1. Creating a Web Service Using Netbeans

Follow the following steps to create a web service that implements simple calculator methods.

1. Create a Netbeans project with main class `MyCalcWS` in a package `mycalcws`.

```java
package mycalcws;

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

2. Add an interface for the service to the package and annotate the interface and the web methods.

You can do this by right-clicking on the package node and selecting the option to add a new item, and then select the option to add a new interface.

```java
package mycalcws;
import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
public interface CalcServiceInterface {
    @WebMethod
    int sum(int x, int y);
    @WebMethod
    int prod(int x, int y);
}
```

3. Add a class to implement the service in the same package. You can do this by right-clicking on the package node and selecting the option to add a new Java class. Name your class `CalcServiceImplementor`.

```java
package mycalcws;
import javax.jws.WebService;

@WebService
public interface CalcServiceInterface {
    @WebMethod
    int sum(int x, int y);
    @WebMethod
    int prod(int x, int y);
}
```
public class CalcServiceImplementor implements CalcServiceInterface {
    @Override
    public int sum(int x, int y) {
        return x + y;
    }

    @Override
    public int prod(int x, int y) {
        return x * y;
    }
}

4. Modify the main class to publish the method

    package mycalcws;
    import javax.xml.ws.Endpoint;
    public class MyCalcWS {
        public static void main(String[] args) {
            Endpoint.publish("http://localhost:50000/Calc",
                new CalcServiceImplementor());
        }
    }

5. Run the program. In addition to the service endpoint, all SOAP-based services automatically implement a metadata exchange endpoint (MEX). By default, this endpoint can be accessed via HTTP to view metadata about the service. The metadata typically consists of a WSDL, as well as the XML schema of the type of data and messages that the service deals with.

It is now time to access the metadata exchange endpoint and view the WSDL. Once the program is running, start a browser session and point your browser to

http://localhost:50000/Calc?wsdl

2. Writing a Client against the Web Service

Leave the server running. Follow the following steps to implement a Java application that will call function as the web service client. The client will make calls to the remote web methods implemented by the service.

1. Start a new project to implement a client.

    package calcclient;

    public class CalcClient {
        public static void main(String[] args) {
            
            }
2. Right-click on the package calcclient, select New, then Web Service Client. In the resulting dialog box, select Web Service Client.

When you do this, Netbeans will ask for the location of the WSDL that describes the service you wish to connect to. The WSDL may be available via HTTP at the metadata exchange endpoint, or it may be stored available via HTTP from some other URL, or it may already be stored in a disk file on the local machine.

Netbeans will automatically invoke some command line tools to download a copy of the WSDL, if necessary. Then, Netbeans will use the same command line tools to generate Java classes that implement the service proxy on the client machine. The classes that implement the service proxy are referred to as the service proxy “artifacts.”

At any rate, when Netbeans asks for the location of the WSDL, type in the URL for the wsdl as in step 5 above. Make sure you select the package calcclient for the web service client “artifacts.” Netbeans will then create the service proxy for you.

It is important to understand that you can do all of this on the command line yourself, so you do not really need Netbeans. See Chapter 32 of Kalin, the section Generating Client-Support Code from the WSDL.

Note that the generated files are automatically added to the package, and you can view them in Netbeans.

Build the project.

3. You can now insert calls to methods on the service proxy. All you need is to know the names of the proxy classes that you need to instantiate, and the methods you need to call. The proxy code follows a certain naming convention, so we can easily figure out these names. Fortunately, we can use Netbeans to automate this process of inserting calls to the service.

Right click inside of the main method and select insert code, the call to web service.

You should get this

```java
package calcclient;

public class CalcClient {
    public static void main(String[] args) {
    }

    private static int prod(int arg0, int arg1) {
        calcclient.CalcServiceImplementorService service = new calcclient.CalcServiceImplementorService();
        calcclient.CalcServiceImplementor port =
```
service.getCalcServiceImplementorPort();
return port.prod(arg0, arg1);
}

private static int sum(int arg0, int arg1) {
    calcclient.CalcServiceImplementorService service
        = new calcclient.CalcServiceImplementorService();
    calcclient.CalcServiceImplementor port
        = service.getCalcServiceImplementorPort();
    return port.sum(arg0, arg1);
}

This inserts a static method in your code that actually calls the corresponding method on the service proxy.

We can edit this machine generated code to make it more presentable.

package calcclient;

class CalcClient {
    public static void main(String[] args) {
    
    }  

    private static int prod(int arg0, int arg1) {
        CalcServiceImplementorService service = new CalcServiceImplementorService();
        CalcServiceImplementor port = service.getCalcServiceImplementorPort();
        return port.prod(arg0, arg1);
    }

    private static int sum(int arg0, int arg1) {
        CalcServiceImplementorService service = new CalcServiceImplementorService();
        CalcServiceImplementor port = service.getCalcServiceImplementorPort();
        return port.sum(arg0, arg1);
    }
}

Next, observe that there is no need to recreate the service proxy object, or the port on the proxy every time we invoke a service method. We end up with this code

package calcclient;

class CalcClient {
    public static void main(String[] args) {
        CalcServiceImplementorService service = new CalcServiceImplementorService();
        CalcServiceImplementor port = service.getCalcServiceImplementorPort();
    
    }  

}
You can embed web service clients in desktop applications, web applications, and even other services. That is, any code can be a client of a web service.

3. Reflection

If you look in the client package, you will see the new code generated by the addition of the web service client. As mentioned before, you can actually carry out this step by hand by using the wsimport utility.

4. Hosting a web service in Glassfish

The previous example hosted a web service in console application. You can host a web service in a GUI application as well. You can also host a web service in a web application which you then deploy to Glassfish. When you deploy the application, glassfish will automatically publish the application.


2. Right-click on the package and add a new WebService. Select Create Web Service From Scratch

Do not select any other options or you will be sorry.

You will get a little web service to get you started

```java
package fruity;

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

@WebService(serviceName = "DeliciousFruits")
public class DeliciousFruits
{
    @WebMethod(operationName = "hello")
    public String hello(@WebParam(name = "name") String txt)
    {
        return "Hello " + txt + " !";
    }
}
```

3. Right-click on the project node and select Deploy. This will deploy the web application to Glassfish and simultaneously publish the service.

4. Expand the Web Services node under the Project node and right Click on the web service DeliciousFruits, and select Test Web Service.

This will display a test web site that you can use to test the web service.

This also show the structure of the SOAP messages sent and received by the web service when you test.
5. Right-click the Web Service node again and select Add Operation.

Enter String [] for the return type and click OK. You will get some code added:
edit the code so it looks like this

```java
package fruity;

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

@WebService(serviceName = "DeliciousFruits")
public class DeliciousFruits {
    @WebMethod(operationName = "hello")
    public String hello(@WebParam(name = "name") String txt) {
        return "Hello " + txt + " !";
    }

    String [] myfruits = {"apples", "oranges", "strawberries", "kiwis"};
    @WebMethod(operationName = "getAllFruits")
    public String [] getAllFruits() {
        return myfruits;
    }
}
```

6. Deploy the service and test. Notice that the it returns a list of strings rather than an array of strings.

7. Follow the previous procedure to add another operation. This time put String for a return type, and add a parameter index of type int. Edit the whole mess to get this

```java
@WebMethod(operationName = "getFruit")
public String getFruit(@WebParam(name = "index") int index) {
    return myfruits[Math.abs(index) % myfruits.length];
}
```

Deploy and Test.