International Journal of Research in Advent Technology (E-ISSN: 2321-9637) Special Issue
1st International Conference on Advent Trends in Engineering, Science and Technology
"ICATEST 2015", 08 March 2015

Developing Asynchronous Web Service and their Invocation

G M Tere[1], R R Mudholkar[2], B T Jadhav[3]

[1] *Research Scholar, Dept of Computer Science, Shivaji University, Kolhapur-416004, girish.tere@gmail.com*
[2] *Professor, Dept of Electronics, Shivaji University, Kolhapur-416004,* rrm_eln@unishivaji.ac.in
[3] *Professor, Dept of Computer Science,* Y.C. Institute of Science, Satara, 415001, btj21875@gmail.com

Abstract: This paper introduces asynchronous Web service concepts and describes how to develop and configure asynchronous Web services. The JAX-WS specification provides an asynchronous client API that can call synchronous methods in an asynchronous way. Web services are really useful if they are called from remote machines/devices. This will be time consuming. To improve performance one must call Web services asynchronously.

Keywords – Asynchronous call**,** JAX-WS, MDB, WSDL

1. INTRODUCTION TO ASYNCHRONOUS WEB SERVICES

Web service invocation in Java API for XML-Web Services (JAX-WS) 2.0 [1] is built upon the concurrent programming support in JDK. When a Web service synchronously invoked, the invoking client application waits for the response to return before it can continue with its work. In cases where the response returns immediately, this method of invoking the Web service might be adequate. However, because request processing can be delayed, it is often useful for the client application to continue its work and handle the response later on. By calling a Web service asynchronously, the client can continue its processing, without interrupt, and will be notified when the asynchronous response is returned.

2. Asynchronous Web Service Using a Request and a Response Queue

Two separate message-driven beans (MDBs) are required in this case. One MDB is required to handle the request processing and one to handle the response processing as shown in Fig. 1. By separating the execution of business logic from the response return, this scenario provides improved error recovery.

The flow of event shown in Fig. 1 is described below:

1 The client calls an asynchronous method.
2 The asynchronous Web services receives the request and stores it in the request queue.
3 The asynchronous Web service sends a receipt confirmation to the client.
4 The MDB listener on the request queue receives the message and initiates processing of the request.

5The request MDB calls the required method in the Web service implementation.

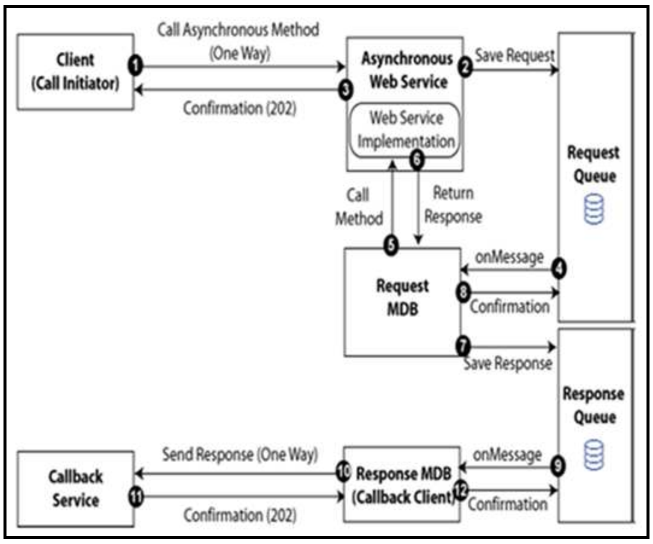

Fig. 1: Asynchronous Web Service Using a Request and Response Queue

5 The Web service implementation returns the response.
6 The request MDB saves the response to the response queue.
7 The request MDB sends a confirmation to the request queue to terminate the process.
8 The onMessage listener on the response queue initiates processing of the response.
9 The response MDB, acting as the callback client, returns the response to the callback service.
10 The callback service returns a receipt confirmation message.

11 The response MDB returns a confirmation message to the response queue to terminate the sequence.

3. Client Calling Asynchronous Web Service

From the client side, the asynchronous method call consists of two one-way message exchanges, as shown in Fig. 2.

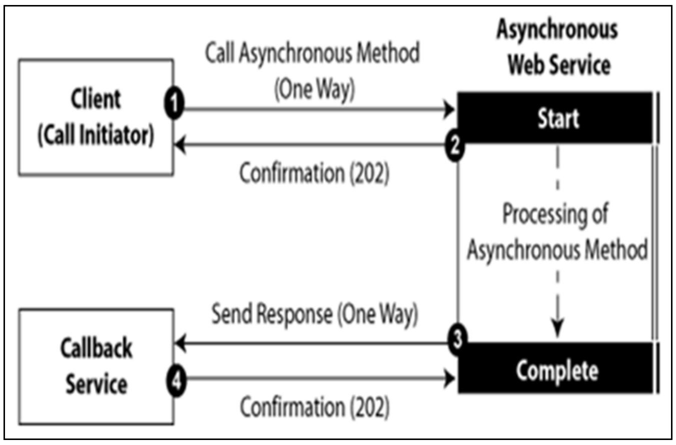

Fig. 2: Asynchronous Web Service Client Flow

As shown in Fig. 2, before initiating the asynchronous call, the client must deploy a callback service to listen for the response from the asynchronous Web service.
The message flow is as follows:

1 The client calls an asynchronous method.
2 The asynchronous Web services receive the request; send a confirmation message to the initiating client, and starts process the request.
3 Once processing of the request is complete, the asynchronous Web service acts as a client to send the response back to the callback service.
4 The callback service sends a confirmation message to the asynchronous Web service.

4. Developing an Asynchronous Web Service

A JAX-WS Web service can be declared an asynchronous Web service using the following annotation:

oracle.webservices.annotations.async.AsyncWebService.

The following provides a very simple POJO example of an asynchronous Web service:

```
import oracle.webservices.annotations.PortableWebService
import
oracle.webservices.annotations.async.AsyncWebService
@PortableWebService
@AsyncWebService
public class HelloService {
   public String hello(String name) {
      return "Hi " + name;
   }
}
```

The generated WSDL for the asynchronous Web service contains two one-way operations defined as two portTypes: one for the asynchronous operation and one for the callback operation.
For example:

```
<wsdl:portType name="HelloService">
  <wsdl:operation name="hello">
  <wsdl:input message="tns:helloInput"
  xmlns:ns1="http://www.w3.org/2006/05/addr
  essing/wsdl"
  ns1:Action=""/>
  </wsdl:operation>
  </wsdl:portType>
  <wsdl:portType
   name="HelloServiceResponse">
  <wsdl:operation name="helloResponse">
  <wsdl:input message="tns:helloOutput"
  xmlns:ns1="http://www.w3.org/2006/05/addr
  essing/wsdl"
  ns1:Action=""/>
  </wsdl:operation>
</wsdl:portType>
```

Calling Web service asynchronously improves performance than calling synchronously. To check this StockQuote Web service is developed using JDK 6.0, JAX-WS 2.1 reference implementation (RI) [1][2], and Tomcat 6.0

An asynchronous invocation of the program must return immediately without waiting for the actual computation to complete. For the calling client to retrieve the computation result, one approach is to return the client an instance of the java.util.concurrent.Future interface as a placeholder for the result. This interface is defined as follows:

```
public interface Future<V>{
   boolean cancel(boolean mayInterruptIfRunning);
   boolean isCancelled();
   boolean isDone();
   V get() throws InterruptedException,
ExecutionException;
   V get(long timeout, TimeUnit unit) throws
InterruptedException,ExecutionException,TimeoutExcepti
on;
}
```

V represents the computation result. get() allows the client to make a rendezvous with the Future. When called, it returns the result if ready; otherwise, it will block until the computation has completed. With get(long timeout, TimeUnit unit), the calling client can specify the maximum waiting time. Computation may be cancelled with the cancel() method, but this can only be done before the computation completes. The method isDone() is used to check the status of the computation.

The second approach in asynchronous interaction is to use callbacks. In this scenario, the calling client provides a callback handler to the program along with the invocation. The callback handler communicates instructions from the client to the program on how to process the result. When the result is ready, the program invokes the handler, and processes the result as instructed. Typically, the program will utilize the Executor frameworkin java.util.concurrent to provide a thread to execute the callback handler, and the process may be tuned through the Executor.
JAX-WS 2.0 [3][4] adopts both approaches in its asynchronous web service invocation mechanism. In this case, the program is the client runtime. A web service client with asynchrony enabled for a particular operation may poll the invocation result using the generic interface javax.xml.ws.Response, which directly extends Future<T>.

5. A Simple Web Service

A simple web service is developed using with JAX-WS 2.0. The service side of a web service can be developed from a service endpoint implementation class or from a WSDL file. This example follows the first approach with the following implementation class:

```
public class StockQuoteImpl {
    public double getQuote(String ticker) {
        double result = 0.0;
        if (ticker.equals("MHP")) {
            result = 50.0;
        } else if (ticker.equals("IBM")) {
            result = 83.0;
        }
        return result;
    }
}
```

To expose a class as a web service, JAX-WS 2.0 utilizes the annotation mechanism of JDK 6.0 [4], particularly uses those annotations defined by JSR. Web Service Metadata for the Java Platform. First step is to annotate the StockQuoteImpl class and its only operation, getQuote(), with @WebService() and @WebMethod() espectively:

```
@WebService(name="StockQuote",
serviceName="StockQuoteService")
public          class          StockQuoteImpl          {
                @WebMethod(operationName="getQuote")
     public     double     getQuote(String     code)     {
                                                   ......
                                                        }
}
```

The build-server-java task produces those annotated classes:

```
@XmlRootElement(name = "getQuote",
 namespace =
"http://server.stockquote.jaxws.company.com/")
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "getQuote",
 namespace =
"http://server.stockquote.jaxws.company.com/")
public class GetQuote {
   @XmlElement(name = "arg0", namespace = "")
   private String arg0;
   public String getArg0() {
      return this.arg0;
   }
   public void setArg0(String arg0) {
      this.arg0 = arg0;
   }
}
```

and

```
@XmlRootElement(name = "getQuoteResponse",
 namespace =
"http://server.stockquote.jaxws.company.com/")
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "getQuoteResponse",
 namespace =
"http://server.stockquote.jaxws.company.com/")
public class GetQuoteResponse {
   @XmlElement(name = "return", namespace = "")
   private double _return;
   public double get_return() {
      return this._return;
   }
   public void set_return(double _return) {
      this._return = _return;
   }
}
```

These are the JavaBeans required by JAXB 2.0 for marshalling and un-marshalling web service request and response messages in JAX/XML binding [5]. To complete our web service development, we need to package the service-side artifacts into a .war archive so that it can be deployed into Tomcat.

```
<target name="create-war">
  <war
warfile="${build.war.home}/${ant.project.name}.war"
      webxml="etc/web.xml">
     <webinf dir="${basedir}/etc" includes="sun-
jaxws.xml"/>
      <classes dir="${build.classes.home}"
includes="**/*.class"/>
  </war>
</target>
```

The content in web.xml is specific to the implementations of Tomcat, GlassFish, and Sun Java System Application Server.

The content of the .war file is shown below:

```
WEB-INF\web.xml
WEB-INF\sun-jaxws.xml
WEB-
INF\classes\com\company\jaxws\stockquote\server\Sto
kQuoteImpl.class
WEB-
INF\classes\com\company\jaxws\stockquote\server\jax
ws\GetQuote.class
WEB-
INF\classes\com\company\jaxws\stockquote\server\jax
ws\GetQuoteResponse.class
```

6. Asynchronous Invocation with a Port Proxy

Asynchronous invocation in JAX-WS 2.0 can be realized completely in the client runtime, and this is the approach JAX-WS 2.0 RI takes. The focus is in generating the service endpoint interface with operations for asynchronously invoking the web service. Then, based on the interface, the client runtime creates the dynamic port proxy, which will carry out the actual asynchronous invocation. In JAX-WS 2.0 RI, customizing the wsimport Ant task can generate this service endpoint interface. The client artifacts are as follows:

```
<target name="generate-client-async" depends="setup">
   <wsimport
         debug="${debug}"
         verbose="${verbose}"
         keep="${keep}"
         extension="${extension}"
         destdir="${build.classes.home}"
         wsdl="${client.wsdl}">
      <binding dir="${basedir}/etc"
includes="${schema.binding}"/>
      <binding dir="${basedir}/etc"
includes="${client.binding.async}"/>
   </wsimport>
</target>
```

The difference from our previous use of wsimport is the file for one of the binding subelements (${client.binding.async}). This element takes external binding files for customizing WSDL binding for JAX-WS 2.0 (the *custom-schema.xml* pointed by ${schema.binding} is for JAXB binding declaration, and we will not discuss it here). In the synchronous case, we use *custom-client.xml* (pointed by${client.binding}) for WSDL binding customization:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<bindings
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"

wsdlLocation="http://localhost:1970/stockquote/stockquote
?wsdl"
   xmlns="http://java.sun.com/xml/ns/jaxws">
   <bindings node="wsdl:definitions">
      <package
name="com.company.jaxws.stockquote.client"/>
   </bindings>
</bindings>
```

In the asynchronous case, one more element is added to <bindings node="wsdl:definitions"> in the above file (and result in *custom-client*-async.xml)

```
<bindings node="wsdl:definitions">
   <package
name="com.company.jaxws.stockquote.client"/>
   <enableAsyncMapping>true</enableAsyncMapping>
</bindings>
```

As a result, the service endpoint interface
StockQuote becomes:

```
@WebService(name = "StockQuote",
 targetNamespace =
"http://server.stockquote.jaxws.company.com/")
public interface StockQuote {
   @WebMethod(operationName = "getQuote")
   @RequestWrapper(localName = "getQuote",
         targetNamespace =
"http://server.stockquote.jaxws.company.com/",
         className =
"com.company.jaxws.stockquote.client.GetQuote")
   @ResponseWrapper(localName = "getQuoteResponse",
         targetNamespace =
"http://server.stockquote.jaxws.company.com/",
         className =
"com.company.jaxws.stockquote.client.GetQuoteResponse"
)
   public Response<GetQuoteResponse>getQuoteAsync(
      @WebParam(name = "arg0", targetNamespace = "")
      String arg0);

   @WebMethod(operationName = "getQuote")
   @RequestWrapper(localName = "getQuote",
         targetNamespace =
"http://server.stockquote.jaxws.company.com/",
         className =
"com.company.jaxws.stockquote.client.GetQuote")
   @ResponseWrapper(localName = "getQuoteResponse",
         targetNamespace =
"http://server.stockquote.jaxws.company.com/",
         className =
"com.company.jaxws.stockquote.client.GetQuoteResponse"
)
   public Future<?>getQuoteAsync(
      @WebParam(name = "arg0", targetNamespace = "")
      String arg0,
      @WebParam(name = "asyncHandler",
targetNamespace = "")
      AsyncHandler<GetQuoteResponse>asyncHandler);

   @WebMethod
   @WebResult(targetNamespace = "")
   @RequestWrapper(localName = "getQuote",
         targetNamespace =
"http://server.stockquote.jaxws.company.com/",
         className =
"com.company.jaxws.stockquote.client.GetQuote")
   @ResponseWrapper(localName = "getQuoteResponse",
         targetNamespace =
"http://server.stockquote.jaxws.company.com/",
         className =
"com.company.jaxws.stockquote.client.GetQuoteResponse"
)
  public double getQuote(
     @WebParam(name = "arg0", targetNamespace = "")
     String arg0);
}
```

With this interface, the web service is invoked through the corresponding proxy asynchronously in two ways as mentioned before. One is pulling:

```
public
Response<GetQuoteResponse>getQuoteAsync(String
arg0);
```

The other employs a callback mechanism through:

```
public Future<?>getQuoteAsync(String arg0,
AsyncHandler<GetQuoteResponse>asyncHandler);
```

In our sample client, StockQuoteClientAsync, the pulling method is called as follows:

```
Response<GetQuoteResponse>resp = port.getQuoteAsync
(code);
Thread.sleep (2000);
GetQuoteResponse output = resp.get();
```

In summary, all the features of the Future interface come under our employment with this asynchronous pulling method. The callback mechanism requires a callback handler to process invocation result. In this example, it is defined as follows:

```
private class GetQuoteCallbackHandler
     implements AsyncHandler<GetQuoteResponse>{
  private GetQuoteResponse output;
  public void handleResponse
(Response<GetQuoteResponse>response) {
    try {
      output = response.get ();
    } catch (ExecutionException e) {
      e.printStackTrace ();
    } catch (InterruptedException e) {
      e.printStackTrace ();
    }
  }
  GetQuoteResponse getResponse (){
      return output;
  }
}
```

handleResponse() is the only method required by the AsyncHandler interface. Client uses this handler as follows:

```
double result;
......
StockQuote port = new
StockQuoteService().getStockQuotePort();
GetQuoteCallbackHandler callbackHandler = new
GetQuoteCallbackHandler ();
......
Future<?>responseCallback;
......
responseCallback = port.getQuoteAsync ("MHP",
callbackHandler);
Thread.sleep (2000);
result = callbackHandler.getResponse().getReturn();
```

The wildcard interface Future<?> returned to the client may be used to check invocation status, and to cancel the invocation. If a web service operation is enabled for asynchronous invocation, JAX-WS 2.0 requires that three methods be generated in the service endpoint interface. Besides the two asynchronous ones described above, we also have the usual synchronous method. Response time has long been a concern for web services. With the asynchronous invocation mechanism in JAX-WS 2.0, web service client applications on the Java platform will be able to execute other client processes and invoke web services concurrently. This should solve the problem for a large number of use cases.

Call to Web Service StockQuote asynchronously and synchronously is compared. Results obtained are shown in Table 1 and graphically shown in Fig 3.

Table 1: Comparison of asynchronous and synchronous call Performance

Method Called	Time required in msec (Calling asynchronously)	Time required in msec (Calling synchronously)
getQuote	203	832
getPrice	168	756

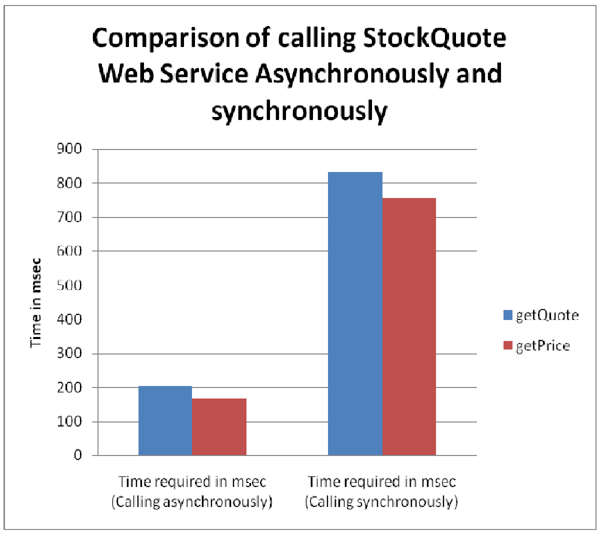

Fig. 3: Performance comparison between asynchronous call and synchronous call

7. Conclusions

Asynchronous web service invocation with a proxy stub in JAX-WS 2.1 is discussed. This new specification also defines a second client programming model, the Dispatch API, which allows a client to interact with a web service at the XML-message level. Asynchronous service invocation can also be performed with the Dispatch API. When web methods of StockQuote Web services called in different way, asynchronous call gives better performance as client need not wait for result from server.

ACKNOWLEDGEMENT

We thank teachers of Department of Computer Science, Shivaji University, Kolhapur for motivating us for this research work. We wish to thank Principal of Thakur College of Science and Commerce, Mumbai and Principal, Y.C. Institute of Science, Satara, for providing resources required to perform experiments to complete this paper.

REFERENCES

[1] JAX-WS Reference Implementation (RI), https://jax-ws.java.net/ Accessed on 15 Feb 2015
[2] Naveen Balani, Rajeev Hathi, Design and develop JAX-WS 2.0 web services, 2007, http://www.ibm.com/developerworks/webservices/tutorials/ws-jax/ws-jax.html
[3] Robert Eckstein and Rajiv Mordani, Introducing JAX-WS 2.0 With the Java SE 6 Platform, Part 1, 2006, http://www.oracle.com/technetwork/articles/javase/jax-ws-2-141894.html
[4] Martin Kalin, Java Web Services: Up and Running, Second Edition, O'Reilly Media 2013
[5] Eric Jendrock, Ian Evans, Devika Gollapudi,Kim Haase, Chinmayee Srivathsa, The Java EE 6 Tutorial, 4th Edition, Addison-Wesley, 2010