Lesson 6

Java Web Services

1. Introduction

Web services today are frequently just *Application Programming Interfaces (API)* or *Web APIs* that can be accessed over a network, such as the Internet, and executed on a remote system hosting the requested services. In common usage the term refers to clients and servers that communicate over the Hypertext Transfer Protocol (HTTP) protocol used on the web. Such services tend to fall into one of two camps: Big Web Services and RESTful Web Services.

Big Web Services" use Extensible Markup Language (XML) messages that follow the Simple Object Access Protocol (SOAP) standard and have been popular with traditional enterprise. In such systems, there is often a machine-readable description of the operations offered by the service written in the Web Services Description Language (WSDL). The latter is not a requirement of a SOAP *endpoint*, but it is a prerequisite for automated client-side code generation in many Java and .NET SOAP frameworks (frameworks such as Spring, Apache Axis2 and Apache CXF being notable exceptions). Some industry organizations, such as the WS-I, mandate both SOAP and WSDL in their definition of a web service.

More recently, *REpresentational State Transfer (RESTful)* web services have been regaining popularity, particularly with Internet companies. By using the PUT, GET and DELETE HTTP methods, alongside POST, these are often better integrated with HTTP and web browsers than SOAP-based services. They do not require XML messages or WSDL service-API definitions.

2. Web service applications with NetBeans IDE

The Java API for XML Web Services (JAX-WS) is the current model for SOAP-based web services in Metro. JAX-WS is built on the earlier JAX-RPC model but uses specific Java EE 5 features, such as annotations, to simplify the task of developing web services. Because it uses SOAP for messaging, JAX-WS is transport neutral. It also supports a wide range of modular WS-* specifications, such as WS-Security and WS-ReliableMessaging.

When you create a web service client, you have the option of using either the JAX-WS or JAX-RPC model. This is because some older JAX-RPC services use a binding style that is not supported by JAX-WS. These services can only be consumed by JAX-RPC clients.

Metro web services are interoperable with Apache Axis2 web services. Apache Axis2 is an open-source implementation of the SOAP (Simple Object-Access Protocol) submission to the W3C. Axis2 not only supports SOAP 1.1 and SOAP 1.2, but it also has integrated support for RESTful web services.

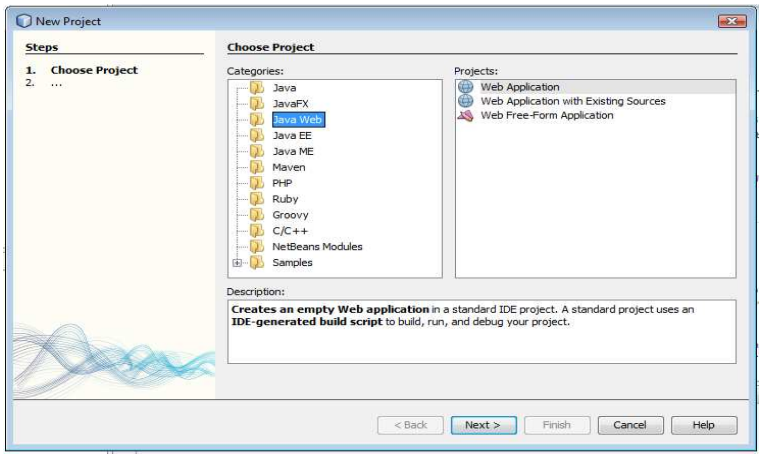
2.1. Creating a Web Service

Step 1: 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 6 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.

- Choose File > New Project (Ctrl-Shift-N). Select Web Application from the Java Web category or EJB Module from the Java EE category;

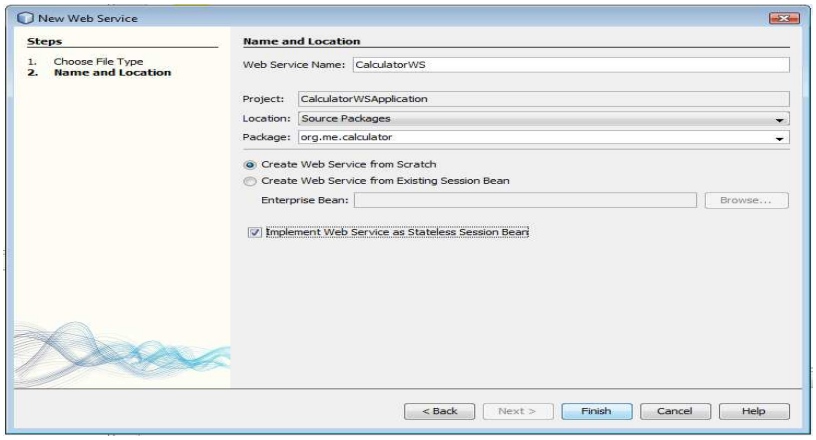


- Name the project CalculatorWSApplication;
- Click through the remaining pages and click Finish;

Step 2: Creating a Web Service from a Java Class

- Right-click the CalculatorWSApplication node and choose New -> Web Service;
- Name the web service CalculatorWS and type org.me.calculator in Package;

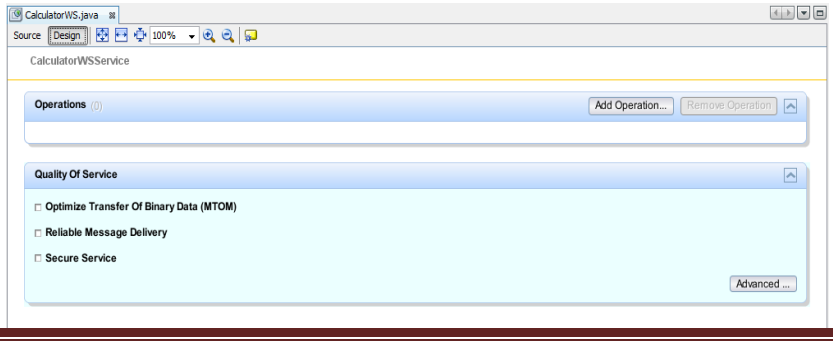
- If you are creating a Java EE 6 project, leave Create Web Service from Scratch selected, and select Implement Web Service as a Stateless Session Bean;



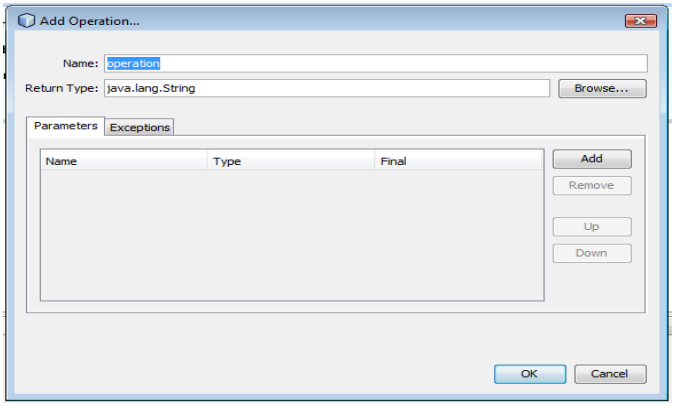
- Click Finish. The Projects window displays the structure of the new web service and the source code is shown in the editor area;

2.2. Designing the Web Service - Adding an Operation to the Web Service

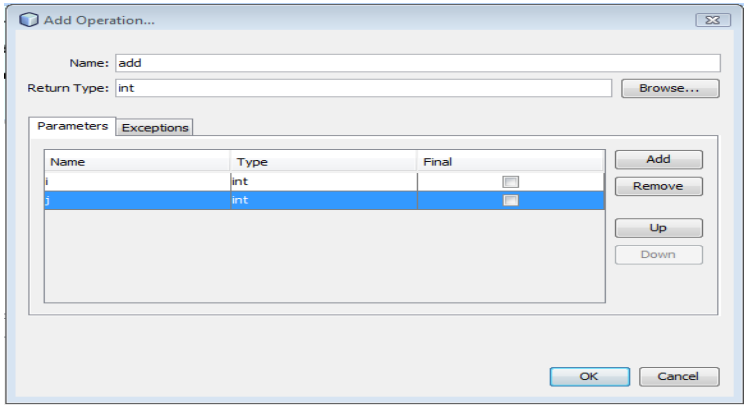
- Change to the Design view;



- Click Add Operation in the visual designer. A dialog box appears where you can define the new operation;



- 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. Then click Add again and create a parameter of type int called j;



- Click OK at the bottom of the Add Operation dialog box;
- Click Source and view the code that you generated in the previous steps;

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

/**
 *
 * @author adina
 */
@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 here:
        return 0;
    }

}
```

- In the editor, extend the skeleton add operation to the following;

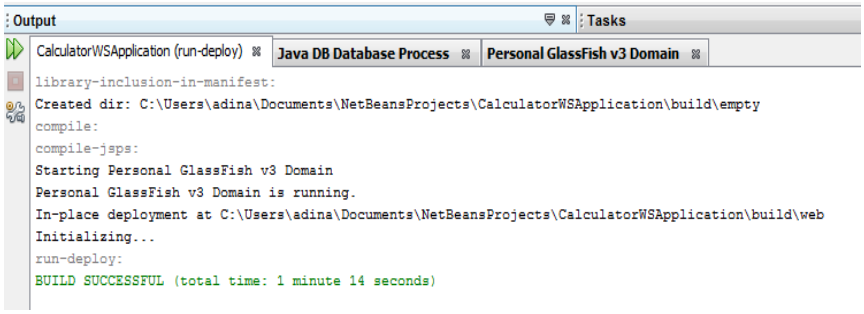
```
public int add(@WebParam(name = "i")
int i, @WebParam(name = "j")
int j) {

    int k = i + j;
    return k;

}
```

2.3. Deploying and Testing the Web Service

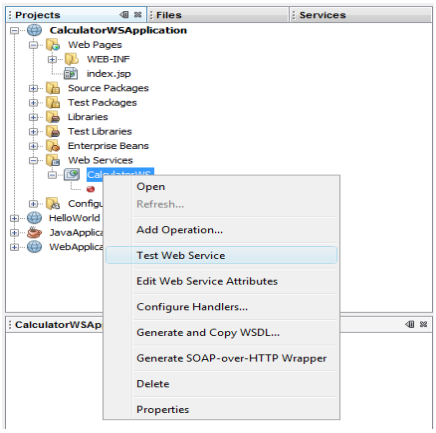
When you deploy a web service to a web container, the IDE lets you test the web service to see if it functions as you expect. The Tester application, provided by GlassFish, is integrated into the IDE for this purpose. For the Tomcat Web Server, there is a similar tool. However, while GlassFish's Tester page lets you enter values and test them, the Tomcat Web Server

does not. In the latter case, you can only see that the web service is deployed, you cannot test the values.

- 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 GlassFish or Tomcat tabs in the Output view;

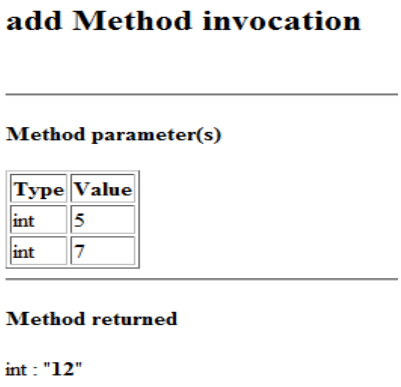


- In the IDE's Projects tab, expand the Web Services node of the CalculatorWSApplication project. Right-click the CalculatorWS node, and choose Test Web Service;



The IDE opens the tester page in your browser, if you deployed a web application to GlassFish:

The sum:

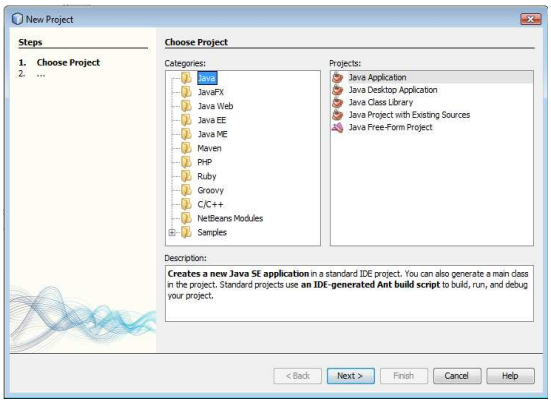


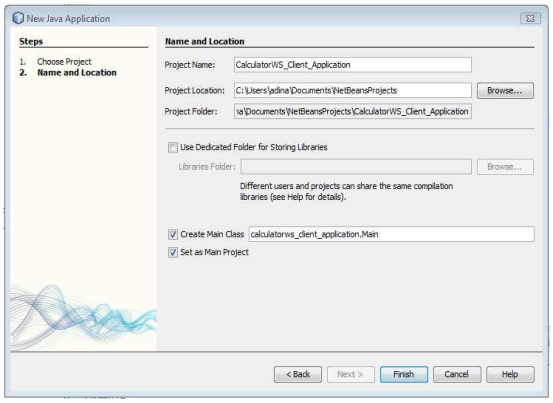
2.4.Consuming the Web Service

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.

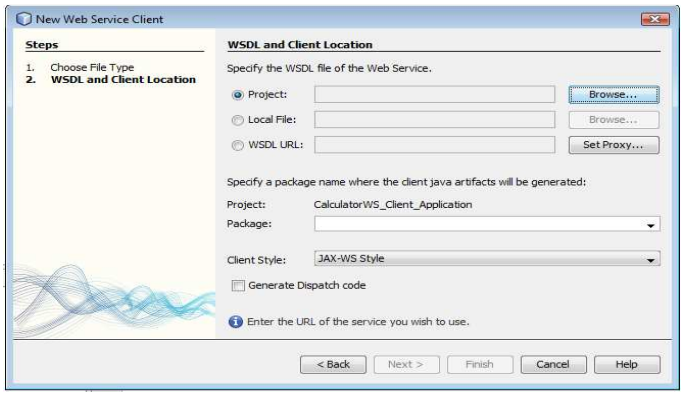
Client 1: Java Class in Java SE Application

- Choose File -> New Project (Ctrl-Shift-N). 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;





- Right-click the CalculatorWS_Client_Application node and choose New -> Web Service Client;



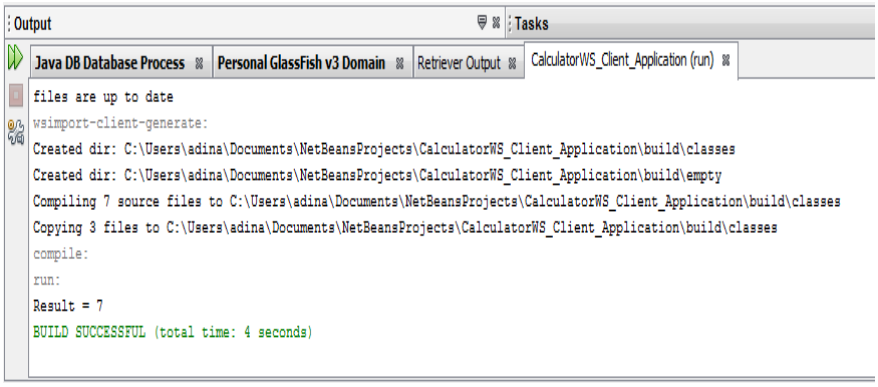
- In Project, click Browse. Browse to the web service that you want to consume. When you have selected the web service, click OK;
- Leave the other settings at default and click Finish;
- Right-click in the editor and then choose Insert Code > Call Web Service Operation;

```
public static void main(String[] args) {

    try { // Call Web Service Operation
        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);
        System.out.println("Result = "+result);
    } catch (Exception ex) {
        // TODO handle custom exceptions here
    }

}
```

- Initialize the two ints. Just change the values of the two ints above from 0 to other integers, such as 3 and 4;
- In the catch block, replace the commented out TODO with System.out.println("exception" + ex);

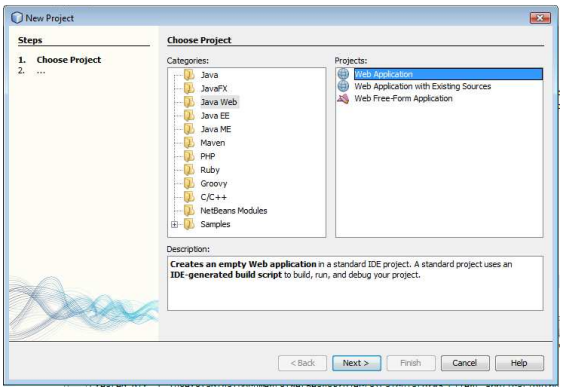
```
    public static void main(String[] args) {

        try { // Call Web Service Operation
            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 = 3;
            int j = 4;
            // TODO process result here
            int result = port.add(i, j);
            System.out.println("Result = "+result);
        } catch (Exception ex) {
           System.out.println("exception"+ex);
        }
    }

}
```

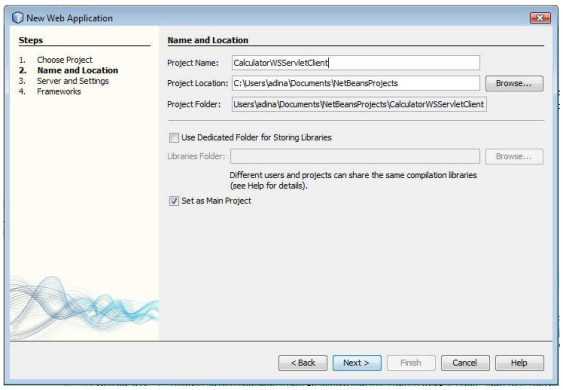
- Right-click the project node and choose Run;



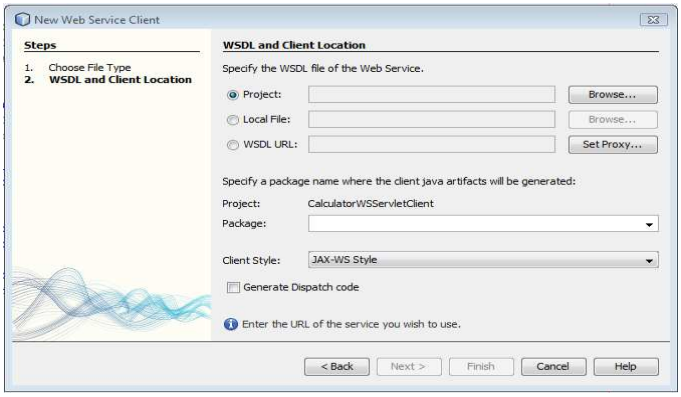
Client 2: Servlet in Web Application

- Choose File -> New Project (Ctrl-Shift-N). Select Web Application from the Java Web category. Name the project CalculatorWSServletClient. Click Next and then click Finish;



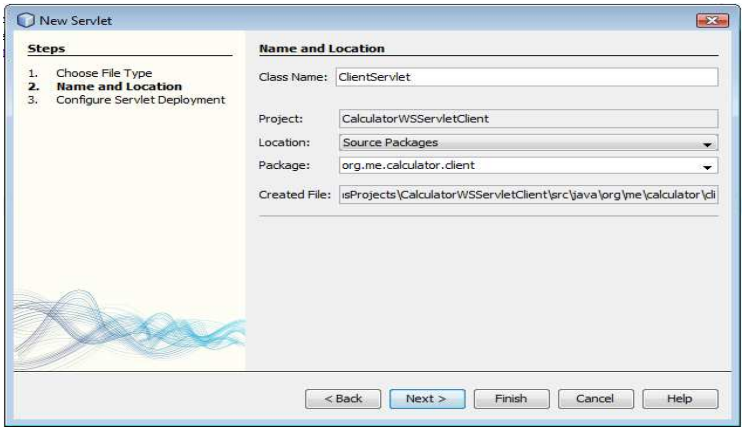


- Right-click the CalculatorWSServletClient node and choose New -> Web Service Client. The New Web Service Client wizard appears;

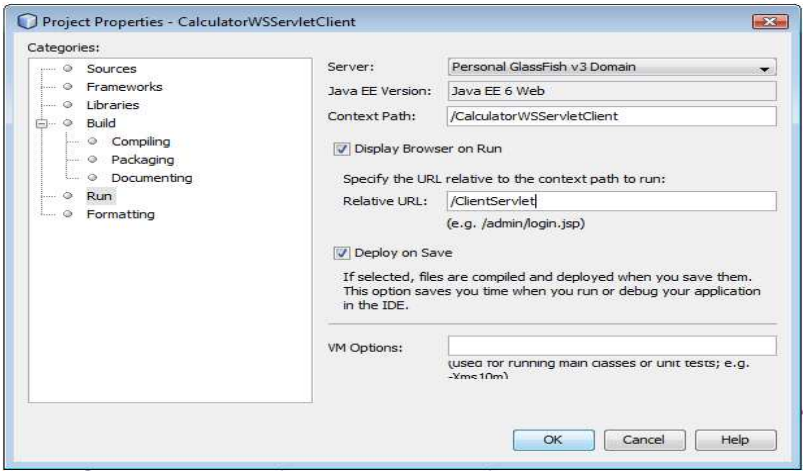


- In Project, click Browse. Browse to the web service that you want to consume. When you have selected the web service, click OK;

- Leave the other settings at default and click Finish;
- 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;



- 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.



- In the Source Editor, remove the line that comments out the body of the processRequest method;

- Drag the node that represents the add operation into the space that you created;

```
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 { // Call Web Service Operation
        org.me.calculator.CalculatorWS port = service.getCalculatorWSPort();
        // TODO initialize WS operation arguments here
        int i = 3;
        int j = 4;
        // TODO process result here
        int result = port.add(i, j);
        out.println("Result = "+result);
    } catch (Exception ex) {
        // TODO handle custom exceptions here
    }
} finally {
    out.println("</body>");
    out.println("</html>");
    out.close();
}
```

- Right-click the project node and choose Run;

Servlet ClientServlet at /CalculatorWSServletClient

Result = 7

Client 3: JSP Page in Web Application

- Choose File -> New Project (Ctrl-Shift-N). Select Web Application from the Java Web category. Name the project CalculatorWSJSPClient. Click Finish;
- Right-click the CalculatorWSJSPClient node and choose New -> Web Service Client;

- In Project, click Browse. Browse to the web service that you want to consume. When you have selected the web service, click OK;
- Leave the other settings at default and click Finish;
- 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;
- 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, a

```
<%-- start web service invocation --%><hr/>
<%
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 = 2;
    int j = 4;
    // TODO process result here
    int result = port.add(i, j);
    out.println("Result = "+result);
} catch (Exception ex) {
    // TODO handle custom exceptions here
}
%>
<%-- end web service invocation --%><hr/>
```

- Right-click the project node and choose Run;

Result = 6

- http://netbeans.org/kb/docs/websvc/jax-ws.html

3. Class work

Ex1: Test the examples presented in chapter 2.

Ex2: Create a simple Web Service that converts the temperature from Fahrenheit to Celsius, and vice versa. Create 3 different clients to make use of the Web service.

Fahrenheit to Celsius:

$$C = (F - 32) \cdot \frac{5}{9}$$

Celsius to Fahrenheit:

$$F = C \cdot \frac{9}{5} + 32$$

4. Home Work

Ex1: Make a presentation about a Real-life Web Services .

- http://www.javapassion.com/webservices/#Real-life_Web_Services__SOA

Lesson 7

C# Web Services

1. Introduction

A Web service is a class that allows its methods to be called by methods on other machines via common data formats and protocols, such as XML and HTTP. In .NET, the over-the-network method calls are commonly implemented through the Simple Object Access Protocol (SOAP), an XML-based protocol describing how to mark up requests and responses so that they can be transferred via protocols such as HTTP. Using SOAP, applications represent and transmit data in a standardized XML-based format.

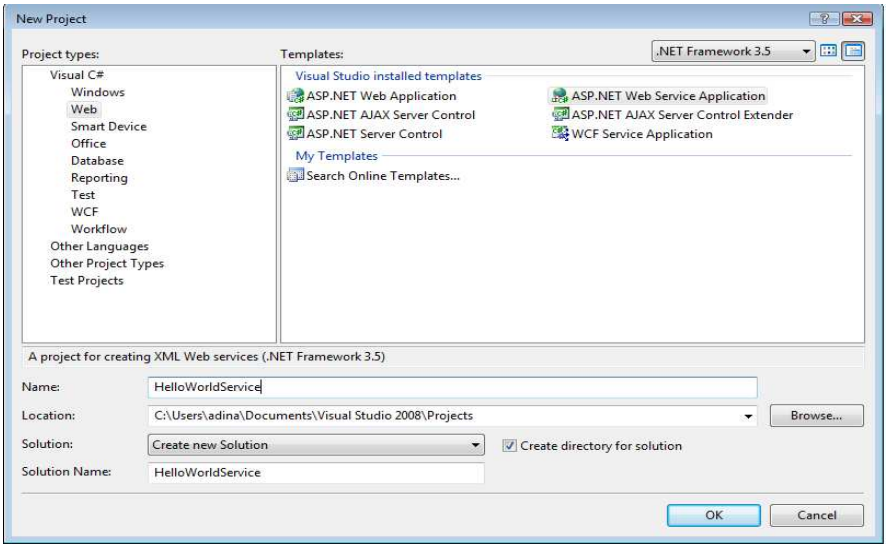
Web services have important implications for business-to-business (B2B) transactions. They enable businesses to conduct transactions via standardized, widely available Web services rather than relying on proprietary applications. Web services and SOAP are platform and language independent, so companies can collaborate via Web services without worrying about the compatibility of their hardware, software and communications technologies. Companies such as Amazon, Google, eBay and many others are using Web services to their advantage. To read case studies of Web services used in business, visit msdn.microsoft.com/webservices/understanding/casestudies/default.aspx.

Visual Web Developer and the .NET Framework provide a simple, user-friendly way to create Web services. In this paper, we learn how to use these tools to create, deploy and use Web services.

2. Web service applications using the wizards in Visual Studio. NET

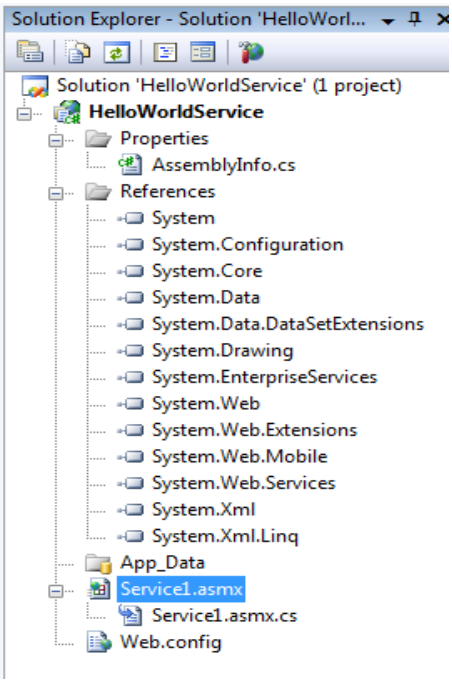
Creating your first web service using the wizards in Visual Studio. NET is incredibly easy. You can have your first service up and running in minutes with no coding.

To create a new web service, fire up Visual Studio .NET and either select the New Project button on the default Start Page or click File →New →Project on the main Visual Studio .NET menu bar. Here you have the option to create a variety of project types from templates. Under Visual C# Projects, one template option creates an ASP.NET web service.



Once you click OK, the IDE (Integrated Development Environment) creates a new solution project and automatically populate the project with several files. The IDE will also create a virtual folder with the same name as the project name, which, in this case, is HelloWorldService.

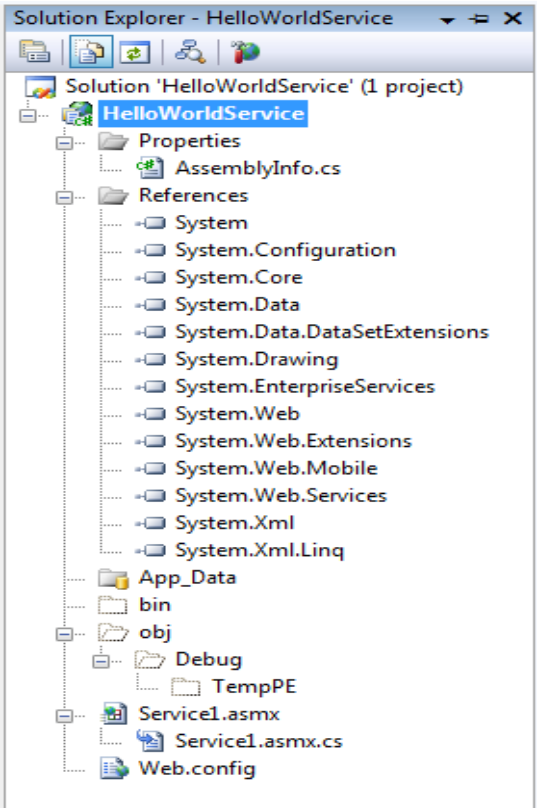
The contents of your new project are displayed in the Solution Explorer window, which should appear on the right side of the VS.NET IDE.



If the Solution Explorer is not visible, you can open it by selecting Solution Explorer from the View menu.

VS.NET has included assembly references to the System, System.Data, System.Web, System.Web.Services, and System.XML namespaces. (The System.Data and System.XML assembly references are not necessary for this example, so you can remove them if you'd like, but there's no real benefit to doing so other than simplicity.)

To see all files, select Show All Files from the Project menu tab (there's also an icon at the top of Solution Explorer to do this). The Solution Explorer view will change to look.



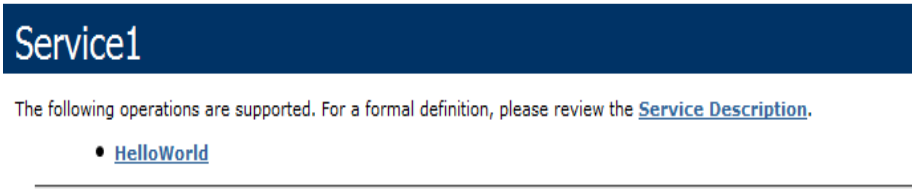
- AssemblyInfo.cs - An information file that provides the compiler with metadata (name, version, etc.) about the assemblies in the project;
- Web.config - An XML file containing configuration information for the application;
- *Service1.asmx.cs* - code view page;

This boilerplate code begins by importing several namespaces generally required for web services and by automatically generating namespace and class definitions. In this example, the namespace and class definitions are HelloWorldService and Service1, respectively.

The namespace definition is generated based on the project name, but you will probably want to change to something more depending on your application. The service name is always autogenerated as *Service1*. Change this to something more appropriate for your application, but you should also remember to change the name of the *.asmx* page to mirror your service name. Your service will run just fine if the names don't match up, but keeping the naming consistent can help make managing your service easier, particularly if you have a project with a large number of services.

The imported namespaces at the beginning of the code are provided as a convenience, and some of them are unnecessary. Specifically, the System.Data, System.Collections namespaces are not used at all. The classes of the System.ComponentModel namespace are used only by the web service designer methods, InitializeComponent() and Dispose(), which work in conjunction with a private member variable of type IContainer called components. To see these methods, you need to expand the Component Designer Generated Code region.

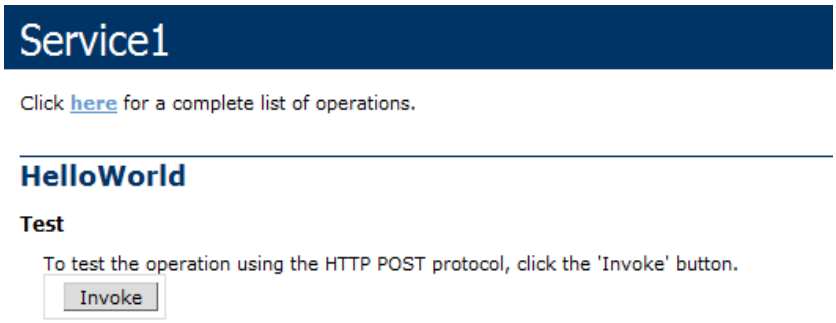
Once your application is complete, select Build Solution from the Build menu (or press Ctrl-Shift-B) and VS.NET will compile your web service and transfer the *.asmx* page and associated compiled assembly to the web server for you. If any errors result from the compile, VS.NET will display them in a panel labeled Output at the bottom of the IDE. Once you have successfully built the web service, it's ready to be used.



If you mouse over the HelloWorld link, you'll see the destination URL:

http://localhost/HelloWorldService.cs.asmx?op=HelloWorld

By clicking this link, you call the *.asmx* page, passing a parameter called `op` (standing presumably for operation) along with the name of the service. This action is the same as calling the `HelloWorld` web method of the web service using HTTP GET.



Here you'll see the name of the service and method along with a button to test the service. Through reflection, the logic in the *DefaultWsdlHelpGenerator.aspx* test page is able to determine the signature of our HelloWorld method. Because our web method takes no arguments, the page need provide only a button for invocation. If the method had a different signature, for example, if it reads a string of text, the *.aspx* help page would also provide a text box to capture this string and pass it, using HTTP GET, to the web method when the form was submitted. This text box method works fine for simple data type arguments, but if the web method were to require an object, this approach would not work.

You can invoke the web method using the IE test page by opening a web browser and navigating to the service's URL. You will see a page listing the

service's operation, which should be HelloWorld. Click the HelloWorld operation to navigate to the web method invocation page. This is a page that allows you to test the operation by clicking a button labeled Invoke. To invoke the service, click the button.

The deployment process in Visual Studio .NET is as simple as choosing the Build Solution option from the Build menu item (or pressing Ctrl-Shift-B). But, in order to take advantage of this two-click deployment, you first need to properly configure Visual Studio .NET to be able to deploy to your instance of IIS.

This service is only returning a string. Strings as well as most numbers are considered simple types. Web services are not limited to these simple types for return values or inbound parameters. The next web service will demonstrate how to return a complex type. For this create a web service that returns the date and time from the server. Instead, create a LocalTime struct that will be returned to the caller. The LocalTime struct holds seven values; Day, Month, Year, Hour, Minute, Seconds, Milliseconds, and Timezone. The struct's definition is below:

```
public struct LocalTime
{
public int Day;
public int Month;
public int Year;
public int Hour;
public int Minute;
public int Seconds;
```

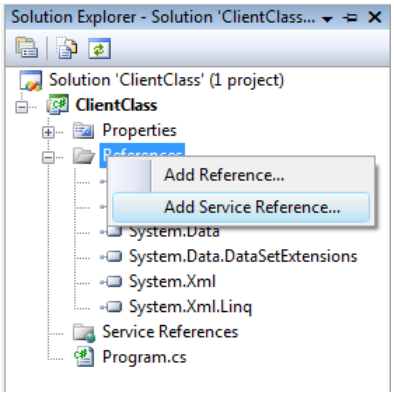
```
public int Milliseconds;
public string Timezone;
}


public class TimeService
{
[WebMethod]
public LocalTime GetTime()
{
LocalTime lt = new LocalTime();
DateTime dt = DateTime.Now;
lt.Day = dt.Day;
lt.Month = dt.Month;
lt.Year = dt.Year;
lt.Hour = dt.Hour;
lt.Minute = dt.Minute;
lt.Seconds = dt.Second;
lt.Milliseconds = dt.Millisecond;
lt.Timezone = TimeZone.CurrentTimeZone.StandardName;
return lt;
}
}
```

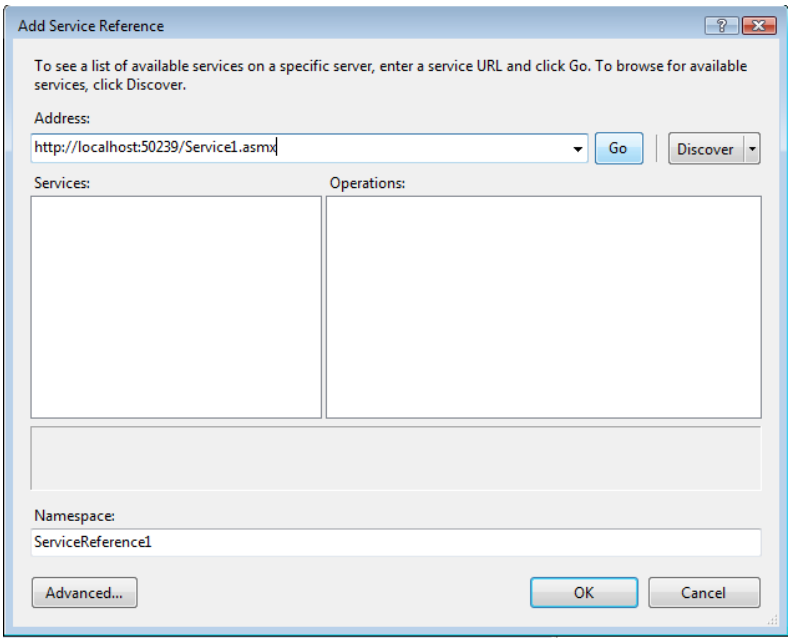
To demonstrate that the time web service is usable by any .Net application, create a simple console application in C# that prints out the time from the remote service.

```
using System;
class ClientClass
{
static void Main()
{
TimeService ts = new TimeService();
LocalTime lt = ts.GetTime();
string stime = lt.Hour + ":" + lt.Minute + ":" + lt.Seconds + "." +
lt.Milliseconds + " " + lt.Timezone;
string sdate = lt.Month + "/" + lt.Day + "/" + lt.Year;
Console.WriteLine("The remote date is: " + sdate);
Console.WriteLine("The remote time is: " + stime);
}
}
```

After creating the client application, add a proxy class to the project that allows the client to access the Web service. Recall that the proxy class (or proxy) is generated from the Web service's WSDL file and enables the client to call Web methods over the Internet. The proxy class handles all the details of communicating with the Web service. The proxy class is hidden from you by default-you can view it in the Solution Explorer by clicking the Show All Files button. The proxy class's purpose is to make clients think that they are calling the Web methods directly. The next example demonstrates how to create a Web service client and generate a proxy class that allows the client to access the Web service. Begin by adding a Web reference to the client project. When you add the Web reference, Visual C# will generate the appropriate proxy class. You will then create an instance of

the proxy class and use it to call the Web service's methods. To add Web reference perform the following steps:





3. Class work

Ex1: Test the examples presented in chapter 2.

Ex2: Create a simple Web Service that converts the temperature from Fahrenheit to Celsius, and vice versa. Create 2 different clients to make use of the Web service.

Fahrenheit to Celsius:

$$C = (F - 32) \cdot \frac{5}{9}$$

Celsius to Fahrenheit:

$$F = C \cdot \frac{9}{5} + 32$$

4. Home Work

Create a Web Service that give access to a Data Base where are store touristic information's about Cluj-Napoca City. Create a client to make use of the Web service.