Faults and Exceptions

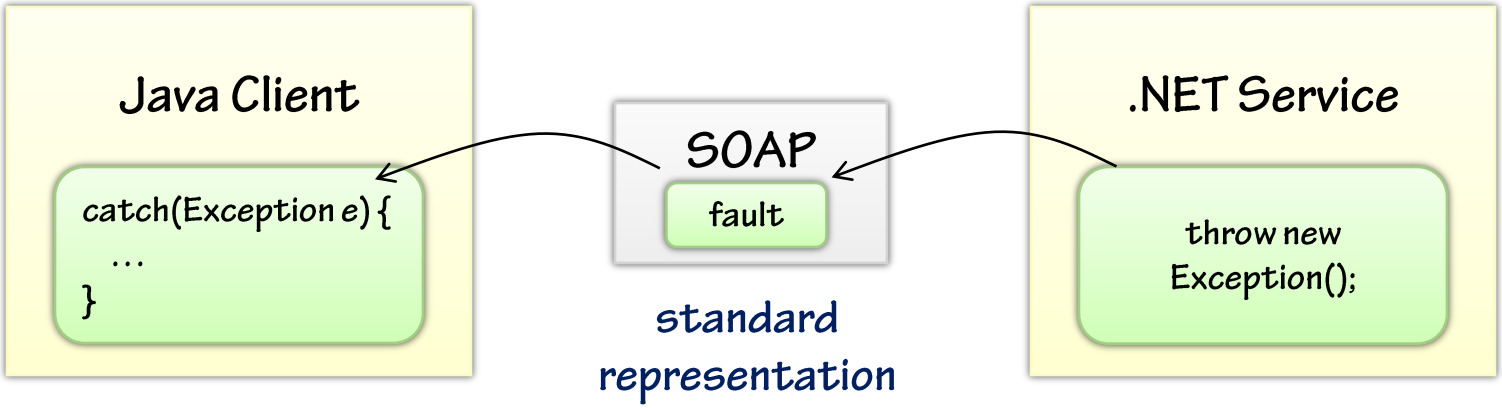
How to expect the unexpected

Outline

- **Exceptions and faults**
  - Moving between the two
- **Undeclared exceptions**
  - Controlling what the client sees
- **FaultException and FaultException<T>**
  - Throwing & catching faults
  - [FaultContract]
- **Global exception handling**
  - IErrorHandler
- **Handling exceptions on the client**

Exceptions and faults

- **Exceptions are technology specific**
  - Hence, they cannot cross the service boundary
- **SOAP defines a standard representation for errors**
  - Referred to as SOAP faults
- **WCF knows how to translate between the two**



SOAP faults

SOAP 1.1

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Body>
    <s:Fault>
      <faultcode>s:Client</faultcode>
      <faultstring xml:lang="en-US">Something bad happened.</faultstring>
      <detail>Anything you want goes here...</detail>
    </s:Fault>
  </s:Body>
</s:Envelope>
```

SOAP 1.2

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
              xmlns:errors='http://example.org/subcodes'>
  <env:Body>
    <env:Fault>
      <env:Code>
        <env:Value>env:Sender</env:Value>
        <env:Subcode><env:Value>errors:MyCode</env:Value></env:Subcode>
      </env:Code>
      <env:Reason>
        <env:Text xml:lang="en-US">Processing error</env:Text>
      </env:Reason>
      <env:Detail>...</env:Detail>
    </env:Fault>
  </env:Body>
</env:Envelope>
```

WCF built-in exception handling

- **What happens when a service operation produces an exception?**
  - The dispatcher catches it and transmits a SOAP fault to the client
  - It doesn't take down the process, but it might fault the channel
- **What information does it send back to the client?**
  - This depends on what type of exception it is
  - If it's a FaultException (or a derivative), all details are transmitted
  - If it's anything else, a generic fault is transmitted by default
  - This is by design to prevent undesired system disclosures

Note: faults cannot be returned from one-way operations

The generic fault

The generic SOAP fault produced by a non-FaultException

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Body>
    <s:Fault>
      <faultcode xmlns:a="..." xmlns="">a:InternalServiceFault</faultcode>
      <faultstring xml:lang="en-US" xmlns="">The server was unable to process the request
      due to an internal error.  For more information about the error, either turn on
      IncludeExceptionDetailInFaults (either from ServiceBehaviorAttribute or from the
      <serviceDebug> configuration behavior) on the server in order to send the exception
      information back to the client, or turn on tracing as per the Microsoft .NET Framework
      3.0 SDK documentation and inspect the server trace logs.</faultstring>
    </s:Fault>
  </s:Body>
</s:Envelope>
```

Including exception details

- **You can instruct WCF to always include exception details in faults**
  - Use either [ServiceBehavior] or <serviceDebug> to enable
  - Useful for debugging purposes, not recommended in production

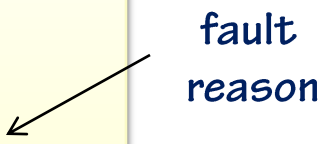
```
<configuration>
    <system.serviceModel>
        <behaviors>
            <behavior name="Default">
               <serviceDebug includeExceptionDetailInFaults="true" />
            </behaviors>
        </behaviors>
    </system.serviceModel>
</configuration>
```

allows exception details to flow

FaultException

- **You can explicitly throw a SOAP fault using the FaultException class**
  - You can specify the fault reason and code when you create it
  - When you do this, the fault details do indeed travel to the client

```
...
public void DoSomething(string input) {
    ...
    throw new FaultException("Something bad happened");
}
```



Catching FaultException

- **The fault will be re-thrown as a FaultException on the client-side**
  - Clients can simply use a traditional try/catch block to handle it
  - The fault code/reason will be available within the instance

catch FaultException to handle SOAP faults

```
try {
    client.SubmitInvoice(invoice);
}
catch (FaultException ex) {
    ...
}
```

Throwing typed faults

- **FaultException produces a simple SOAP fault with a code/reason**
  - Makes it difficult for clients to distinguish between different errors
- **A better technique is to throw typed faults**
  - Define a data contract type to represent the fault information
  - Throw the typed fault using FaultException<T>

Throwing a typed fault

- **Simply throw an instance of FaultException<T>**
  - Where T is the type of fault you wish to throw

throw a specific fault type

```
public class InvoiceService : IInvoiceService
{
    public void SubmitInvoice(Invoice invoice)
    {
        if (!CheckId(invoice))
            throw new FaultException<InvalidId>(
                new InvalidId());
        ...
```

So how do clients find out about typed faults?

Fault contracts

- **You advertise fault types via [FaultContract] your service contracts**
  - Use [FaultContract] to specify each fault type it can throw
  - The detail type must be serializable with DataContractSerializer
  - You cannot use [FaultContract] on one-way operations

list of possible faults →

```
[ServiceContract]
public interface IInvoiceService {
    [FaultContract(typeof(InvalidId))]
    [FaultContract(typeof(InvalidDate))]
    [FaultContract(typeof(MaximumAmountExceeded))]
    [OperationContract]
    void SubmitInvoice(Invoice invoice); ...
}
```

Handling typed faults on the client

- **The [FaultContract] information is included in the service metadata**
  - When clients import the metadata, the fault types are generated
  - And the client's contract is also annotated with [FaultContract]
- **So, clients can catch specific fault types using FaultException<T>**
  - Or clients can simply handle all faults by catching FaultException

handle each fault type →

```
...
try {
    client.SubmitInvoice(invoice);
}
catch (FaultException<InvalidId> ex1) { ... }
catch (FaultException<InvalidDate> ex2) { ... }
catch (FaultException<MaximumAmountExceeded> ex3) { ... }
...
```

Global exception handling/shielding

- **WCF makes it possible to implement a global exception handler**
  - Allows you to shield clients from all undeclared exceptions
  - Allows you to perform a mapping to your declared fault types
  - Allows you to centralize error logging/notification logic
- **You accomplish this by implementing the IErrorHandler interface**
  - Then you write a custom behavior to apply your implementation

IErrorHandler

- **Implement IErrorHandler when you want to customize error handling**
  - Defines two methods: ProvideFault and HandleError
- **ProvideFault is called immediately after an exception is thrown**
  - Allows you to generate a custom fault message
- **HandleError is called on a separate thread after return to client**
  - Allows you to perform more time-consuming error logging techniques

```
public interface IErrorHandler {
    bool HandleError(Exception error);
    void ProvideFault(Exception error,
        MessageVersion version, ref Message fault);
}
```

Applying the IErrorHandler

- **You inject your IErrorHandler implementation using a behavior**
  - Implement a service behavior (derive from IServiceBehavior)
  - Add it to the ChannelDispatcher's ErrorHandlers property
- **You can apply your behavior to the WCF runtime a few different ways**
  - Add it explicitly to the ServiceHost.Description.Behaviors property
  - Make the behavior an attribute an add the attribute to the service
  - Define a behavior extension element and add via configuration

See the modules on "extensibility" for more details on these techniques

Handling exceptions within clients

- Client channels need to be prepared for three main exception types

FaultException or FaultException<T>	• User-defined faults
CommunicationException	• Various runtime communication errors • FaultException derives from this
TimeoutException	• Send timeout limit exceeded • Thrown by underlying transport channel

You typically want to handle them in this order

Handling exceptions in the proper order

invoke operation and call Close for gracefull shutdown

handle service-thrown FaultExceptions

handle other WCF runtime errors

handle timeout errors

```
InvoiceServiceClient client =
    new InvoiceServiceClient("httpEndpoint");
Invoice invoice = ... // create invoice
try {
    client.SubmitInvoice(invoice);
    client.Close();
}
catch (FaultException fe) {
    Console.WriteLine(fe);
    client.Abort();
}
catch (CommunicationException ce) {
    Console.WriteLine(ce);
    client.Abort();
}
catch (TimeoutException te) {
    Console.WriteLine(te);
    client.Abort();
}
...
```

The impact of exceptions on the channel

- **What happens when a client channel receives an exception?**
  - It depends on the type of fault and the type of channel
- **What causes the channel to enter a "faulted" state is not documented**
  - It usually happens when sessionful channels receive InternalServiceFault
  - Once the client channel has faulted, you can only call Abort
  - Check the channel's State property to be sure
- **FaultException-derived types never fault the client channel**
  - FaultException<T> derives from FaultException

Gotcha with "using"

- **You will be tempted to wrap the client channel in a using statement**
  - Doing so automates the call to Close when leaving scope
- **However, Close can also throw communication/timeout exceptions**
  - Close always throws an exception on faulted channels

if service returns a fault, client may enter "faulted" state

Dispose calls Close causing another exception

```
using (InvoiceServiceClient client =
    new InvoiceServiceClient("httpEndpoint"))
{
    client.SubmitInvoice(invoice);
}
```

Proper disposing techniques

- **Either don't use the "using" statement (like our first example)**
  - Or override Dispose in a partial class as follows:

call Abort in "faulted" state

in case Close throws an excpetion

```
public partial class InvoiceServiceClient : IDisposable
{
    void IDisposable.Dispose(){
        try{
            if (this.State == CommunicationState.Faulted)
                this.Abort();
            else
                this.Close();
        }
        catch{
            this.Abort();
        }
    }
}
```

Summary

- Exceptions are technology-specific, faults are technology-neutral
- WCF catches unhandled exceptions and translates them into faults
- A generic fault message is returned for undeclared exceptions
- You explicitly throw faults using FaultException/FaultException<T>
- You advertise typed faults using [FaultContract] on your contracts
- Clients catch FaultException/FaultException<T> objects
- IErrorHandler lets you implement a global error handling scheme
- Certain exceptions can fault the client-side channel

References

- **Specifying and Handling Faults in Contracts and Services (MSDN)**
  - http://msdn.microsoft.com/en-us/library/ms733721.aspx
- **Exception Shielding (Patterns & Practices)**
  - http://msdn.microsoft.com/en-us/library/aa480591.aspx