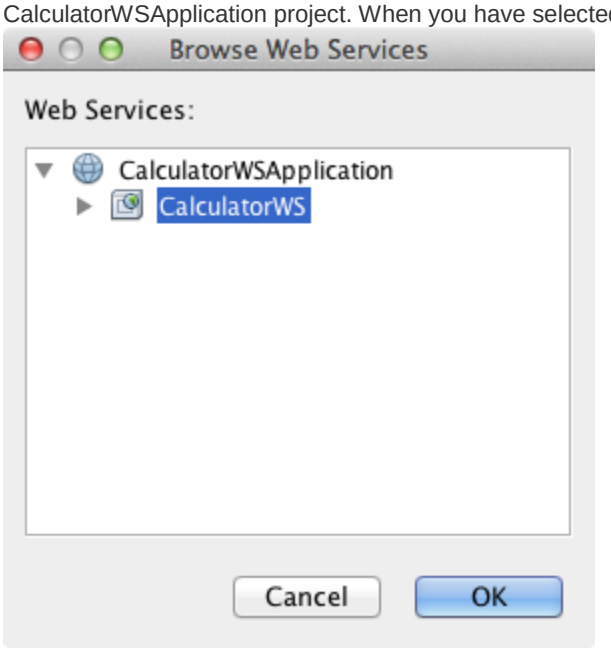
Client 1: Java Class in Java SE Application

In this section, you create a standard Java application. The wizard that you use to create the application also creates a Java class. You then use the IDE's tools to create a client and consume the web service that you created at the start of this tutorial.

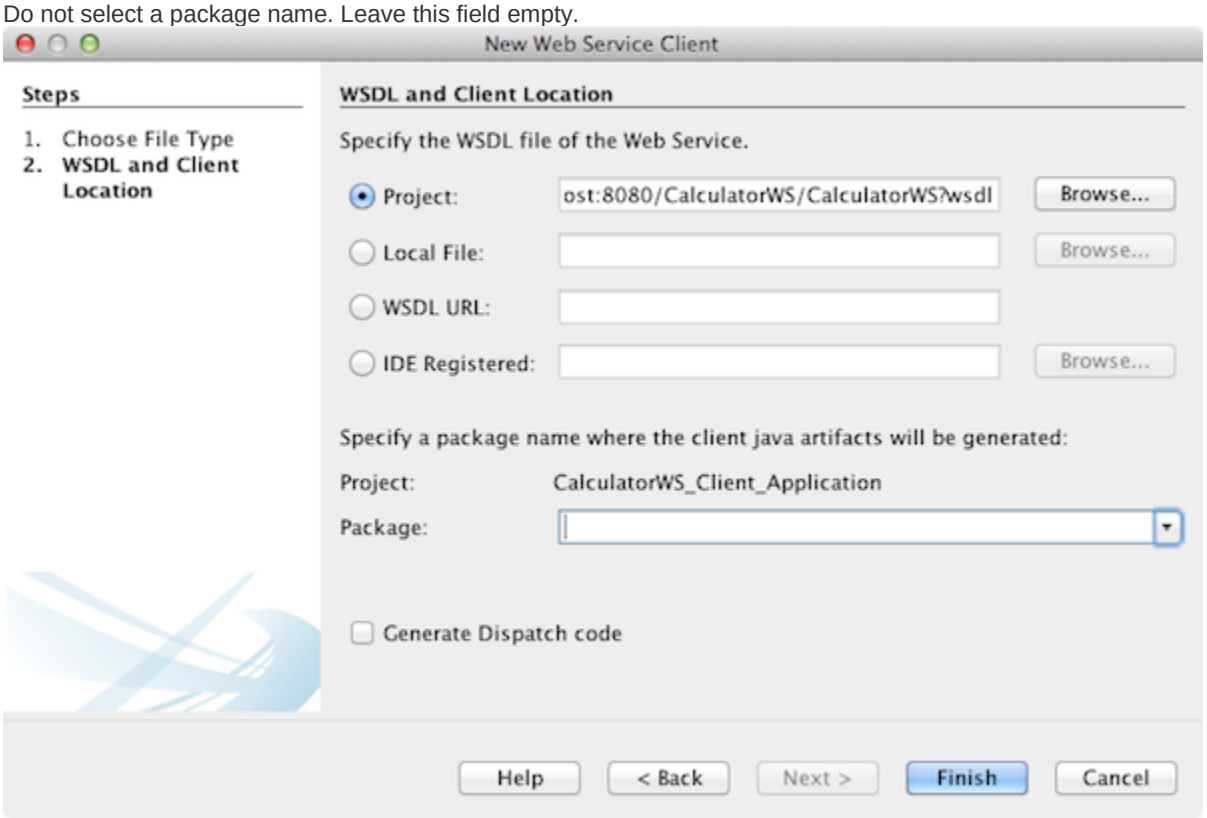
1. Choose File > New Project (Ctrl-Shift-N on Linux and Windows, ⌘-Shift-N on MacOS). Select Java Application from the Java category. Name the project `CalculatorWS_Client_Application`. Leave Create Main Class selected and accept all other default settings. Click Finish.

2. Right-click the `CalculatorWS_Client_Application` node and choose New > Web Service Client. The New Web Service Client wizard opens.

3. Select Project as the WSDL source. Click Browse. Browse to the CalculatorWS web service in the CalculatorWSApplication project. When you have selected the web service, click OK.

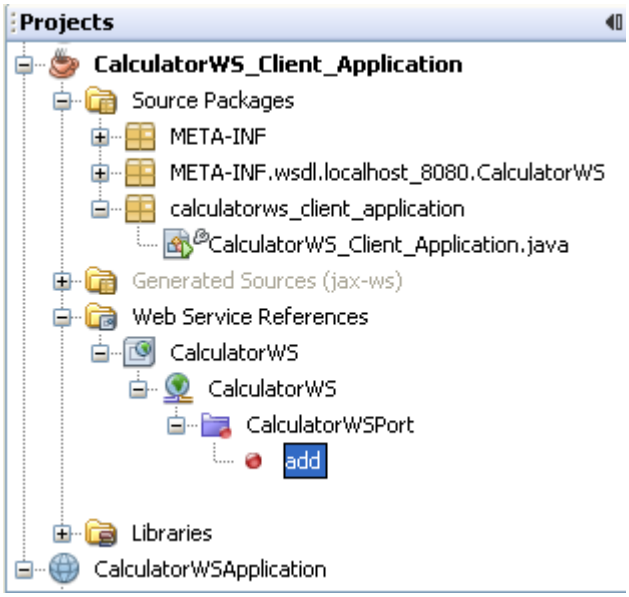


4. Do not select a package name. Leave this field empty.

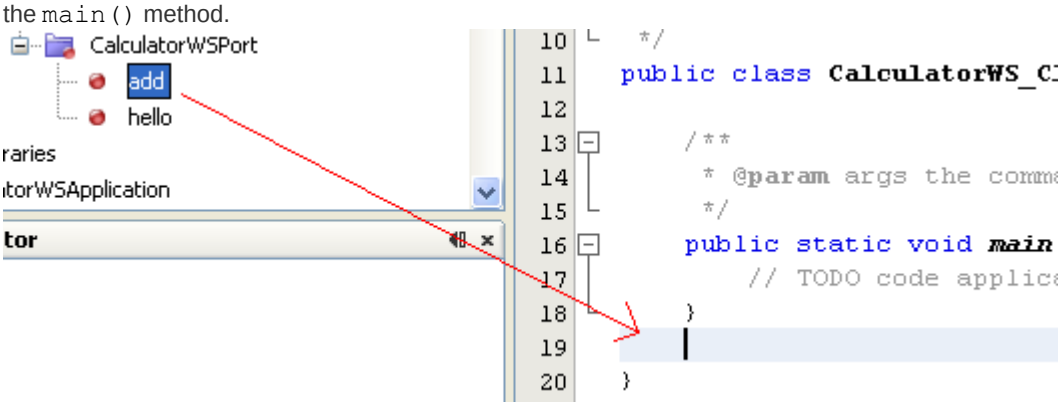


5. Leave the other settings at default and click Finish.

The Projects window displays the new web service client, with a node for the `add` method that you created:



6. Double-click your main class so that it opens in the Source Editor. Drag the `add` node below the `main()` method.



You now see the following:

```
public static void main(String[] args) {
    // TODO code application logic here
}
private static int add(int i, int j) {
    org.me.calculator.CalculatorWS_Service service = new
org.me.calculator.CalculatorWS_Service();
    org.me.calculator.CalculatorWS port =
service.getCalculatorWSPort();
    return port.add(i, j);
}
```

Note: Alternatively, instead of dragging the `add` node, you can right-click in the editor and then choose Insert Code > Call Web Service Operation.

7. In the `main()` method body, replace the TODO comment with code that initializes values for `i` and `j`, calls `add()`, and prints the result.

```
8.   public static void main(String[] args) {
        int i = 3;
        int j = 4;
        int result = add(i, j);
        System.out.println("Result = " + result);
    }
```

9. Surround the `main()` method code with a try/catch block that prints an exception.

```
10.  public static void main(String[] args) {
        try {
            int i = 3;
            int j = 4;
            int result = add(i, j);
            System.out.println("Result = " + result);
        } catch (Exception ex) {
            System.out.println("Exception: " + ex);
        }
    }
```

11. Right-click the project node and choose Run.

The Output window now shows the sum:

```
    compile:
    run:
    Result = 7
      BUILD SUCCESSFUL (total time: 1 second)
```

Client 2: Servlet in Web Application

In this section, you create a new web application, after which you create a servlet. You then use the servlet to consume the web service that you created at the start of this tutorial.

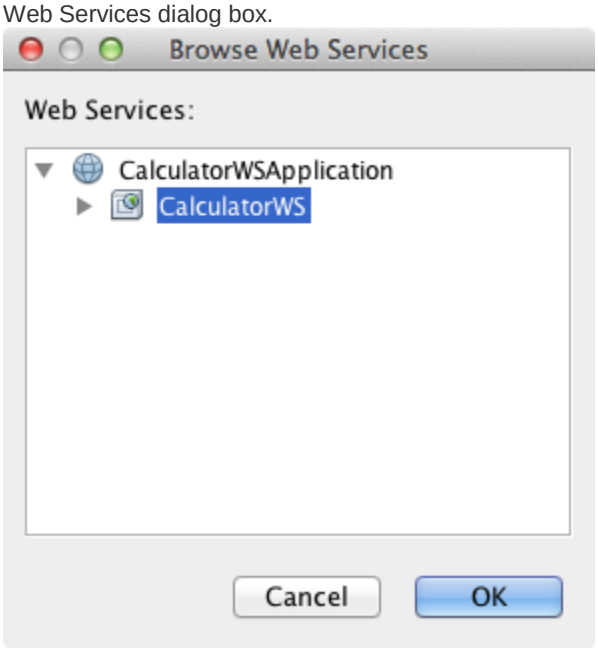
1. Choose File > New Project (Ctrl-Shift-N on Linux and Windows, ⌘Shift-N on MacOS). Select Web Application from the Java Web category. Name the project `CalculatorWSServletClient`. Click Next and then click Finish.

2. Right-click the `CalculatorWSServletClient` node and choose New > Web Service Client.

   The New Web Service Client wizard opens.

3. Select Project as the WSDL source and click Browse to open the Browse Web Services dialog box.

4. Select the CalculatorWS web service in the CalculatorWSApplication project. Click OK to close the Browse Web Services dialog box.



5. Confirm that the package name is empty in the New Web Service Client wizard and leave the other settings at the default value. Click Finish.

The Web Service References node in the Projects window displays the structure of your newly created client, including the `add`operation that you created earlier in this tutorial.

6. Right-click the `CalculatorWSServletClient` project node and choose New > Servlet. Name the servlet`ClientServlet` and place it in a package called `org.me.calculator.client`. Click Finish.

7. To make the servlet the entry point to your application, right-click the CalculatorWSServletClient project node and choose Properties. Open the Run properties and type `/ClientServlet` in the Relative URL field. Click OK.

8. If there are error icons for `ClientServlet.java`, right-click the project node and select Clean and Build.

9. In the `processRequest()` method, add some empty lines after this line:

```
    out.println("<h1>Servlet ClientServlet at " +
request.getContextPath () + "</h1>");
```

10. In the Source Editor, drag the `add` operation anywhere in the body of the `ClientServlet` class. The `add()` method appears at the end of the class code.

Note: Alternatively, instead of dragging the `add` node, you can right-click in the editor and then choose Insert Code > Call Web Service Operation.

```
private int add(int i, int j) {
    org.me.calculator.CalculatorWS port =
service.getCalculatorWSPort();
    return port.add(i, j);
```

```
}
```

11. Add code that initializes values for `i` and `j`, calls `add()`, and prints the result. The added code is in **boldface**:

```
protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
         throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    try {
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet ClientServlet</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Servlet ClientServlet at "
+request.getContextPath () + "</h1>");

       int i = 3;
    int j = 4;
    int result = add(i, j);
    out.println("Result = " + result);

        out.println("</body>");
        out.println("</html>");

    } finally {
   out.close();
  }
}
```

12. Surround the added code with a try/catch block that prints an exception.

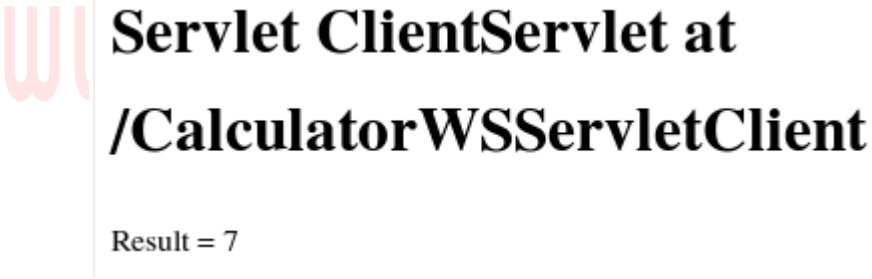
```
protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
         throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    try {
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet ClientServlet</title>");
        out.println("</head>");
        out.println("<body>");
```

```
        out.println("<h1>Servlet ClientServlet at " +
request.getContextPath () + "</h1>");
        try {
            int i = 3;
            int j = 4;
            int result = add(i, j);
            out.println("Result = " + result);
        } catch (Exception ex) {
     out.println("Exception: " + ex);
   }
        out.println("</body>");
        out.println("</html>");

    } finally {
        out.close();
    }
}
```

13. Right-click the project node and choose Run.

    The server starts, the application is built and deployed, and the browser opens, displaying the calculation result, as shown below:



Client 3: JSP Page in Web Application

In this section, you create a new web application and then consume the web service in the default JSP page that the Web Application wizard creates.

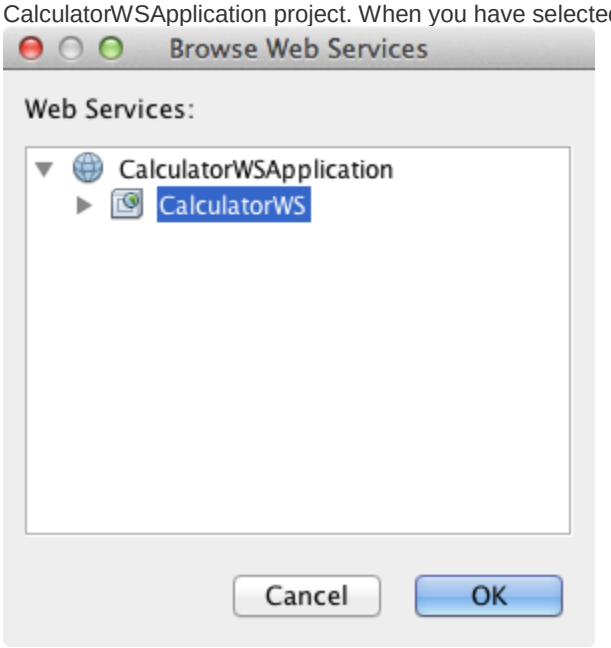
Note: If you want to run a JSP web application client on Oracle WebLogic, see Running a Java Server Faces 2.0 Application on WebLogic.

1. Choose File > New Project (Ctrl-Shift-N on Linux and Windows, ⌘Shift-N on MacOS). Select Web Application from the Java Web category. Name the project `CalculatorWSJSPClient`. Click Next and then click Finish.
2. Expand the Web Pages node under the project node and delete `index.html`.
3. Right-click the `Web Pages` node and choose New > JSP in the popup menu.

   If JSP is not available in the popup menu, choose New > Other and select JSP in the Web category of the New File wizard.
4. Type **index** for the name of the JSP file in the New File wizard. Click Finish.

5. Right-click the `CalculatorWSJSPClient` node and choose New > Web Service Client.

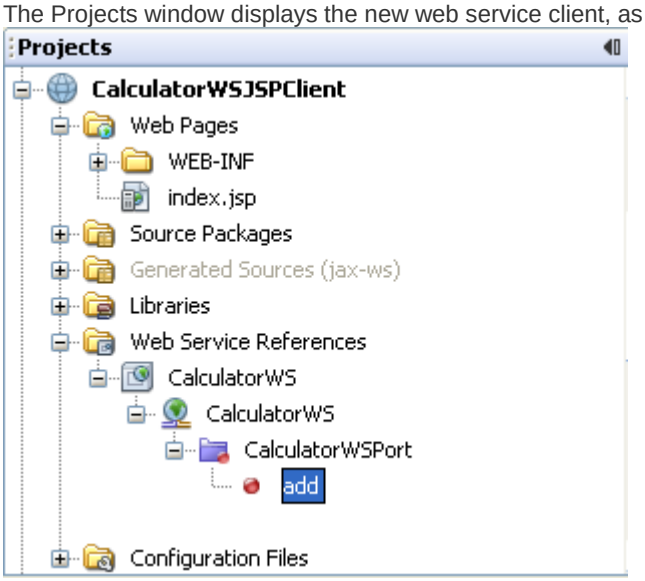
6. Select Project as the WSDL source. Click Browse. Browse to the CalculatorWS web service in the CalculatorWSApplication project. When you have selected the web service, click OK.



7. Do not select a package name. Leave this field empty.

8. Leave the other settings at default and click Finish.

The Projects window displays the new web service client, as shown below:



9. In the Web Service References node, expand the node that represents the web service. The `add` operation, which you will invoke from the client, is now exposed.

10. Drag the `add` operation to the client's `index.jsp` page, and drop it below the H1 tags. The code for invoking the service's operation is now generated in the `index.jsp` page, as you can see here:

```
<%
try {
    org.me.calculator.CalculatorWSService service = new
org.me.calculator.CalculatorWSService();
    org.me.calculator.CalculatorWS port =
service.getCalculatorWSPort();
     // TODO initialize WS operation arguments here
    int i = 0;
    int j = 0;
    // TODO process result here
    int result = port.add(i, j);
    out.println("Result = "+result);
} catch (Exception ex) {
    // TODO handle custom exceptions here
}
%>
```

Change the value for `i` and `j` from 0 to other integers, such as 3 and 4. Replace the commented out TODO line in the catch block with `out.println("exception" + ex);`.

11. Right-click the project node and choose Run.

The server starts, if it wasn't running already. The application is built and deployed, and the browser opens, displaying the calculation result: