Simplifying WCF: Using Exceptions as Faults

Posted by Oleg Sych July 26, 2008

This article describes an overview of error handling in WCF, discusses its advantages and drawbacks and shows how to extend WCF to marshal .NET exceptions as SOAP faults. Ready-to-use source code and examples are available for download.

Overview

When using WCF to create distributed applications, .NET developers have to deal with the issue of passing error information from service to its consumer. In modern applications and frameworks, error information is usually communicated with the help of exceptions. Unfortunately, WCF was designed for service interoperability and uses SOAP faults instead of exceptions. In order to be interoperable, fault messages are much simpler than exceptions. They reduce error information to Code, which provides machine-readable error information and Reason, which provides human-readable error information. If additional information about the error is necessary, the service must define a fault data contract that can be consumed by the client.

Let's review examples of typical error handling code used by WCF service and client applications. Consider the following service contract.

```
[ServiceContract]
public interface IService
{
    [OperationContract]
    void Operation();
}
```

Simple Error Handling

To pass information about data access errors to the client, the service code shown below catchesDbException and throws a FaultException with a specific FaultCode. WCF catches *FaultException* and generates a fault Message that will be sent to the client instead of a normal response a service operation would return.

```
public class Service : IService
{
    public void Operation()
    {
        try
        {
            // Access a database …
        }
        catch (DbException e)
        {
            throw new FaultException(
                new FaultReason(e.Message),
                new FaultCode( “Data Access Error” ));
        }
    }
}
```

Below is an example of client code handling the data access error.

```
public class Client
{
  public void InvokeOperation()
  {
    IService service = // Obtain service proxy…
    try
    {
      service.Operation();
    }
    catch (FaultException e)
    {
      if (e.Code.Name ==  “Data Access Error” )
      {
        Console.WriteLine( “Handling data access exception {0}” , e.Reason);
      }
    }
  }
}
```

FaultCode is a machine-readable error description, similar to error codes used in Win32 and COM APIs. The client code above uses an if statement to examine the *FaultCode* and determine if it is caused by a data access error. Although this error-handling approach is simple, as the number of possible error conditions that need to be handled on the client side increases, developer has to add a new else-if clause for each specific *FaultCode*. In other words, this approach encourages developers to create procedural, difficult-to-maintain error handling code we used to write in good old days, before exceptions.

Advanced Error Handling

WCF also allows developers to define custom fault contracts, which helps to deal with a large number of error conditions and more complex error conditions in a more object-oriented way.

Below is an example of a custom fault contract that provides more details about data access errors. For illustration purposes, this fault contract is simplistic. A more realistic data access fault contract could provide machine-readable information about a foreign key violation that would allow a client application to handle Customer deletion failure by displaying a list of Orders that need to be deleted first.

```
[DataContract]
public class DataAccessFault
{
    [DataMember]
    public string AdditionalDetails { get; set; }
}
```

Having the fault contract defined, you need to modify service contract to indicate that a particular method may return it. WCF provides FaultContractAttribute for this purpose.

```
[ServiceContract]
public interface IService
{
    [OperationContract]
    [FaultContract(typeof(DataAccessFault))]
    void Operation();
}
```

Here is the updated version of the service code. Note that it now throws a generic version of FaultExceptionthat takes fault contract as a type parameter.

```
public class Service : IService
{
    public void Operation()
    {
        try
        {
            // Access database …
        }
        catch (DbException e)
        {
            DataAccessFault fault = new DataAccessFault();
            fault.AdditionalDetails = e.Message;
            throw new FaultException<DataAccessFault>(fault);
        }
    }
}
```

On the client side, the error-handling code can now look like this.

```
public class Client
{
    public void InvokeOperation()
    {
        IService service = // Obtain service proxy…
        try
        {
            service.Operation();
        }
        catch (FaultException<DataAccessFault> e)
        {
            Console.WriteLine(
                "Handling data access error {0}",
                e.Detail.AdditionalDetails);
        }
    }
}
```

The big improvement here is that code now uses strongly-typed try/catch statement instead of weakly-typed if-else-if statement to detect a particular error condition. Additional information about the error can be passed from service to client using properties of the fault class. Unfortunately this improvement came at a cost of having to maintain yet another class that does nothing by transfer data from service to client. In order to handle error conditions in a type-safe, object-oriented manner, developer has to define a separate fault contract class for each error condition that needs to be handled on the client side.

Using Exceptions as Faults

In most cases, fault contracts and fault codes directly correspond to exceptions already defined in the framework and application code. During development of distributed applications, where service and client are parts of a single closed system, having to convert exceptions to faults and back to exceptions is a pure overhead, plumbing that adds no value to the end result. In a distributed application, it would be ideal to have error information travel as exceptions from the service to the client, without forcing developer to do extra coding.

It is possible to use exceptions themselves as fault contracts. For example, here is what service code could look like.

```
public class Service : IService
{
    public void Operation4()
    {
        try
        {
            // …
        }
        catch (DbException e)
        {
            throw new FaultException<DbException>(e);
        }
    }
}
```

And here is what client code could look like.

```
public class Client
{
  public void InvokeOperation()
  {
    IService service = null; // …
    try
    {
      service.Operation();
    }
    catch (FaultException<DbException> e)
    {
      Console.WriteLine( "Handling data access error {0}" , e.Detail.Message);
    }
  }
}
```

This approach relies on support WCF provides for runtime serialization, the same mechanism used by .NET remoting. All exception classes must be serializable by design. All exceptions defined by the .NET framework class libraries *are* serializable. Custom application exceptions that define additional properties can support serialization by implementing GetObjectData method and serialization constructor.

Unfortunately, WCF supports runtime serialization only for primitive types to ensure interoperability. This creates a problem with exception serialization, in particular with Exception.Data property, which is exposed as IDictionary and implemented as an internal dictionary class. WCF's DataContractSerializer cannot serialize complex types, like this, because resulting XML infoset would not be interoperable.

Ideal Scenario

There is one more problem with the last example above. It forces developer to use FaultException class on both service and client side. This hard-wires WCF to every non-trivial piece of code in a distributed application. If, for any reason, you later decide to deploy the application on a single tier, you will not be able call the service directly from the client in the same AppDomain and will have to either live with messaging overhead or go through the application code and remove *FaultException* from error handling code.

Below is an ideal implementation of the error-handling in the service. Note that there is no special, WCF-specific error handling code. Service simply accesses the database and allows *DbException* to propagate to the caller.

```
public class Service : IService
{
    public void Operation()
    {
        // Access database …
    }
}
```

Here is an ideal implementation of the error-handling in the client code. Here we also have no WCF-specific error handling code. Client simply catches and handles *DbException* thrown by the service code.

```
public class Client
{
    public void InvokeOperation()
    {
        IService service = // Obtain service proxy …
        try
        {
            service.Operation();
        }
        catch (DbException e)
        {
            Console.WriteLine( "Handling data access exception {0}" , e.Message);
        }
    }
}
```

In fact, there is nothing "distributed" left in this code, which is a good thing. One would hope that, this code follows Martin Fowler's first law of distributed objects and simply uses good design practices to separate application into layers, with *Client* calling *Service* through an explicitly defined interface without incurring the overhead of WCF at all. Later when developer determines a good reason for separating *Client* and*Service* into separate physical tiers and using WCF for communication, it would be ideal if this code simply continued to work without developer having to go through and add logic to wrap and unwrap errors usingFaultException.

Solution

This section describes a solution that uses WCF extensibility mechanisms to automatically convert exceptions to faults on the service side and faults to exceptions on the client side, making possible the ideal scenario described above.

NetDataContractSerializer

Unlike DataContractSerializer used by WCF by default to produce interoperable XML messages,NetDataContractSerializer embeds type information into XML it generates. This allows it serialize any .NET type, but messages it generates are not interoperable, as they require a .NET application to deserialize. This behavior is similar to that of SoapFormatter used by Remoting and can be appropriate and beneficial for closed distributed application, where interoperability is not required. Here is an example of an exception serialized by *NetDataContractSerializer*.

```
<ApplicationException z:Id= “1”  z:Type= “System.ApplicationException”  z:Assembly= “0”
  xmlns= “http://schemas.datacontract.org/2004/07/System”
  xmlns:i= “http://www.w3.org/2001/XMLSchema-instance”
  xmlns:x= “http://www.w3.org/2001/XMLSchema”
  xmlns:z= “http://schemas.microsoft.com/2003/10/Serialization/ “>
  <ClassName z:Id= “2”  z:Type= “System.String”  z:Assembly= “0”  xmlns= “” >System.ApplicationExc
  <Message z:Id= “3”  z:Type= “System.String”  z:Assembly= “0”  xmlns= “” >Exception raised on the
  <Data z:Id= “4”  z:Type= “System.Collections.ListDictionaryInternal”  z:Assembly= “0”  xmlns= “
    <count xmlns= “http://schemas.microsoft.com/2003/10/Serialization/Arrays “>1</count>
    <head z:Id= “5”  xmlns= “http://schemas.microsoft.com/2003/10/Serialization/Arrays”
      xmlns:b= “http://schemas.datacontract.org/2004/07/System.Collections “>
      <b:key z:Id= “6”  z:Type= “System.String”  z:Assembly= “0 “>exception</b:key>
      <b:next i:nil= “true “></b:next>
      <b:value z:Id= “7”  z:Type= “System.String”  z:Assembly= “0 “>data</b:value>
    </head>
    <version xmlns= “http://schemas.microsoft.com/2003/10/Serialization/Arrays “>1</version>
  </Data>
  <InnerException i:nil= “true”  xmlns= “” ></InnerException>
  <HelpURL i:nil= “true”  xmlns= “” ></HelpURL>
  <StackTraceString i:nil= “true”  xmlns= “” ></StackTraceString>
  <RemoteStackTraceString i:nil= “true”  xmlns= “” ></RemoteStackTraceString>
  <RemoteStackIndex z:Id= “9”  z:Type= “System.Int32”  z:Assembly= “0”  xmlns= “” >0</RemoteStack
  <ExceptionMethod i:nil= “true”  xmlns= “” ></ExceptionMethod>
  <HResult z:Id= “11”  z:Type= “System.Int32”  z:Assembly= “0”  xmlns= “” >-2146232832</HResult>
  <Source z:Id= “12”  z:Type= “System.String”  z:Assembly= “0”  xmlns= “” >Server</Source>
</ApplicationException>
```

ExceptionMarshallingErrorHandler

On the service side, WCF provides IErrorHandler interface. An object implementing this interface can be added to the list of ChannelDispatcher.ErrorHandlers to process unhandled exceptions that reached WCF runtime.

ExceptionMarshallingErrorHandler implements *IErrorHandler* to serialize unhandled exceptions thrown by service code as fault messages. When an exception bubbles up to WCF runtime, it calls ProvideFaultmethod of the error handler to process the exception. As you can see below, code in *ProvideFault* method uses NetDataContractSerializer to serialize the exception and generate a fault message that will be sent to the client. WCF then calls HandleError method to determine if it needs to perform standard error handling for this exception.

```
using System;
using System.Collections.ObjectModel;
using System.Runtime.Serialization;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Description;
using System.ServiceModel.Dispatcher;

namespace WcfExtensions
{
    public class ExceptionMarshallingErrorHandler : IErrorHandler
    {
        public bool HandleError(Exception error)
        {
            if (error is FaultException)
                return false; // Let WCF do normal processing
            else
                return true; // Fault message is already generated
        }

        public void ProvideFault(
            Exception error, MessageVersion version, ref Message fault)
        {
            if (error is FaultException)
                // Let WCF do normal processing
            else
            {
                // Generate fault message manually
                MessageFault messageFault = MessageFault.CreateFault(
                    new FaultCode( “Sender” ), new FaultReason(error.Message),
                    error, new NetDataContractSerializer());
                fault = Message.CreateMessage(version, messageFault, null);
            }
        }
    }
}
```

ExceptionMarshallingMessageInspector

On the client side, WCF provides IClientMessageInspector interface. An object implementing this interface can be added to the list of ClientRuntime.MessageInspectors to process messages received from the service before they are processed by the rest of the WCF runtime.

ExceptionMarshallingMessageInspector implements *IClientMessageInspector* interface to convert a fault message back to the original exception and re-throw it. When reply message is received from the service,ClientRuntime calls the AfterReceiveReply method of the inspector. Code in this method checks if the message contains a fault and tries to deserialize exception from it. If exception was deserialized successfully, the *AfterReceiveReply* re-throws it, otherwise it allows WCF to finish normal processing.*ReadFaultDetail* method uses *NetDataContractSerializer* to deserialize the exception. Due to its length, body of this method is not included here. It is available for download in the attached source code.

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Description;
using System.ServiceModel.Dispatcher;
using System.Xml;

namespace WcfExtensions
{
  public class ExceptionMarshallingMessageInspector : IClientMessageInspector
  {
    public void AfterReceiveReply(ref Message reply, object correlationState)
    {
      if (reply.IsFault)
      {
        // Create a copy of the original reply to allow default WCF processing
        MessageBuffer buffer = reply.CreateBufferedCopy(Int32.MaxValue);
        Message copy = buffer.CreateMessage();  // Create a copy to work with
        reply = buffer.CreateMessage();         // Restore the original message

        object faultDetail = ReadFaultDetail(copy);
        Exception exception = faultDetail as Exception;
        if (exception != null)
          throw exception;
      }
    }

    public object BeforeSendRequest(ref Message request, IClientChannel channel)
    {
      return null;
    }
  }
}
```

Note that *ExceptionMarshallingMessageInspector* may not be able to deserialize service exception if its type is not available on the client side. In this case, it will allow WCF to perform standard fault handling, which will throw FaultException.

ExceptionMarshallingBehavior

To configure communication channels with the custom *IErrorHandler* and *IClientMessageInspector* objects*ExceptionMarshallingBehavior* implements IServiceBehavior, IEndpointBehavior and IContractBehaviorinterfaces provided by WCF. *IServiceBehavior, IEndpointBehavior* and *IContractBehavior* are used to install*ExceptionMarshallingErrorHandler* on service side. *IEndpointBehavior* and *IContractBehavior* are used to install *ExceptionMarshallingMessageInspector* on the client side. Strictly speaking, it is not required to implement all three behavior interfaces. Implementing either *IEndpointBehavior* or *IContractBehavior* would be sufficient. *ExceptionMarshallingBehavior* implements them all to give developer maximum configuration flexibility. See the *Usage* section below for details.

Extract from the source code of *ExceptionMarshallingBehavior* is included below. Empty methods and some code were removed to make it easier to read. Complete implementation of this class is available for download with attached source code.

```
using System;
using System.Collections.ObjectModel;
using System.Diagnostics;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Description;
using System.ServiceModel.Dispatcher;

namespace WcfExtensions
{
  public class ExceptionMarshallingBehavior: Attribute,
    IServiceBehavior, IEndpointBehavior, IContractBehavior
  {
    void IContractBehavior.ApplyClientBehavior(ContractDescription contract,
      ServiceEndpoint endpoint, ClientRuntime runtime)
    {
      this.ApplyClientBehavior(runtime);
    }

    void IContractBehavior.ApplyDispatchBehavior(ContractDescription contract,
      ServiceEndpoint endpoint, DispatchRuntime runtime)
    {
      this.ApplyDispatchBehavior(runtime.ChannelDispatcher);
    }

    void IEndpointBehavior.ApplyClientBehavior(
      ServiceEndpoint endpoint, ClientRuntime runtime)
    {
      this.ApplyClientBehavior(runtime);
    }

    void IEndpointBehavior.ApplyDispatchBehavior(
      ServiceEndpoint endpoint, EndpointDispatcher dispatcher)
    {
      this.ApplyDispatchBehavior(dispatcher.ChannelDispatcher);
    }

    void IServiceBehavior.ApplyDispatchBehavior(
      ServiceDescription service, ServiceHostBase host)
    {
      foreach (ChannelDispatcher dispatcher in host.ChannelDispatchers)
        this.ApplyDispatchBehavior(dispatcher);
    }

    private void ApplyClientBehavior(ClientRuntime runtime)
    {
      // Don’ t add a message inspector if it already exists
      foreach (IClientMessageInspector inspector in runtime.MessageInspectors)
        if (inspector is ExceptionMarshallingMessageInspector)
          return;
      runtime.MessageInspectors.Add(new ExceptionMarshallingMessageInspector());
    }

    private void ApplyDispatchBehavior(ChannelDispatcher dispatcher)
    {
      // Don’ t add an error handler if it already exists
      foreach (IErrorHandler errorHandler in dispatcher.ErrorHandlers)
        if (errorHandler is ExceptionMarshallingErrorHandler)
          return;
      dispatcher.ErrorHandlers.Add(new ExceptionMarshallingErrorHandler());
    }
  }
}
```

ExceptionMarshallingElement

WCF includes a class called BehaviorExtensionElement, which serves as a base for classes that encapsulate configuration elements you can use in app.config files to configure service and endpoint behaviors. *ExceptionMarshallingElement* inherits from *BehaviorExtensionElement* to provide support for*ExceptionMarshallingBehavior* in application configuration files.

```
using System;
using System.ServiceModel.Configuration;

namespace WcfExtensions
{
    public class ExceptionMarshallingElement : BehaviorExtensionElement
    {
        public override Type BehaviorType
        {
            get { return typeof(ExceptionMarshallingBehavior); }
        }

        protected override object CreateBehavior()
        {
            return new ExceptionMarshallingBehavior();
        }
    }
}
```

Usage

1. Download and compile exception marshalling extensions

Attached source code includes WcfExtensions project that contains source code of the exception marshalling classes described in this article. You can either compile it as a standalone DLL or merge the source code in one of your projects. Either way, the assembly that contains this code needs to be deployed with both client and service code.

2. Configure your application to use exception marshalling extensions

There are three ways to configure exception marshalling behavior. They produce the same end result. And if you want to find out what maintenance developers *really* think about you, you can mix and match different options on client and service side or even use more than one option on each side.

a) In Application Code, Programmatically

Here is how to configure exception marshalling behavior programmatically on the service side.

```
ServiceHost host = new ServiceHost(typeof(Service));
ServiceEndpoint endpoint = host.AddServiceEndpoint(…);
endpoint.Behaviors.Add(new ExceptionMarshallingBehavior());
// or host.Description.Behaviors.Add(new ExceptionMarshallingBehavior());
// or endpoint.Contract.Behaviors.Add(new ExceptionMarshallingBehavior());
host.Open();
```

Here is how to configure exception marshalling behavior programmatically on the client side.

```
ChannelFactory<IService> factory = new ChannelFactory<IService>(…);
factory.Endpoint.Behaviors.Add(new ExceptionMarshallingBehavior());
// factory.Endpoint.Contract.Behaviors.Add(new ExceptionMarshallingBehavior());
IService channel = factory.CreateChannel();
```

If you are using by svcutil or Add Service Reference in Visual Studio to generate a proxy class for accessing the service, you can access the factory using its ChannelFactory property.

b) In Application Code, Declaratively

Here is how to configure exception marshalling behavior in the service code using attributes.

```
[ExceptionMarshallingBehavior]
public class Service: IService
{
    // service implementation
}
```

If you are using a shared service contract definition on both service and client side, such as when you have the interface defined in a common assembly deployed on both sides, or if you are compiling the same source file that defines it in both service and client assemblies, you can also configure exception marshalling behavior on the service contract.

```
[ServiceContract]
[ExceptionMarshallingBehavior]
public interface IService
{
    // operation contracts
}
```

If both client and service use this interface definition, exception marshalling behavior will be enabled automatically on both sides.

c) In Application Configuration Files

Here is how to configure exception marshalling behavior in the application configuration file on the service side. Note that example below demonstrates both endpoint and service behavior configuration. Choose one particular approach appropriate for your needs. For example, if you choose endpoint behavior configuration, you need to remove <serviceBehaviors/> element and service->behaviorConfiguration attribute from this example.

```
<?xml version=“1.0” encoding=“utf-8” ?>
<configuration>
  <system.serviceModel>
    <behaviors>
      <endpointBehaviors>
        <behavior name=“endpointBehavior“>
          <exceptionMarshalling/>
        </behavior>
      </endpointBehaviors>
      <serviceBehaviors>
        <behavior name=“serviceBehavior“>
          <exceptionMarshalling/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <extensions>
      <behaviorExtensions>
        <add name=“exceptionMarshalling”
          type=“WcfExtensions.ExceptionMarshallingElement, WcfExtensions,
                Version=1.0.0.0, Culture=neutral, PublicKeyToken=null“/>
      </behaviorExtensions>
    </extensions>
    <services>
      <service name=“Server.Service” behaviorConfiguration=“serviceBehavior“>
        <endpoint address=“net.tcp://localhost:9001/Service”
          binding=“netTcpBinding” contract=“Common.IService”
          behaviorConfiguration=“endpointBehavior“/>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

Here is how to configure exception marshalling behavior in the application configuration file on the client side.

```
<?xml version=“1.0” encoding=“utf-8” ?>
<configuration>
  <system.serviceModel>
    <behaviors>
      <endpointBehaviors>
        <behavior name=“endpointBehavior“>
          <exceptionMarshalling/>
        </behavior>
      </endpointBehaviors>
    </behaviors>
    <client>
      <endpoint name=“MyEndpoint” address=“net.tcp://localhost:9001/Service”
        binding=“netTcpBinding” contract=“Common.IService”
        behaviorConfiguration=“endpointBehavior“/>
    </client>
    <extensions>
      <behaviorExtensions>
        <add name=“exceptionMarshalling”
          type=“WcfExtensions.ExceptionMarshallingElement, WcfExtensions,
                Version=1.0.0.0, Culture=neutral, PublicKeyToken=null“/>
      </behaviorExtensions>
    </extensions>
  </system.serviceModel>
</configuration>
```

Note that on the client side, only endpoint behavior can be specified in the configuration file. Contract behaviors cannot be specified in configuration files on either side. Also note that we <behaviorExtensions/> elements contain fully-qualified type name of the *ExceptionMarshallingElement* class, which includes its namespace and assembly name. You will need to update this information if you compile these classes in one of your own assemblies.

3. Use Exceptions to Pass Error Information from Server to Client

On the service side, use code like this.

```
public class Service : IService
{
    public void Operation()
    {
        // Execute code that can throw exceptions…
    }
}
```

On the client side, use code like this if you need to handle a particular error condition.

```
public class Client
{
    public void DoStuff()
    {
        try
        {
            // Execute code that can access services
        }
        catch (DbException e)
        {
            Console.WriteLine(“Handling data access exception {0}”, e.Message);
        }
    }
}
```

Consider using an unhandled exception handler, such as Application.ThreadException orAppDomain.UnhandledException check for and to log detailed information about FaultException. An unhandled *FaultException* may occur when an exception could not be deserialized on the client side because its type is only available on the service side. If you find that you have such an exception, consider handling it on the service side or making its type available on the client side if it needs to be handled there.

Sample Application

The attached solution includes two projects - Client and Server, which you can use to test exception marshalling. You can also open messages.svclog file produced by Server in its bin\Debug using Service Trace Viewer to see how exceptions are serialized in fault messages.

Download

Source code of exception marshalling WCF extensions and code examples are available for downloadhere. The code was compiled using Visual Studio 2008 and .NET framework 3.5, but should work without changes with Visual Studio 2005 and .NET framework 3.0.