Best practices for service interface design in SOA, Part 2: Using services to report errors to service consumer applications

Mikhail Genkin

August 21, 2007

In this article, learn how to report service errors using service interfaces in an SOA environment. Service-Oriented Architecture (SOA) emphasizes loose coupling between different systems within an enterprise. Applications communicate with each other only with well-designed service interfaces and remain unaware of each others' implementation. Service interface structure is of primary importance in SOA. Poorly designed service interfaces can have a negative effect on all applications that need to use them. Find out how well-designed service interfaces help accelerate project schedules and make your SOA solution more responsive to business needs.

View more content in this series

Introduction

Part 1 of this series focused on best practices for service interface design, including design and development approaches, service granularity, synchronous compared with asynchronous design, and operation signatures.

In this article, best practices are highlighted to explain how services should report errors to service consumer applications. Production-quality systems must be able to handle a wide variety of error conditions, in addition to successful responses. The way a service reports error information to service consumer applications impacts the structure of the service interface, and consequently, the way service consumer applications are constructed.

WSDL constructs for reporting errors

Part 1 established that the best way to describe the service interface is using Web Services Description Language (WSDL). The WSDL specification stipulates that:

- A service interface (port type) contains a number of operations.
• Each operation can be request-response or one-way.
• Each request-response operation can define a single input message, a single response message, and any number of fault messages.
• One-way operations cannot define response messages or fault messages.

The WSDL specification further defines that messages can contain one or more parts. The value of each part can be either a simple XSD type, or a user-defined complex type. See Related topics for a link to the WSDL specification.

Types of errors

Your service needs to report errors that essentially come in two flavors:

System level
Represent failures of run time software components, hardware, and network communication protocols that are part of your service implementation. These errors represent failures that are not related to the business logic or data that is being executed.
For example, a typical system-level error occurs if the RDBMS server that your service implementation needs to accesses crashes, and your server cannot process the request message because the application server can't allocate a Java™ Database Connectivity (JDBC) connection to that server.

Business level
Errors that your service raises in response to violations in business logic or data. For example, if your request message represents a request to book a trip and the return date field indicates an earlier date than the departure date field, your service implementation will likely want to raise a business-level error.

Best practice: For synchronous architectures with request-response operations, define both system and business faults in your operation signature. System faults and business faults should be described by different XSD types.

Service consumer applications typically need to respond differently to system-level and business-level errors. With a system-level error, the service consumer application might want to wait and retry making the request with the original data. In the case of a business level error, the application will likely want to send a message back to the end user requesting that the input errors be corrected.

Using WSDL faults in request-response operations

Faults can be used with request-response operations only. Figure 1 shows an example of how a WSDL port type can be structured to return business-level and system-level errors to client applications. The operation is based on the popular StockQuote example. The StockQuote service contains a single StockQuote port type (interface). This port type contains an operation called getQuote(). The operation receives the input parameter as a part of the getQuoteRequest message. The result is returned as a part in the getQuoteResponse message.
The `getQuote()` operation also defines two messages to return faults to the service consumer applications:

- `getQuoteBusinessFault`. Returns errors related to incorrect usage of the service, such as unknown or empty stock symbol provided by the service consumer application.
- `getQuoteSystemFault`. Returns errors caused by the system, such as database connection problems.

Following the best practices in Part 1, the error types are defined in a separate XSD file. This XSD contains complex types to represent a `SystemError` and a `BusinessError`.

**Best practice:** A fault message should contain a single WSDL part. The value of this part is a complex XSD type containing a complete description of the error.

The example port type in Figure 1 above shows a request-response operation that uses faults to return errors. The structure of the operation is shown as viewed in the WSDL editor provided with IBM® WebSphere® Integration Developer 6.0 or with IBM Rational® Application Developer 6.0 or 7.0.

Listing 1 shows the structure of the complex types defined in the XSD file. You can download the sample code, which shows how complex error types are defined. `BusinessErrorType` and `SystemErrorType` are derived by extension from a common base type: `BaseErrorType`. `BaseErrorType` defines two elements:

- `faultName`. Conveys the application-specific name of the error to the service consumer application. For example, if the service consumer application passes in a null or empty string for the stock symbol parameter, the fault name returned to that application could be "Invalid symbol format fault."
- `message`. Provides a deeper explanation and indication of corrective action to be taken. For example, "Null or empty symbol. Please pass a valid stock symbol to the getQuote operation."

The `faultName` and `message` elements are common to any type of fault that can be raised by the `getQuote` operation and therefore, are defined as the base type for all errors.

**Listing 1. Excerpt from ErrorTypes.xsd**

```xml
<complexType name="BaseErrorType" abstract="true">
```

Best practices for service interface design in SOA, Part 2: Using services to report errors to service consumer applications
SystemError and BusinessError types both define an errorCode element. This element contains an alphanumeric code that uniquely identifies each type of error that can be returned by the service. It's important for services to return an alphanumeric code, along with textual information describing the error, because client applications (service consumers) might want to programmatically examine this code and direct subsequent invocations differently, depending on the type of error as indicated by the error code.

**Best practice:** Fault messages should contain an error code that allows the service consumer applications (clients) to process errors programmatically. Error codes should be published as an integral part of the WSDL service definition to make it easier for service consumer applications to understand the meaning of reported errors.

**Listing 2** shows definitions for simple types used to convey error codes to the service consumer applications. Two simple types, BusinessErrorCodeType and SystemErrorCodeType, are derived from the xsd:string type by restriction. Enumerations listed in the restriction define error codes that can be returned for business-level errors and system-level errors. Comments embedded in the XSD describe the meaning of error codes to service consumer (client) application developers.
This example groups business-level error codes and system-level error codes in different types to explicitly separate them and to facilitate programmatic processing of error codes by service consumer applications, if necessary. It would be sufficient in many cases to define a single simple type to contain all error codes describing business level and system level error semantics. However, it is definitely advisable to group business and system error codes (for example, business level errors 00010 to 00100 and system level errors from 001000 to 010000) into separate numeric ranges to facilitate programmatic processing of these codes.

In cases where performance is critical even on exception paths, you may want to make your error code fields of type int or short, as comparisons on numeric fields are faster than on string fields.

**Best practice:** Numeric values for system and business errors should be grouped in different ranges (see Download) to facilitate programmatic checking by service consumer application code.

The SystemErrorType type contains optional elements to capture information about the error that started the problem. For example, the StockQuote service may catch a `java.sql.SQLException` when trying to communicate with the back-end database. In response to this, our service constructs and throws a SystemErrorType. The SystemErrorType should contain the name of the original exception and, optionally, additional messages stack trace information from the `java.sql.SQLException` exception that started the problem.

In most cases, service developers might choose not to report original errors to the service consumer applications if the service consumer and the service producer applications belong to the same organization. This is a good practice that facilitates problem determination and resolution.

**Returning errors in the operation response**

An alternative to using WSDL faults is to return error information as part of the regular response message. Application developers tend to favor this approach, particularly when batch processing with Web services is involved. Batch processing in this case applies to operations with input messages that contain multiple distinct inputs and response messages than can contain multiple distinct results, each corresponding to an input. For example, an input message to the StockQuote service...
service could contain multiple symbols and the response message would contain multiple results. This approach involves a number of drawbacks and benefits. (An upcoming article in this series covers issues related to batch processing in more detail.)

The primary benefit of returning error codes in the response rather than the fault is that it allows the service consumer application (client) to process error information in the same manner as nonerroneous responses. Figure 2 shows how error information can be returned in this manner.

The client application can then take the response Java object that is returned by the Web service and simply pass it to the presentation tier (usually JavaServer Pages) for display to the end user. This avoids additional coding involved in creation of transfer objects that could be used to decouple the presentation tier of the service consumer application from the Web services tier.

### Figure 2. One-way and request-response operations that return error information as part of response message

Significant drawbacks with this approach include:

- The client application has to access the response, check to see whether the result element provided in the response is null, examine and return `systemError` and `businessError` values, and react appropriately.

  Business logic code gets cluttered with many lines of technical look-up and conditional statements. Business processes written in WSBPEL can be particularly afflicted, in a bad way. Web Services Business Process Execution Language (WSBPEL) provides fault-handler constructs that can be neatly separated by the graphical builder tool from the main sequence of business logic activities. If the WSBPEL flow has to continuously introspect the response structure and evaluate the value of the result and error fields, the resulting graph is very hard to present and explain to business users.

- The user interface of the service consumer application (client) code becomes tightly coupled to the service interface. Future changes to the service interface may necessitate a rewrite of the user interface.

  **Best practice:** For request-response operations that don't involve batch processing, use WSDL faults to return error information from your service interface to service consumer applications, rather than return error information in the response message and status code.

- The application server on which the service consumer application runs will not be aware of problems being raised by the Web service. All responses that return a result will appear to be error-free invocations, and no indication of problems will display in the trace files or `system.out` logs, thereby hampering problem determination.
Because of these drawbacks, the best practice for returning errors and messages from a request-response operation is to rely on the WSDL fault construct and to return errors as faults using the approach just described.

**Returning errors in asynchronous topologies**

The WSDL specification (see [Related topics](#)) does not allow you to define faults for one-way operations. However, applications built around the asynchronous model do need the ability to access error information.

Asynchronous invocations can be structured in one of three styles:

- **One way asynchronous invocation without response**
  A publish-subscribe interaction is an example of this style. When applications implement one-way asynchronous invocation without response, there is no way to provide error feedback to the service consumer application because the WSDL specification does not allow definition of a fault for one-way operations that are used. The service consumer application is also not interested in obtaining the response at a later time, so there is no way to provide errors to the client using the response message. In this case, you have to pay very careful attention to infrastructure configuration to ensure that the request message is not lost on the way to its intended destination.

  **Best practice:** When implementing asynchronous topologies, return faults using the response message.

- **Asynchronous invocation followed by request for response at a later time**
  Involves execution to two request-response operations, although the bulk of the processing is performed by the target service asynchronously. For example:

  1. The service consumer application invokes the `submitStockQuoteRequest()` operation. This operation returns a transaction confirmation number to the service consumer application.
  2. The `stockQuoteService` implementation performs the look-up of the stock value.
  3. The service consumer application invokes the `getStockQuoteResponse()` operation at a later time, and passes in the transaction number returned with `submitStockQuoteRequest()`. The response message returned by this operation contains both the stock value result or error information, if any.

  **Figure 3** shows how operation signature should be structured in this case. The request operation can raise system faults. These faults convey errors related to delivery of input parameters and initiation of asynchronous processing only. The operation used to retrieve the response message should raise both system and business faults in the manner described for regular request-response operations above.
Figure 3. Structure of StockQuoteAsynch port type

- Asynchronous invocation with callback to return the request
  The service consumer application provides information that allows the service to send the response message back to an interface defined by the client (service consumer) application. If a business error occurred during the asynchronous processing stage, the service implementation raises the business fault when queried for the response message.

**Best practice:** When initiating asynchronous processing with request-response operations, define system faults only on the request operation and both system-level and business-level faults on the operation used to retrieve the response message.

**Handling errors in client applications**

Listing 3 shows an example of how a client application would handle business and system errors raised by the StockQuote service.

The Java API for XML-based RPC (JAX-RPC) and Java API for XML Web Services (JAX-WS) specifications describe how types described by the WSDL and XSD are converted into Java objects that are used by the client. Because we defined different complex types to represent these errors, the Rational Application Developer Web services tooling generates different Java beans to represent them. These beans derive from the Java Exception class and can be handled in different catch blocks.

**Best practice:** When invoking Web services, catch all declared faults and handle system and business errors in separate catch blocks.

When the client application invokes the Web service, it has to handle exceptions defined in the method signatures of the generated proxy. Along with these exceptions, the client stack may also raise additional errors related to serialization and communication. It is generally a good idea to handle base `java.lang.Throwable` in addition to exceptions defined in the proxy method signature. This ensures that your client application code is rugged enough to handle unforeseen exceptions that may be raised by the client stack in response to, for example, malformed response messages (serialization type errors).

Listing 3. Example of client-side error handling for request-response processing

```java
public class SQFacade {
    private int retryCount = 0;
```
private StockQuoteProxy sqProxy = new StockQuoteProxy();

public String getQuote(String symbol) throws UIError
{

String result = null;
UIError error = new UIError();

try {
    result = sqProxy.getQuote(symbol);
} catch (BusinessErrorType e) {

    // Check the code and return appropriate action to user
    BusinessErrorCodeType errorCode = e.getErrorCode();

    if (errorCode != null)
    {
        if (errorCode.getValue().equals(BusinessErrorCodeType._SQ00010))
        {
            error.setMessage("Invalid input format!");
            error.setAction("Please re-enter the symbol value correctly.");
        }
        else if(errorCode.getValue().equals(BusinessErrorCodeType._SQ00020))
        {
            error.setMessage("Stock symbol not found!");
            error.setAction("Please enter a different stock symbol.");
        }
        else // Just in case!
        {
            error.setMessage("User error!");
            error.setAction("Please contact site administrator.");
        }
    }
    else // Just in case!
    {
        error.setMessage("User error!");
        error.setAction("Please contact site administrator.");
    }

    throw error;
}

catch(SystemErrorType e)
{

    // Check the code and return appropriate action to user
    SystemErrorCodeType errorCode = e.getErrorCode();

    if (errorCode != null)
    {
        if (errorCode.getValue().equals(SystemErrorCodeType._SQ001001))
        {
            error.setMessage("Authentication problem!");
            error.setAction("Please please contact the site administrator.");
        }
        else if(errorCode.getValue().equals(SystemErrorCodeType._SQ001002))
        {
            // Retry 3 times to see if db connection problem clears up
            if (retryCount < 3)
            {
                retryCount++;
                this.getQuote(symbol);
            }
            retryCount = 0;
        }
    }

}
Best practice: While catching system and business errors, also catch the base error type java.lang.Throwable to deal with any unexpected error conditions raised by the client stack.

Listing 3 shows one approach to implementing best practices described in this article. Our hypothetical client application defines a simple façade class called SQFacade, which has a method called getQuote(). The signature of this method is very similar to the signature of the getQuote() method defined by the Web service proxy, but it throws a different simple error type (UIError) that is used to decouple UI code from the generated proxy code and return information more meaningful to the user.

The SQFacade class encapsulates exception-handling logic. It uses a recursive approach to retry the call three times in case of system-level database connection problems.

Summary

Web services provide an ideal technology for implementing an SOA. In this article you learned best practices for how services should report error information to service consumer applications. Applications should seek to leverage the fault mechanism provided by the WSDL specification when performing request-response processing, and use the response message to report errors when performing asynchronous invocations.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error handling code</td>
<td>ar-servdsgn2code.zip</td>
<td>3.113KB</td>
</tr>
</tbody>
</table>
Related topics

- Part 1 of this series, "Exploring the development, interfaces, and operation semantics of services," focuses on best practices for service interface design, including high-level aspects of development approaches, service granularity, and operation signatures.
- "Service-Oriented Architecture expands the vision of Web services, Part 1: Characteristics of Service-Oriented Architecture" (developerWorks, Apr 2004) discusses moving forward from simple models to those that represent real-world business models of arbitrary complexity.
- Read how to "Implement implicit and explicit SOAP headers" (developerWorks, Feb 2005).
- "Web services programming tips and tricks: Using SOAP headers with JAX-RPC" (developerWorks, Oct 2003) explains how to create and process information that is transferred in the header portion of a SOAP message.
- Read about addressing and security in the Web Services Description Language (WSDL) 1.1 specification.
- Learn more about soap:header and soap:headerfault elements.
- Trial download: Rational Application Developer for WebSphere Software.
- Trial download: Rational Software Architect.
- Get more information about WebSphere Integration Developer.