

The SOA Magazine

Feature Article



Service Error Content Patterns (Part II)

by Ronald Murphy

Published: August 30, 2009 (SOA Magazine Issue XXXI: August 2009)

[Download PDF](#)

[Digg This](#) • [De.licio.us](#) • [Slashdot](#) • [Technorati](#) • [StumbleUpon](#) • [Google Bookmark](#)

Abstract: In the previous article of the two-part "Service Error Content Patterns" article series, we established some of the common problems and challenges when working with standard exceptions in messaging environments (Web services and SOAP messaging in particular). In this second series installment of the introduce a set of proposed solutions and techniques for addressing these issues. Specifically, we introduce the Response-Resident Error Pattern, Merged Format Pattern, and the Mixed Usage Pattern. Each of these patterns addresses a different exception problem and scenario. By applying proven design techniques, such as those documented in these three patterns, we can take full control of exception conditions. This leads to a much more sophisticated and flexible enterprise solution design. Furthermore, as patterns, these design techniques are repeatable in that they can be applied to a variety of solution architectures and message exchange frameworks and scenarios.

Response-Resident Error Pattern

The response-resident error pattern works around the limitations of using faults, by providing for all error information to be modeled as part of your standard operation response messages. Although this does send error modeling somewhat underground from a WSDL standpoint, they can address many of the limitations we identified with fault-based modeling

1. For a given Web service or group of services that you are designing, you can define a bigger "lowest common denominator". You can standardize on a richer error structure to use for all your errors. You can still occasionally extend this structure with optional information used by some services or operations.
2. By having an error structure in your responses, you can mix normal data with response output. This allows for warnings.
3. Your standard error structure can provide for multiple errors. Sooner or later, you will probably want this in a larger Web service. For example, compound operations can easily produce multiple errors.

One feature you may be giving up when using response-resident errors is the automatic re-throwing of faults as exceptions on the client side. Many SOAP toolkits do provide this feature, and in some client application environments, it may be a convenient way to terminate processing. But the sacrifice is small in return for the flexibility you might gain around handling warnings and multiple errors.

Here is an example WSDL pattern for response-resident errors. First, we declare a base response type to hold error information in a standard way across all responses for all our service operations. (This could also hold any other standardized information for responses, such as a timestamp of when the response was generated, the version of the Web service or operation, etc.)

```
<xs:complexType name="BaseServiceResponseType">
  <xs:sequence>
    <xs:element name="Status" type="tns:ServiceStatusType" />
  </xs:sequence>
</xs:complexType>
```

```

<xs:element name="ErrorList" type="tns:ServiceErrorType"
  minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>

```

Next, we declare the response for FindItem specifically, deriving from the base response type:

```

<xs:element name="FindItemResponse">
  <xs:complexType>
    <xs:complexContent>
      <!-- Derived from base response type -->
      <xs:extension base="tns:BaseServiceResponseType">
        <xs:sequence>
          <!-- Result data -->
          <xs:element name="Item" type="tns:ItemType"
            minOccurs="0"
            maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

```

Note that this response now contains both error and result data. In this example, the error data is picked up from the base response class, while the result data is declared specifically to this type of response.

The following WSDL declaration references the FindItemResponse element as the response message. Unlike the fault example, it does not reference a fault message:

```

<wsdl:operation name="FindItem">
  <wsdl:input ... />
  <wsdl:output message="tns:FindItemResponse"/>
</wsdl:operation>

```

The impact of the response resident error style on client software can vary, depending on how clients are built. Generated clients, such as some test tool instrumentation based on your WSDL, may not understand your error at all without customization - and such customization is not possible across all available test tools.

However, in practice, most clients will be applications custom built for your service, either by you or by your customers. Clients usually have to understand most of your service's response format in order to use your service in the first place. If you provide a standardized "lowest common denominator" error structure in your response, you can steer clients toward a centralized software function that will decode and display this error format at minimum.

Response-resident errors are a reasonable approach when standard tooling and standardized clients do not pose an obstacle. In fact, the less standardized your overall Web service ecosystem, the more likely you will want to use response-resident errors. For example, REST style service usage treats service messages as bare payload exchanged over HTTP - there is no SOAP envelope to contain your fault. You will have to either provide a bare fault-like message payload, or just standardize on a single kind of message that unifies response and error information.

What information should go into response-resident errors? Since this pattern gives you the freedom to model warnings alongside critical (terminating) errors, you may want to build provisions to distinguish the two. Clients might want to treat warnings in a more cursory way, by logging them or showing them in an unobtrusive information area. In contrast, errors usually must terminate any outstanding client-side end user activities. The following enumerated type can be featured in your errors to allow you to distinguish these two kinds of errors:

```

<xs:simpleType name="SeverityType">
  <xs:restriction base="xs:token">
    <xs:enumeration value="Critical"/>
    <xs:enumeration value="Warning"/>
  </xs:restriction>

```

```
</xs:simpleType>
```

We also suggest you provide not only a string error message in your error, but also a unique code (preferably an integer) to designate each specific error instance. Clients might want to take special actions on particular errors, for example, if it turns out that a bug causes an error frequently, they may introduce workaround code or special handling (extra logging, etc.). In most cases, application coders won't go to the trouble of tracking all your error codes, but the approach can really help in a pinch. This can aid your technical support process, as well, by succinctly identifying a particular error. The following definition collects together our recommendations for a minimum error structure:

```
<xs:complexType name="ServiceErrorType">
  <xs:sequence>
    <xs:element name="ErrorMessageString" type="xs:string"/>
    <xs:element name="ErrorCode" type="xs:int"/>
    <xs:element name="Severity" type="tns:SeverityType"
      minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

For responses themselves, since the response-resident error pattern calls for these structures to contain such a variety of data, you'll want client applications be able to easily distinguish what they are receiving. Consider having an enumerated status element in all your responses. This can have, at minimum, the values "success" and "failure". It is also useful to indicate "warning", meaning "successful execution with warnings present". Here's an example definition of this enumeration:

```
<xs:simpleType name="ServiceStatusType">
  <xs:restriction base="xs:token">
    <xs:enumeration value="Success"/>
    <xs:enumeration value="Failure"/>
    <xs:enumeration value="Warning"/>
  </xs:restriction>
</xs:simpleType>
```

This enumerated type is featured in the base response type we first showed up above in this section.

One final word of caution: If you choose the response-resident error approach, you will usually find server programming error cases that still produce SOAP faults: Stray exceptions may be caught by your SOAP engine and reported automatically as faults, and any issues within the engine itself will certainly be emitted as faults by default. Many solutions can be customized at some level to divert the flow back to a response level flow, but you'll need to research this area if you deviate from faults.

Merged Format Pattern

One hypothetically interesting error content pattern is to simply unify response and fault data structures and use the same structure for both. We don't know of any Web service standard prohibiting this usage, although it does seem to deviate from the spirit of the fault concept, and it's likely some SOAP toolkits may trip over the approach.

To use merged response/fault formats, for each operation, your WSDL would declare "output" and "fault" messages of the same XML schema type. This shared type is a structure holding both normal response information and error information - so it should use definitions similar to our response-resident error section, such as `BaseServiceResponseType` and `FindItemResponse`. The type might vary from operation to operation, but for any given operation, it would serve as both a fault structure and an output structure. The key twist in the merged approach is the duplicated reference to the response message in *both* the `wsdl:output` and `wsdl:fault` definitions, such as:

```
<wsdl:operation name="FindItem">
  <wsdl:input ... />
  <wsdl:output message="tns:FindItemResponse"/>
  <wsdl:fault message="tns:FindItemResponse"/>
</wsdl:operation>
```

On closer examination, this approach has issues, however. In theory, it consolidates handling for both server and client, allowing for uniform processing of data. But remember that SOAP implementations are supposed to return fault messages as HTTP 500 errors, and that most application servers also return HTTP 500 errors in other situations. A particular client won't necessarily have the intelligence to separate faults from out of memory errors or other non-SOAP-related problems.

If the server happens to violate the WSI-BP rule and return faults using a normal HTTP 200 error (as some implementations do), then clients have a different problem - they can't really distinguish faults from normal output. Clients would need to look for presence of an error structure, a flag, or some other non-standard indication that an error is present. In fact, from the point of view of the client, a merged fault/response structure returned with HTTP 200 is basically a response-resident error approach! The client can't tell the difference.

Overall, the strategy of merging response and fault information flows places a few too many requirements on the client side for it to really be a top choice. If you need to support a variety of clients and don't control their design and implementation, this approach is probably not the best.

Mixed Usage Pattern

Many larger scale implementations may wind up using some form of mixed approach, a blend of fault and response-resident error modeling. The mixing of faults and response-resident errors can appear as an unwanted guest strategy, if a Web service modeler starts by using faults, and later finds a use case that requires warnings or multiple-error payloads. But preferably, mixed fault and response usage should be planned into schemas as a conscious design choice where appropriate.

Earlier, we mentioned that faults can be either declared in WSDL, or left to be thrown generically should unexpected processing trouble arise. This separation can also be applied to a mix of response-resident errors and faults. Model your expected error information as response resident errors, and leave faults as a "fall-through" measure to cover unexpected processing glitches. See Figure 4.

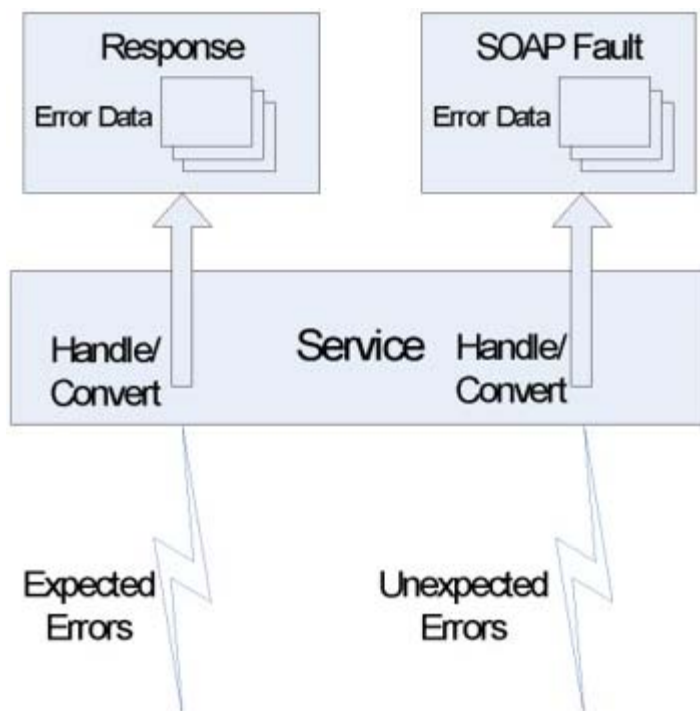


Figure 4: Handling Unexpected Exceptions with SOAP Faults.

In fact, the "fall-through" approach to using faults can be the path of least resistance. Recall that SOAP engines like to handle their error conditions as faults, as we discussed in the response resident error section. You can define the sphere of "expected" errors to be your own business logic, and the realm of "unexpected" errors to be SOAP infrastructure and possibly any supporting software (data tier, etc.) underneath your business logic. All the errors you directly control become response-resident errors, and everything else becomes a SOAP fault. The best of both worlds! - Almost. One common processing area that produces

faults in SOAP engines is the parsing of XML content. Any syntax errors encountered during parsing are likely to be reported as faults, so that clients may have to process some of their formatting errors as faults, and other (business constraint) errors as response-resident errors. There is really a difference in these two kinds of data error anyway, since syntax errors usually indicate incorrectly coded client applications. So in the final analysis, it is hard to find "fault" with the approach!

The dual error format of the mixed usage approach is really its main drawback, seen from the point of view of clients. These client applications will have to deal with a variety of error conditions - first of all, transport and application server-related errors, and then due to the choice of the mixed strategy, both faults and response-resident errors. Alas, this is really only a slight extra complication in an already complicated error handling world.

We have one bit of advice to smooth the mixed error processing road for clients: If possible, standardize your error structures within both forms of error. A repeating error element with at least a severity, error code, and message string is usually a good top-level structure. For your responses, you can define a base response type that always features the same repeating error element as well (along with an overall status field, as we discussed earlier). For any faults that you model yourself, you can include this same repeating element in your proprietary fault details. This allows clients to parse the same structure in as many error cases as possible.

As a final consolation for clients, most of the transport, application server, and fault related errors will surface at HTTP level, through HTTP 400 and 500 series status codes. Client error handling can report the status codes, along with some of the text from the HTTP response - this will usually be enough to diagnose the issue.

Conclusion: Recap of Patterns

Here is a quick summary of the various approaches we've discussed:

- In simple error situations or RPC style Web services, exceptions and faults might be a fine solution to your negative use cases. Moreover, fault-based reporting mechanisms have the advantage of being formalized in SOAP and WSDL, and exceptions and faults are well integrated with one another in most mainstream SOAP programming environments. Even your standard SOAP tooling will lead you down the fault-oriented path. So if you don't have fancy error processing and don't ever expect to, faults can be a simple and practical choice.
- In more sophisticated Web services where we want to provide for response/error mixtures, consider using the response-resident error pattern. REST-style services push us into this choice almost by definition, since faults are a SOAP feature. The main drawbacks for the response-resident error approach are the lack of a standardized specification method in WSDL, and the fact that some tooling such as test tooling may be unable to detect your errors without customization
- Although theoretically you could fold error messages and fault messages together into a single message type, you should be careful about doing so, because it goes against the grain of conventional usage, and may confuse some client applications.
- If you don't mind exposing clients to a little more error variety than they might otherwise see, you can return both faults and response resident errors, letting your specific use cases drive the decision. Be sure you document the circumstances leading to each kind of error, so that client applications can sensibly respond.

Summary for the fault example:

```
<!-- Request information for FindItem operation -->
<xs:element name="FindItemRequest">
  <!-- (keywords to search for, etc.) -->
</xs:element>

<!-- Response information for FindItem operation - has result data -->
<xs:element name="FindItemResponse">
  <xs:complexType>
    <xs:sequence>
```

```

    <xs:element name="Item" type="tns:ItemType"
      minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
</xs:element>

<!-- ItemNotFound fault, thrown by FindItem operation -->
<xs:element name="ItemNotFound">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="tns:BaseServiceFaultType">
        <xs:sequence>
          <!-- specific fault data here -->
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<!-- Base fault to be used across a series of faults -->
<xs:complexType name="BaseServiceFaultType">
  <xs:sequence>
    <xs:element name="ErrorDetails"
      type="tns:ServiceErrorType"/>
  </xs:sequence>
</xs:complexType>

<!-- Message definitions for FindItem request, response, and fault -->
<wsdl:message name="FindItemRequest">
  <wsdl:part name="FindItemRequest" element="tns:FindItemRequest"/>
</wsdl:message>
<wsdl:message name="FindItemResponse">
  <wsdl:part name="FindItemResponse"
    element="tns:FindItemResponse"/>
</wsdl:message>
<wsdl:message name="ItemNotFound">
  <wsdl:part name="ItemNotFound" element="tns:ItemNotFound"/>
</wsdl:message>

<!-- Operation definition for FindItem operation -->
<wsdl:operation name="FindItem">
  <wsdl:input message="tns:FindItemRequest"/>
  <wsdl:output message="tns:FindItemResponse"/>
  <wsdl:fault message="tns:ItemNotFound"/>
</wsdl:operation>

```

Summary for the response-resident error example:

```

<!-- Base response type to be used across all operations -->
<xs:complexType name="BaseServiceResponseType" abstract="true">
  <xs:sequence>
    <xs:element name="Status" type="tns:ServiceStatusType"/>
    <xs:element name="ErrorList" type="tns:ServiceErrorType"
      minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<!-- Response information for FindItem operation -->
<xs:element name="FindItemResponse">
  <xs:complexType>
    <xs:complexContent>
      <!-- Derived from base response type -->
      <xs:extension base="tns:BaseServiceResponseType">
        <xs:sequence>
          <!-- Result data -->
          <xs:element name="Item" type="tns:ItemType"
            minOccurs="0"

```

```

        maxOccurs="unbounded"/>
    </xs:sequence>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>

<!-- Common enumeration to show result status of any operation -->
<xs:simpleType name="ServiceStatusType">
  <xs:restriction base="xs:token">
    <!-- The operation succeeded with no errors or warnings -->
    <xs:enumeration value="Success"/>
    <!-- The operation failed; processing must stop -->
    <xs:enumeration value="Failure"/>
    <!-- The operation succeeded with warnings present -->
    <xs:enumeration value="Warning"/>
  </xs:restriction>
</xs:simpleType>

<!-- Operation definition, with no fault declaration -->
<wsdl:operation name="FindItem">
  <wsdl:input message="tns:FindItemRequest"/>
  <wsdl:output message="tns:FindItemResponse"/>
</wsdl:operation>

<!-- Common type to carry error details -->
<xs:complexType name="ServiceErrorType">
  <xs:sequence>
    <xs:element name="ErrorMessageString" type="xs:string"/>
    <xs:element name="ErrorCode" type="xs:int"/>
    <xs:element name="Severity" type="tns:SeverityType"
      minOccurs="0"/>
  </xs:sequence>
</xs:complexType>

<!-- Common enumeration to distinguish errors from warnings -->
<xs:simpleType name="SeverityType">
  <xs:restriction base="xs:token">
    <!-- Critical: requestor must terminate execution -->
    <xs:enumeration value="Critical"/>
    <!-- Warning: processing can continue -->
    <xs:enumeration value="Warning"/>
  </xs:restriction>
</xs:simpleType>

```

THE PRENTICE HALL SERVICE-ORIENTED COMPUTING SERIES FROM THOMAS ERL

