This article takes an in-depth look at fault handling within a business process, to provide a clearer understanding of business process faults and their handling. We explain how a fault is classified, when it can occur (from business related to system failure), and what you can do with it.

We also discuss what Business Process Execution Language (BPEL) offers for fault handling mechanisms and show how to use constructs such as Catch, Catch All, Reply, Throw and Rethrow in fault handling. Using WebSphere Integration Developer 6.0.2, we show how to define fault handlers to catch and handle all classifications of faults. In addition, we look at some of the IBM extensions to BPEL within WebSphere Process Server that aid fault handling.

Introduction

This article explains fault-handling mechanisms in the context of a microflow business process designed using BPEL. A microflow business process is a short-running process that constitutes a single transaction. In contrast, a macroflow or long-running process could constitute multiple transactions over a prolonged period. The term fault refers to any exceptional condition that can alter the normal processing of a business process. Within the context of a business process, a fault need not result in a process ending condition; instead, a fault should lead to an actionable event. If a fault is not handled, it will lead to unexpected conditions, outcomes or even unpredictable failures of the business process. A well-designed business process should handle faults so that failures lead to predictable outcomes. Within BPEL, fault handlers can catch faults and attach business relevant execution logic to deal with the exceptional situation.

To correctly approach fault handling, the team identifies all exceptional conditions that can occur in a business process. The fault handling logic can be different depending on the nature of the fault. Identifying and classifying the faults helps you to design the fault handlers and place them within the appropriate scope of the business process.

Faults are either application faults or system and standard faults:
• **Application faults:** These faults occur at set points during a business process due to application issues. The fault condition could be the result of business rules or constraint violations. For example, invoking a bank service to transfer funds can result in an insufficient funds fault. The process designer defines application faults in BPEL. An external service provider defines them in the service’s WSDL. In WebSphere Integration Developer, they are displayed as *User-defined* faults as Figure 1 shows:

**Figure 1. User-defined faults**

```
Properties X Problems X Servers X Console

ApplicationFault

Fault Type: Built-in  User-defined
Namespace: Http://example/ExampleFaultService/
Fault Name: ApplicationFault
Variable Name: appFaultVar
Data Type Interface
Interface: raiseFaultService
Operation: raiseFault
Direction: Input  Output  Fault
Fault: ApplicationFault
```

• **System faults and BPEL standard faults:** These exceptions are defined by the BPEL runtime and not by the process designer or external service provider. These faults occur due to system-related issues not directly related to data content (for example, service unavailable or network failure). These faults are unexpected and can occur any time within the process execution. These failures could be short-term (transient) failures (brief network partition) and long-term failures (disk failure). To resolve short-term failures, failure-handling logic such as retries sometimes work. Long-term failures might require human intervention to correct the system failure, followed by a recovery mechanism.

As a subset of system failures, the WS-BPEL specification defines built-in standard fault types for commonly encountered problems. For example, the `uninitializedVariable` fault indicates an attempt to access the value of an uninitialized part in a message variable. WebSphere Integration Developer provides the standard fault types as built-in faults, as Figure 2 shows:
Figure 2. Built-in faults

Typical fault handling flow

When a fault occurs in a process, the current operational flow moves to the fault handler within the immediate scope. If the current fault handler does not have the appropriate Catch element defined to trap and handle this fault, the Business Process Container checks from inner scope to outer scope until it finds an appropriate fault handler that can trap this fault. You can visualize the various fault handlers as a chain of fault handlers going from inner to outer scopes; we refer to this as a fault handler chain. This chain, in various scopes, is responsible for trapping the appropriate fault and taking an appropriate action.

The Catch mechanism handles a specific application or standard fault. You can define multiple Catch elements each corresponding to a different exception type. In cases when the type of a fault is unknown, you can use the Catch All element, but you may only define one of these elements per fault handler. It is a best practice to use a Catch All element in the Global Process's fault handler.

Attaching fault handlers

You can also specify fault handlers for invoke activities, scopes, and at the global process level using the WebSphere Integration developer process editor. In this section, we show fault handlers that are attached to the various scopes.

Fault handler attached to an invoke activity

This handler (Figure 3) catches faults specific to an invoke activity. Usually, you define these application faults on the invoking WSDL interface.
If the exception resulting from the invoke activity is not caught by its fault handler, it uses a handler further up the handler chain.

**Fault handler attached to a scope**

This fault handler (Figure 4) catches faults resulting from activities within a scope. These could be uncaught faults from an invoke activity or faults from any other task, such as an exception within a Java snippet.

**Global process level handler**

If there is no handler defined for a fault occurring within a given scope, the fault automatically propagates to the immediately enclosing scope. Faults progressively propagate from the inner scopes to the enclosing outer scopes until they reach a specific fault handler or reach the application-defined, global process fault handler, as Figure 5 shows. If you have not defined a global process level handler, the process will fail to complete gracefully.
**Figure 5. Global process level handler**

If a fault is not handled within a scope, it propagates to the next hierarchical scope and could eventually bubble up to the main process's fault handler, if one is included.

### Matching a fault to a Catch activity

When a fault occurs, the Business Process Container (BPC) needs to match the fault with a fault handler. As specified in the BPEL spec, the BPC uses following rules to select the catch activity that processes a fault:

1. If the fault has no associated fault data, the BPC uses a Catch activity with the matching fault name. Otherwise, it uses the default CatchAll element.
2. If the fault does have associated fault data, the BPC uses a Catch activity with matching fault name and variable values. If there is no fault name specified, then it uses a Catch with a matching fault type. Otherwise, it uses the default CatchAll element.

If you do not define a CatchAll handler anywhere in the fault handler chain, the fault reaches the global business process level and the process ends in a failed state, which is obviously undesirable.

One objective of good process design and implementation is to handle all exceptional conditions. It is an admittedly lofty goal to anticipate all exceptional scenarios at the inception of process development. Remember to include time in the development cycle for additional fault-handler development as the process moves through testing and even after production deployment.

When designing fault handlers, consider the following options:

- Catch a fault and try to correct the problem, allowing the business process to continue to normal completion.
- Catch a fault and find that it is not resolvable at this scope. Now, you have additional options:
  - Throw a new fault.
  - Re-throw the original fault to allow another scope to handle it.
  - Reply with a fault to the process initiator.
  - Invoke a human task to correct the issue.
  - If the fault handler cannot resolve the issue, you might need to rollback and compensate.
BPEL activities to propagate a fault

BPEL allows fault propagation using throw, rethrow and reply within a fault handler. Next we’ll look at each of these ways to propagate a fault:

Throw

A throw activity indicates a problem that a business process flow cannot handle. You use it to throw an exception corresponding to an internal error condition. You can use a throw activity within the flow of a business process or within a fault handler, to allow an outer fault handler to handle the fault. A throw activity can throw one of the standard BPEL faults or a custom fault. The throw activity is required to provide a name for the fault and can optionally include a variable with data providing further information about the fault, as Figure 6 shows:

Figure 6. A throw activity

Rethrow

You use a rethrow activity (Figure 7) when the current fault handler cannot handle the fault and wants to propagate it to an outer-scoped fault handler. In the absence of a rethrow activity, a fault propagated to a higher level using a throw activity would be a new fault. When a rethrow activity is invoked, the fault is the same instance. The rethrow activity is available only within a fault handler because only an existing fault can be rethrown.

Figure 7. A rethrow activity

Reply with a fault

This construct allows the propagation of the fault to the client that initiated the process. A reply with fault activity (Figure 8) can only return a fault defined on the interface the process is implementing. This is useful when the business process cannot properly respond to the caught fault, and the process initiator may be better equipped to respond; for example if the client passes an account
number that is not found by the business process, the process should reply to the service call with an AccountNotFound Fault.

**Figure 8. Reply with a fault construct**

![Fault Construct](image)

**Compensation**

Most microflows are atomic transaction, and normal rollback logic will handle most cases. However, there are cases where a fault handler might need to compensate work that has already been accomplished. All or part of this work would need to be reversed. This topic is the subject of another article: **Using compensation in business processes with Business Process Choreographer**.

**Continue on Error setting**

This setting is an IBM extension available for fault handling for a long running process. In the Server tab area of the Properties page of BPEL invoke activity, snippet or human task, you will find a check box called **Continue On Error**. When no fault handler exists for an activity, snippet, or human task and continue or error is set to:

- True (default setting) - the business process creates and throws a fault.
- False - the business process stops the activity and creates a task (work item) for the administrator of the business process. The administrator has to take action and the activity can be “fixed” and retried, forced complete, or failed.

**Extracting fault details**

There may be situations, especially with unknown faults, where you need more details about a fault (root cause). In some cases, the fault contains no associated fault variable. This is especially true when the Catch All clause handles the fault.

Before WebSphere Process Server version 6.0.2, there was no supported way to get more details on the fault. WebSphere Process Server 6.0.2 introduces the capability to get the fault programmatically. It provides a new Java snippet method, `getCurrentFaultAsException()`. When invoked within a BPEL Java snippet, this method lets you get the current fault in the form of an exception object of the type `com.ibm.bpe.api.BpelException`. The BpelException object provides several operations to get more details on the fault, such as the fault name. The BpelException wraps the actual exception instance. Thus, you can access the fault message as well as the root exception, as this listing shows:

```java
com.ibm.bpe.api.BpelException bpelexception = getCurrentFaultAsException();
System.out.println("Fault Name" + bpelexception.getFaultName());
bpelexception.printStackTrace(System.out);
Throwable rootCause = bpelexception.getRootCause();
```
Examples of handling faults within BPEL

In this section, we look at the following fault scenarios:

- Faults thrown within a BPEL process.
- Faults thrown when invoking an service.
- Faults thrown within a Java Snippet, either raising Standard BPEL faults or Java exceptions.

We include a sample project you can download and instructions to execute the scenarios below. Please use the provided values for faultName and raiseFault documented in the following sections with this sample project.

Application faults raised within a BPEL process

If a BPEL process reaches an error condition, you can throw a fault by creating an application fault. We will illustrate how to create, handle and respond to application and system faults at different scope levels. We'll keep the process interface simple; it takes an input value that indicates which fault to throw.

The process interface looks like Figure 9:

Figure 9. Process interface

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>faultName</td>
<td>string</td>
</tr>
<tr>
<td>raiseFault</td>
<td>boolean</td>
</tr>
<tr>
<td>output</td>
<td>string</td>
</tr>
</tbody>
</table>
```

Figure 10 shows the process:

Figure 10. Process model
If the input to the process is `faultName = invalid` and `raiseFault = true`, then it executes the `raiseInvalidException` case. We throw an application exception using the throw activity. Adding the throw to the Process editor creates a throw activity.

To create a throw activity:

1. In the bottom section of the Process editor, select the **Properties** tab. Click on the **Details** section, as Figure 11 shows:

   **Figure 11. ThrowInvalidInputException properties**

2. Select **User-defined** for the **Fault Type**. The **Namespace** drop-down list lets you select a namespace for the fault. We have selected `http://FaultHandlingSampleModule`, which is the name of the SCA module.
3. You can choose any name for the fault name; we have defined it as `InvalidInputException`.
4. Next we'll define a fault handler on the scope to catch this fault. Select the Scope by clicking within its area. Click the **Add Fault Handler** icon as shown in Figure 4 above.
5. Click on the catch that appears within the fault handler, and in the button section of the Process Editor select the **Properties** tab.
6. Choose **User-defined** for the **Fault Type**.
7. When you choose the namespace, the `InvalidInputException` fault name displays.
8. When we created the fault, we did not specify a fault variable, so the input box for the fault variable should be empty. Your Property editor should now look like Figure 12.

   **Figure 12. Catch properties**

Figure 13 shows the BPEL process with the fault handler:
Next run a test with inputs to trigger the throw (faultName = invalid and raiseFault = true). You'll see that the fault is caught and handled within the fault handler defined for this application exception.

In the Console view of WebSphere Integration Developer, you should see the following output:

```
SystemOut     0 InvalidInputException Caught
```

In this example, we raised a fault using a throw activity; you can also raise an application fault through a Java snippet, as you'll see in the next section.

**Raising application faults from a Java snippet**

Instead of defining a throw with the appropriate properties to raise an application fault, you can define the fault name with a Java snippet. The methods to raise fault programmatically in a Java snippet are:

```
raiseFault( QName faultName )
raiseFault( QName faultName, String variableName )
```

You define the `QName` with the fully qualified namespace. This listing shows the Java snippet that raises an application exception in the BPEL process in this sample:

```java
QName faultName =
    new QName("http://FaultHandlingSampleModule", "JavaSnippetException");
raiseFault(faultName);
```

Figure 14 shows how this same snippet looks in the Visual Snippet Editor.
You define the catch clause in the fault handler as we did in the previous case. The one difference is that the drop-down list for the Fault Name in the properties for the catch clause does not list JavaSnippetException (the fault is not defined in the tooling so the properties menu cannot automatically discover the name), so you need to type in the fault name.

We extended the sample process to demonstrate throwing a fault from a Java snippet, as Figure 15 shows. (The inputs to the process to execute this branch are faultName = Java and raiseFault = true).

Figure 15. Process throwing a fault from a Java snippet

The output in the console shows this message:

SystemOut O JavaSnippetException Caught

Handling faults that are not in your immediate scope

In this case, we are not going to create a fault handler at the scope level for the application fault, but we will create one at the global process level, as in Figure 5. We will use an application fault called InnerScopeException and a Catch at the process level. To do this we will add an additional case to the first scope activity.

When you catch a fault at a global process level, you need to take some corrective measures, to terminate, rethrow or reply. In this example, we reply. If we just catch the fault and log it, the exception message displays that the process ended prior to a reply being sent. To allow the process to complete gracefully, we add a Reply activity to the catch block. Your process should look like Figure 16. Please note that there is no catch for this exception at the scope activity, so it is handled at the global process level.
Figure 16. Global process level fault handling

Figure 17 shows the details of the reply from the global catch. This Catch block catches the InnerScopeException, logs it, sets the output1 variable to a value and sends a Reply. Note that the process will complete gracefully because we have handled the exceptional condition. You should also add a CatchAll to this catch block and handle all other faults in a similar way using a Human Task or some other fashion.

Figure 17. Reply from Global Catch details

Figure 18 shows the throw fault definition:
Figure 18. Throw fault definition

![Figure 18. Throw fault definition](image)

Figure 19 shows the catch definition:

**Figure 19. Catch definition**

![Figure 19. Catch definition](image)

If the input to the process is $\text{faultName} = \text{innerscope}$ and the $\text{raiseFault} = \text{true}$, then it executes the $\text{raiseInnerScopeException}$ case. In this branch, we throw an application exception using the throw activity.

The Console window shows the following message:

```
SystemOut  0 InnerScopeException was caught by global process catch
BpelEngine  I  CWWBE0061E: A fault 'InnerScopeException'
was raised by activity 'ThrowInnerScopeException'.
```

You might notice that there is an extra line of trace in the Console. This occurs when there is no immediate scope fault handler and the BPC needs to use a fault handler further up the handler chain.

**Catching and handling faults for BPEL with an invoke activity to an external service**

When a BPEL process invokes an external service, it would need to handle application faults defined on the invoked operation for the service interface, and unknown runtime faults. The fault on the operations on the WSDL interface translates to an application fault within the process.

In this section we define a simple external service with an interface containing one operation that can raise an application fault, as Figure 20 shows:
Figure 20. External service fault operation

When the process invokes this service, it defines a fault handler on the invoke activity to catch and handle the application fault, as well as a catch activity as a built in fault type to trap runtime faults.

To catch the application fault, set the catch activity properties in the fault handler as shown in Figure 21. Note that the fault is defined on the Interface. The **Interface** option in the Catch properties lets you choose the interface **Operation** and the associated **Fault**.

Figure 21. Catch activity properties

Figure 22 shows how the runtime fault is caught and handled:
To test for application faults from an external service, test the FaultHandlingSampleModule with faultName = external and raiseFault = true.

The Console displays the following message:

```
SystemOut 0 This has found the Otherwise case. we will log and let fall through to the invoke
SystemOut 0 Application Fault from external service caught
```

You can simulate the built-in runtimeFailure fault by stopping the ExternalFaultRaiseModuleApp enterprise application from the administration console, or by removing the project from the server via the Servers view.

Now redo the test of FaultHandlingSampleModule with faultName= external and raiseFault = true.

The Console log output displays this message:

```
ExternalFaultRaiseModuleApp SystemOut 0 This has found the Otherwise case. we will log and let fall through to the invoke
... faultString: org.xml.sax.SAXParseException: An invalid XML character (Unicode: 0x45) was found in the prolog of the document. Message being parsed: Error 404: No target servlet configured for uri: /ExternalFaultRaiseModuleWeb/sca/ExternalRaiseFaultInterfaceExport1
... SystemOut 0 Runtime Fault from external service caught
```

Conclusion

In this article, we looked at BPEL fault handling techniques, the artifacts available to raise faults, and ways to define fault handlers. Fault handling is an important component of good process
design. The various constructs within BPEL to trap faults let integration developers design and handle the faults so that the process behaves in a predictable manner even when there are errors.

Acknowledgements

The authors would like to thank Peter Van Sickel, Ed Grossman and Wangming Ye for reviewing this article.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample module</td>
<td>faulthandlingsamplemodule_pi.zip (HTTP</td>
<td>FTP)</td>
</tr>
</tbody>
</table>
Related topics

- Instructions for running the sample application
- WebSphere Integration Developer product documentation
- IBM Business Process Manager Developer Center
- WebSphere Integration Developer product information
- Meet the experts: WebSphere Integration Developer

© Copyright IBM Corporation 2007
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)