Web services programming tips and tricks: Exception Handling with JAX-RPC

Throw the right exception from the service endpoint

Ping Wang
Russell Butek
February 06, 2004

Explicitly declaring faults in WSDL operations, like explicitly declaring exceptions in Java methods, is good programming practice. This tip first examines the exception behavior in the absence of \texttt{wsdl:fault}. It then focuses on how a \texttt{wsdl:fault} is mapped to a checked Java exception and how a JAX-RPC runtime handles this checked exception.

In the SOAP Web services world, a fault flows from the server to the client in the form of SOAP fault. A SOAP fault consists of the faultcode, faultstring, and optional fault actor and detail. The JAX-RPC specification defines various rules about how to map from a Java exception to a SOAP fault (server side) and from the SOAP fault back to the Java exception (client side).

There are four types of exceptions that can be thrown from the server.

- \texttt{java.rmi.RemoteException}
- \texttt{java.lang.RuntimeException}
- \texttt{javax.xml.rpc.soap.SOAPFaultException} (a special, subclass of \texttt{RuntimeException})
- a checked, user-defined exception (mapped from the WSDL's \texttt{wsdl:fault} construct)

The client side will receive one of the following types of exceptions. Note that the client can not catch any \texttt{RuntimeException} other than \texttt{SOAPFaultException}.

- \texttt{java.rmi.RemoteException}
- \texttt{javax.xml.rpc.soap.SOAPFaultException}
- a checked, user-defined exception

This article first discusses the expected behaviour on the client when the server throws various exceptions, and then emphasizes the use of checked exceptions.

\section*{RemoteException}

JAX-RPC requires that all remote methods in a service endpoint interface (SEI) throw the standard \texttt{java.rmi.RemoteException}. This allows exceptions which arise from communications or runtime...
difficulties to propagate back to the caller. However, there is no portable means to send specific subclasses of `RemoteException`.

The application itself could also throw a `RemoteException`. However, since there is no portable means of sending specific `RemoteExceptions`, the client cannot catch specific `RemoteExceptions`. For a given SOAP fault returned from the server, different client-side JAX-RPC runtimes may have different interpretations and generate different `RemoteExceptions`. Because of this interoperability problem, the application should avoid throwing `RemoteExceptions`.

**RuntimeException**

When a problem occurs in a server-side JAX-RPC runtime which results in a `RuntimeException` being thrown (for example, `NullPointerException`), that exception will propagate back to the client, but it will do so as a SOAP fault. The client runtime will map SOAP fault to either `RemoteException` or `SOAPFaultException` (described below). Therefore, a service endpoint should not throw a `RuntimeException` expecting the client to always catch that `RuntimeException` because the client may receive a `RemoteException` instead.

**SOAPFaultException**

There is one special `RuntimeException`: `javax.xml.rpc.soap.SOAPFaultException`. `SOAPFaultException` is more descriptive than a `RuntimeException` and dictates the exact SOAP fault message which flows to the client. In other words, whoever throws this fault, whether the runtime or the application, controls the SOAP fault response. Therefore, how to map the SOAP fault to an appropriate exception really depends on the content of `SOAPFaultException`, it may be mapped to `SOAPFaultException`, `RemoteException` or even a checked user exception. `SOAPFaultException` is often used by JAX-RPC handlers. A JAX-RPC application itself normally should avoid throwing the `SOAPFaultException`.

**Checked user exception**

A good programming practice often involves explicitly defining checked user exceptions as part of the interface contract. In the JAX-RPC world, programmers need to first define `wsdl:fault` as part of a `wsdl:operation`. A `wsdl:operation` allows multiple `wsdl:fault` elements, just like a Java method allows multiple exceptions. Each `wsdl:fault` is mapped to a user exception as part of the SEI. In most cases, Java exceptions do not have complicated data structures; similarly for `wsdl:fault`, the schema definition referenced by the `wsdl:fault` is often straightforward. Nevertheless, it's still very important for programmers to think over which kind of exceptions are expected to be thrown, and then define appropriate `wsdl:faults`.

**Mapping rules**

Unlike `wsdl:input` and `wsdl:output`, the message referenced by `wsdl:fault` is only allowed a single message part which could refer to a simple type or a complex type. If the part element has a type attribute, then you can tell directly whether the type is simple (for example, `xsd:int`, `xsd:string`, etc.) or complex. If the part element has an element attribute, then you have to step to the element to see whether the type is simple or complex.
Mapping rules for simple types

For a simple type, the Java exception name is mapped from the name attribute of the 
wsdl:message element. The wsdl:part name is mapped to a getter method and a parameter in 
the constructor of the Java exception. For example, the fault information in the WSDL in Listing 1 
maps to the Java language exception in Listing 2.

Listing 1. WSDL definition with a simple fault

```xml
<definitions ...>
    <message name="empty"/>
    <message name="InsufficientFundsFault">
        <part name="balance" type="xsd:int"/>
    </message>

    <portType name="Bank">
        <operation name="throwException">
            <input message="tns:empty"/>
            <output message="tns:empty"/>
            <fault name="fault" message="tns:InsufficientFundFault"/>
        </operation>
    </portType>
</definitions>
```

Listing 2. Java exception from the fault in Listing 1

```java
public class InsufficientFundFault extends java.lang.Exception {
    private int balance;
    public int getBalance() {
        return this.balance;
    }

    public InsufficientFundFault() {
    }

    public InsufficientFundFault(int balance) {
        this.balance = balance;
    }
}
```

Mapping rules for complex types

For complexTypes, the Java exception name is mapped from the name of the complexType (or 
the name of the element if the fault message's part refers to an element). Each element inside 
the complexType is mapped to a parameter in the constructor of the Java exception and a getter 
method. Note that, unlike beans, there is no setter method. The only way to set such a field is 
through the exception constructor. For example, the fault information in the WSDL in Listing 3 
maps to the Java language exception in Listing 4.

Listing 3. WSDL definition for a complex fault

```xml
<definitions ...>
    <types>
        <schema xmlns="http://www.w3.org/2001/XMLSchema">
            <element name="InsufficientFundFault">
            </element>
        </schema>
    </types>
</definitions>
```
Listing 4: Java exception from the fault in Listing 3

```java
public class InsufficientFundFault
    extends java.lang.Exception
    implements java.io.Serializable {
    private int balance;
    private int requestedFund;

    public InsufficientFundFault(int balance, int requestedFund) {
        this.balance = balance;
        this.requestedFund = requestedFund;
    }

    public int getBalance() {
        return balance;
    }

    public int getRequestedFund() {
        return requestedFund;
    }
}
```

Mapping rules for fault inheritance

Suppose you want to define a subclass of `InsufficientFundFault`, called `AccountInsufficientFundFault`, to carry the account number of the requesting client (see Listing 5). Your application can throw this subclass exception and your client would receive this exception. Very straightforward.

Listing 5: WSDL faults with inheritance

```xml
<wsdl ...>
```
By doing this, the method "withdraw" of the generated SEI stays the same (see Listing 6). One item to note, however, is that any subclass of `InsufficientFundFault` must appear in the WSDL file for the client to receive that exception. You cannot create a new subclass of `InsufficientFundFault` in the server-side Java component without doing the same in the WSDL file. WSDL is a declarative language. Every possible fault that a service can throw must be explicitly defined in the XML, otherwise the client will not know about it and will not be able to receive it.

**Listing 6: Java SEI mapped from the above WSDL definition**

```java
public interface Bank extends java.rmi.Remote {
    public boolean withdraw(java.lang.String account, int amount)
        throws java.rmi.RemoteException,
        example.InsufficientFundFaultType;
}
```

**SOAP fault content**

The JAX-RPC runtime catches a user exception and serializes it to XML data based on the schema definition referenced by the message part of the `wsdl:fault`. Such XML data is used to
fill in the content of the `detail` element of the SOAP fault. **Listing 7** is the SOAP message for the complex `InsufficientFundFault` example. The SOAP message for a simpleType fault is similar except that the `detail` section is different.

**Listing 7: SOAP Fault example**

```xml
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <soapenv:Fault>
      <faultcode>...</faultcode>
      <faultstring>...</faultstring>
      <detail>
        <InsufficientFundFault xmlns="http://example">
          <balance>1000</balance>
          <requestedFund>2000</requestedFund>
        </InsufficientFundFault>
      </detail>
    </soapenv:Fault>
  </soapenv:Body>
</soapenv:Envelope>
```

The key thing here is that the `detail` section carries the content which must match the schema definition referenced by `wsdl:fault`. In this way, the client runtime will know which user exception it should be mapped to. Also note that the SOAP fault does not carry the exception stack trace as you normally expect for the Java exception; therefore, a Web services client should not expect to see the stack trace originating from the server side.

**Summary**

It is good programming practice to introduce user-defined faults. Using `RemoteException`s or `RuntimeException`s is not only too general, there is also no guarantee that every vendor will handle these in the same manner.

Once you decide to introduce user-defined faults, you must decide what kinds of faults to use -- faults of simple types, faults of complexTypes, or an inheritance tree of faults -- and you must understand how those faults map to Java programming artifacts.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code for this article</td>
<td>DoNotLeaveThisLink.zip</td>
<td>40KB</td>
</tr>
</tbody>
</table>
Related topics

- The "Web Services Description Language (WSDL) 1.1" is the specification of WSDL.
- Download the source code used in this article.
- "Java API for XML-Based RPC (JAX-RPC) Downloads & Specifications" provides links to the JAX-RPC 1.0 specification itself, as well as javadocs, class files, and Sun's JAX-RPC reference implementation.
- Read the SOAP 1.1 specification to see how SOAP faults appear in a SOAP message.