

JAX-WS Hello World Example – RPC Style

By [mkyong](#) | November 16, 2010 | Updated : August 29, 2012 | Viewed : 1,041,971 | +1,223 pv/w

JAX-WS is bundled with JDK 1.6, which makes Java web service development easier to develop. This tutorial shows you how to do the following tasks:

1. Create a SOAP-based RPC style web service endpoint by using JAX-WS.
2. Create a Java web service client manually.
3. Create a Java web service client via **wsimport** tool.
4. Create a Ruby web service client.

You will be surprise of how simple it is to develop a RPC style web service in JAX-WS.

Note

In general words, “web service endpoint” is a service which published outside for user to access; where “web service client” is the party who access the published service.

JAX-WS Web Service End Point

The following steps showing how to use JAX-WS to create a RPC style web service endpoint.



AD Password Management

Ad Active Directory Password Reset. No l Calls! Free Trial.

[manageengine.com](#)

[Learn more](#)

1. Create a Web Service Endpoint Interface

File : HelloWorld.java

```
package com.mkyong.ws;

import javax.xml.ws.WebMethod;
import javax.xml.ws.WebService;
import javax.xml.ws.soap.SOAPBinding;
import javax.xml.ws.soap.SOAPBinding.Style;

//Service Endpoint Interface
@WebService
@SOAPBinding(style = Style.RPC)
public interface HelloWorld{

    @WebMethod String getHelloWorldAsString(String name);

}
```

AD Password Management

Active Directory Password Reset. No More Helpdesk Calls! Free Trial. [manageengine.com](#)

2. Create a Web Service Endpoint Implementation

File : HelloWorldImpl.java

```

package com.mkyong.ws;

import javax.jws.WebService;

//Service Implementation
@WebService(endpointInterface = "com.mkyong.ws.HelloWorld")
public class HelloWorldImpl implements HelloWorld{

    @Override
    public String getHelloWorldAsString(String name) {
        return "Hello World JAX-WS " + name;
    }

}

```

3. Create a Endpoint Publisher

File : *HelloWorldPublisher.java*

```

package com.mkyong.endpoint;

import javax.xml.ws.Endpoint;
import com.mkyong.ws.HelloWorldImpl;

//Endpoint publisher
public class HelloWorldPublisher{

    public static void main(String[] args) {
        Endpoint.publish("http://localhost:9999/ws/hello", new HelloWorldImpl());
    }

}

```

Run the endpoint publisher, and your “**hello world web service**” is deployed in URL “**http://localhost:9999/ws/hello**”.

4. Test It

You can test the deployed web service by accessing the generated WSDL (Web Service Definition Language) document via this URL “**http://localhost:9999/ws/hello?wsdl**” .

Web Service Clients

Ok, web service is deployed properly, now let’s see how to create web service client to access to the published service.

1. Java Web Service Client

Without tool, you can create a Java web service client like this :

```

package com.mkyong.client;

import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import com.mkyong.ws.HelloWorld;

public class HelloWorldClient{

    public static void main(String[] args) throws Exception {

        URL url = new URL("http://localhost:9999/ws/hello?wsdl");

        //1st argument service URI, refer to wsdl document above
        //2nd argument is service name, refer to wsdl document above
        QName qname = new QName("http://ws.mkyong.com/", "HelloWorldImplService");

        Service service = Service.create(url, qname);

        HelloWorld hello = service.getPort(HelloWorld.class);

        System.out.println(hello.getHelloWorldAsString("mkyong"));

    }

}

```

Output

```
Hello World JAX-WS mkyong
```

2. Java Web Service Client via wsimport tool

Alternative, you can use “**wsimport**” tool to parse the published wsdl file, and generate necessary client files (stub) to access the published web service.

Where is wsimport?

This **wsimport** tool is bundle with the JDK, you can find it at “*JDK_PATH/bin*” folder.

Issue “**wsimport**” command.

```
wsimport -keep http://localhost:9999/ws/hello?wsdl
```

It will generate necessary client files, which is depends on the provided wsdl file. In this case, it will generate one interface and one service implementation file.

File : HelloWorld.java

```
package com.mkyong.ws;

import javax.xml.ws.WebMethod;
import javax.xml.ws.WebParam;
import javax.xml.ws.WebResult;
import javax.xml.ws.WebService;
import javax.xml.ws.soap.SOAPBinding;

/**
 * This class was generated by the JAX-WS RI.
 * JAX-WS RI 2.1.1 in JDK 6
 * Generated source version: 2.1
 *
 */
@WebService(name = "HelloWorld", targetNamespace = "http://ws.mkyong.com/")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public interface HelloWorld {

    /**
     *
     * @param arg0
     * @return
     *     returns java.lang.String
     */
    @WebMethod
    @WebResult(partName = "return")
    public String getHelloWorldAsString(
        @WebParam(name = "arg0", partName = "arg0")
        String arg0);

}
```

File : HelloWorldImplService.java

```

package com.mkyong.ws;

import java.net.MalformedURLException;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import javax.xml.ws.WebEndpoint;
import javax.xml.ws.WebServiceClient;
import javax.xml.ws.WebServiceFeature;

/**
 * This class was generated by the JAX-WS RI.
 * JAX-WS RI 2.1.1 in JDK 6
 * Generated source version: 2.1
 *
 */
@WebServiceClient(name = "HelloWorldImplService",
    targetNamespace = "http://ws.mkyong.com/",
    wsdlLocation = "http://localhost:9999/ws/hello?wsdl")
public class HelloWorldImplService
    extends Service
{

    private final static URL HELLOWORLDIMPLSERVICE_WSDL_LOCATION;

    static {
        URL url = null;
        try {
            url = new URL("http://localhost:9999/ws/hello?wsdl");
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
        HELLOWORLDIMPLSERVICE_WSDL_LOCATION = url;
    }

    public HelloWorldImplService(URL wsdlLocation, QName serviceName) {
        super(wsdlLocation, serviceName);
    }

    public HelloWorldImplService() {
        super(HELLOWORLDIMPLSERVICE_WSDL_LOCATION,
            new QName("http://ws.mkyong.com/", "HelloWorldImplService"));
    }

    /**
     *
     * @return
     *     returns HelloWorld
     */
    @WebEndpoint(name = "HelloWorldImplPort")
    public HelloWorld getHelloWorldImplPort() {
        return (HelloWorld)super.getPort(
            new QName("http://ws.mkyong.com/", "HelloWorldImplPort"),
            HelloWorld.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 HelloWorld
     */
    @WebEndpoint(name = "HelloWorldImplPort")
    public HelloWorld getHelloWorldImplPort(WebServiceFeature... features) {
        return (HelloWorld)super.getPort(
            new QName("http://ws.mkyong.com/", "HelloWorldImplPort"),
            HelloWorld.class,
            features);
    }

}

```

Now, create a Java web service client which depends on the above generated files.

```

package com.mkyong.client;

import com.mkyong.ws.HelloWorld;
import com.mkyong.ws.HelloWorldImplService;

public class HelloWorldClient{

    public static void main(String[] args) {

        HelloWorldImplService helloService = new HelloWorldImplService();
        HelloWorld hello = helloService.getHelloWorldImplPort();

        System.out.println(hello.getHelloWorldAsString("mkyong"));

    }

}

```

Here's the output

```

Hello World JAX-WS mkyong

```

3. Ruby Web Service Client

Often time, web service development is mixed use with other programming language. So, here's a Ruby web service client example, which is used to access the published JAX-WS service.

```

# package for SOAP-based services
require 'soap/wsdlDriver'

wsdl_url = 'http://localhost:9999/ws/hello?wsdl'

service = SOAP::WSDLDriverFactory.new(wsdl_url).create_rpc_driver

# Invoke service operations.
data1 = service.getHelloWorldAsString('mkyong')

# Output results.
puts "getHelloWorldAsString : #{data1}"

```

Output

```

getHelloWorldAsString : Hello World JAX-WS mkyong

```

Tracing SOAP Traffic

From top to bottom, showing how SOAP envelope flows between client and server. See #1 web service client again :

```

URL url = new URL("http://localhost:9999/ws/hello?wsdl");
QName qname = new QName("http://ws.mkyong.com/", "HelloWorldImplService");
Service service = Service.create(url, qname);

HelloWorld hello = service.getPort(HelloWorld.class);

System.out.println(hello.getHelloWorldAsString("mkyong"));

```

Note

To monitor SOAP traffic is very easy, see this guide – [“How to trace SOAP message in Eclipse IDE”](#).

1. Request a WSDL file

First, client send a wsdl request to service endpoint, see HTTP traffic below :

Client send request :

```

GET /ws/hello?wsdl HTTP/1.1
User-Agent: Java/1.6.0_13
Host: localhost:9999
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive

```

Server send response :

```

HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: text/xml; charset=utf-8

<?xml version="1.0" encoding="UTF-8"?>

<definitions
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://ws.mkyong.com/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  targetNamespace="http://ws.mkyong.com/"
  name="HelloWorldImplService">

  <types></types>

  <message name="getHelloWorldAsString">
    <part name="arg0" type="xsd:string"></part>
  </message>
  <message name="getHelloWorldAsStringResponse">
    <part name="return" type="xsd:string"></part>
  </message>

  <portType name="HelloWorld">
    <operation name="getHelloWorldAsString" parameterOrder="arg0">
      <input message="tns:getHelloWorldAsString"></input>
      <output message="tns:getHelloWorldAsStringResponse"></output>
    </operation>
  </portType>

  <binding name="HelloWorldImplPortBinding" type="tns:HelloWorld">

    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc"></soap:binding>
    <operation name="getHelloWorldAsString">
      <soap:operation soapAction=""></soap:operation>
      <input>
        <soap:body use="literal" namespace="http://ws.mkyong.com/"></soap:body>
      </input>
      <output>
        <soap:body use="literal" namespace="http://ws.mkyong.com/"></soap:body>
      </output>
    </operation>

  </binding>

  <service name="HelloWorldImplService">
    <port name="HelloWorldImplPort" binding="tns:HelloWorldImplPortBinding">
      <soap:address location="http://localhost:9999/ws/hello"></soap:address>
    </port>
  </service>
</definitions>

```

2. hello.getHelloWorldAsString()

A second call, client put method invoke request in SOAP envelope and send it to service endpoint. At the service endpoint, call the requested method and put the result in a SOAP envelope and send it back to client.

Client send request :

```

POST /ws/hello HTTP/1.1
SOAPAction: ""
Accept: text/xml, multipart/related, text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Content-Type: text/xml; charset=utf-8
User-Agent: Java/1.6.0_13
Host: localhost:9999
Connection: keep-alive
Content-Length: 224

<?xml version="1.0" ?>
  <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
      <ns2:getHelloWorldAsString xmlns:ns2="http://ws.mkyong.com/">
        <arg0>mkyong</arg0>
      </ns2:getHelloWorldAsString>
    </S:Body>
  </S:Envelope>

```


Server send response :

```
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: text/xml; charset=utf-8

<?xml version="1.0" ?>
  <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
      <ns2:getHelloWorldAsStringResponse xmlns:ns2="http://ws.mkyong.com/">
        <return>Hello World JAX-WS mkyong</return>
      </ns2:getHelloWorldAsStringResponse>
    </S:Body>
  </S:Envelope>
```

Done, any comments are appreciated.

Download Source Code



Download It – [JAX-WS-HelloWorld-RPC-Example.zip](#) (14KB)

- [hello world](#)
- [jax-ws](#)
- [web services](#)

DO YOU KNOW YOUR TECH ODDS AND ENDS?

One theory about how QWERTY came to be is:

15

A

It was designed to slow typists down

B

It was to limit kids' ability to type

C

It is the easiest setup to type HTML

D

It was completely random

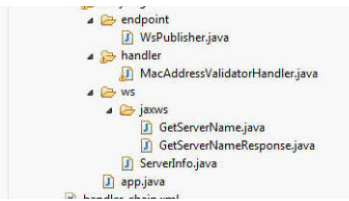
Sponsored

Sponsored

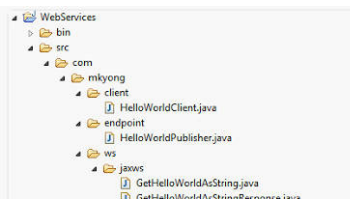
About the Author



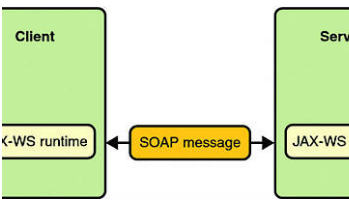
mkyong
Founder of [Mkyong.com](#), love Java and open source stuff. Follow him on [Twitter](#), or befriend him on [Facebook](#) or [Google Plus](#). If you like my tutorials, consider make a donation to [these charities](#).



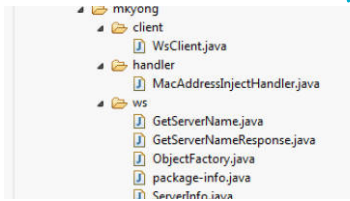
JAX-WS : SOAP handler in server side



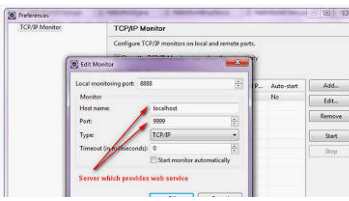
JAX-WS Hello World Example – Document Style



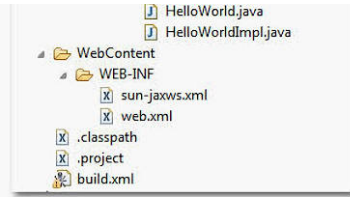
JAX-WS Tutorial



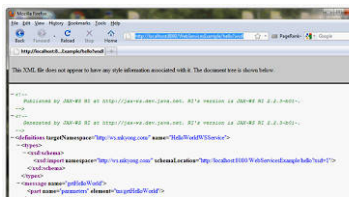
JAX-WS : SOAP handler in client side



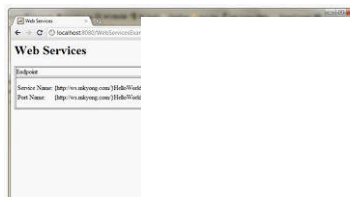
How to trace SOAP message in Eclipse IDE



Deploy JAX-WS web services on Tomcat




JAX-WS + Spring integration example



JAX-WS + Application Example

Comments



Join the discussion...