This page is specific to

Microsoft Visual Studio 2010/.NET Framework 4

**Using Message Contracts**

Typically when building Windows Communication Foundation (WCF) applications, developers pay close attention to the data structures and serialization issues and do not need to concern themselves with the messages in which the data is carried. For these applications, creating data contracts for the parameters or return values is straightforward. (For more information, see [Specifying Data Transfer in Service Contracts](http://msdn.microsoft.com/en-us/library/ms732038.aspx).)

However, sometimes complete control over the structure of a SOAP message is just as important as control over its contents. This is especially true when interoperability is important or to specifically control security issues at the level of the message or message part. In these cases, you can create a *message contract* that enables you to use a type for a parameter or return value that serializes directly into the precise SOAP message that you need.

This topic discusses how to use the various message contract attributes to create a specific message contract for your operation.

**Using Message Contracts in Operations**

WCF supports operations modeled on either the *remote procedure call (RPC) style* or the *messaging style*. In an RPC-style operation, you can use any serializable type, and you have available to you the features that are available to local calls, such as multiple parameters and *ref* and *out* parameters. In this style, the form of serialization chosen controls the structure of the data in the underlying messages, but the WCF runtime creates the messages themselves to support the operation. This enables developers who are not familiar with SOAP and SOAP messages to quickly and easily create and use service applications.

The following code example shows a service operation modeled on the RPC style.

C#

[OperationContract]

public BankingTransactionResponse PostBankingTransaction(BankingTransaction bt);

Normally, a data contract is sufficient to define the schema for the messages. For instance, in the preceding example, it is sufficient for most applications if BankingTransaction and BankingTransactionResponse have data contracts to define the contents of the underlying SOAP messages. For more information about data contracts, see [Using Data Contracts](http://msdn.microsoft.com/en-us/library/ms733127.aspx).

However, occasionally it is necessary to precisely control how a type is mapped to a SOAP message transmitted over the wire. The most common scenario for this is inserting custom SOAP headers. Another common scenario is to define security properties for the message's headers and body, that is, to decide whether these elements are digitally signed and encrypted. Messaging-style operations provide this control.

A messaging-style operation has at most one parameter and one return value where both types are message types; that is, they serialize directly into a specified SOAP message structure. This may be any type marked with the [MessageContractAttribute](http://msdn.microsoft.com/en-us/library/system.servicemodel.messagecontractattribute.aspx) or the [Message](http://msdn.microsoft.com/en-us/library/system.servicemodel.channels.message.aspx) type. The following code example shows an operation similar to the preceding RCP-style, but which uses the messaging style.

For example, if BankingTransaction and BankingTransactionResponse are both types that have message contracts, then the code in the following operations is valid.

C#

[OperationContract]

BankingTransactionResponse Process(BankingTransaction bt);

[OperationContract]

void Store(BankingTransaction bt);

[OperationContract]

BankingTransactionResponse GetResponse();

However, the following code is invalid.

C#

[OperationContract]

bool Validate(BankingTransaction bt);

// Invalid, the return type is not a message contract.

[OperationContract]

void Reconcile(BankingTransaction bt1, BankingTransaction bt2);

// Invalid, there is more than one parameter.

An exception is thrown for any operation that involves a message contract type and that does not follow one of the valid patterns. Of course, operations that do not involve message contract types are not subject to these restrictions.

If a type has both a message contract and a data contract, only its message contract is considered when the type is used in an operation.

**Defining Message Contracts**

To define a message contract for a type (that is, to define the mapping between the type and a SOAP envelope), apply the **MessageContractAttribute** to the type. Then apply the [MessageHeaderAttribute](http://msdn.microsoft.com/en-us/library/system.servicemodel.messageheaderattribute.aspx) to those members of the type you want to make into SOAP headers, and apply the [MessageBodyMemberAttribute](http://msdn.microsoft.com/en-us/library/system.servicemodel.messagebodymemberattribute.aspx) to those members you want to make into parts of the SOAP body of the message.

The following codes provides an example of using a message contract.

C#

[MessageContract]

public class BankingTransaction

{

[MessageHeader] public Operation operation;

[MessageHeader] public DateTime transactionDate;

[MessageBodyMember] private Account sourceAccount;

[MessageBodyMember] private Account targetAccount;

[MessageBodyMember] public int amount;

}

When using this type as an operation parameter, a SOAP envelope is generated with extra headers called operation and transactionDate, which contain the contents of the operation and transactionDate fields. The SOAP body consists of a wrapper element containing elements called sourceAccount and targetAccount, of the Account data contract type and the amount of the integer type.

You can apply the **MessageHeaderAttribute** and **MessageBodyMemberAttribute** to all fields, properties, and events, regardless of whether they are public, private, protected, or internal.

|  |
| --- |
| **Description: ms730255.note(en-us,VS.100).gifNote:** |
| [KnownTypeAttribute](http://msdn.microsoft.com/en-us/library/system.runtime.serialization.knowntypeattribute.aspx) attributes are ignored in message contracts. If a **KnownTypeAttribute** is required, place it on the operation that is using the message contract in question. |

**Using Custom Types Inside Message Contracts**

Each individual message header and message body part is serialized (turned into XML) using the chosen serialization engine for the service contract where the message is used. The default serialization engine, the **XmlFormatter**, can handle any type that has a data contract, either explicitly (by having the [System.Runtime.Serialization.DataContractAttribute](http://msdn.microsoft.com/en-us/library/system.runtime.serialization.datacontractattribute.aspx)) or implicitly (by being a primitive type, having the [System.SerializableAttribute](http://msdn.microsoft.com/en-us/library/system.serializableattribute.aspx), and so on). For more information, see [Using Data Contracts](http://msdn.microsoft.com/en-us/library/ms733127.aspx).

In the preceding example, the Operation and BankingTransactionData types must have a data contract, and transactionDate is serializable because [DateTime](http://msdn.microsoft.com/en-us/library/system.datetime.aspx) is a primitive (and so has an implicit data contract).

However, it is possible to switch to a different serialization engine, the **XmlSerializer**. If you make such a switch, you should ensure that all of the types used for message headers and body parts are serializable using the **XmlSerializer**.

**Using Arrays Inside Message Contracts**

You can use arrays of repeating elements in message contracts in two ways.

The first is to use a **MessageHeaderAttribute** or a **MessageBodyMemberAttribute** directly on the array. In this case, the entire array is serialized as one element (that is, one header or one body part) with multiple child elements. Consider the class in the following example.

C#

[MessageContract]

public class BankingDepositLog

{

[MessageHeader] public int numRecords

[MessageHeader] public DepositRecord records[];

[MessageHeader] public int branchID;

}

This results in SOAP headers is similar to the following.

<BankingDepositLog>

<numRecords>3</numRecords>

<records>

<DepositRecord>Record1</DepositRecord>

<DepositRecord>Record2</DepositRecord>

<DepositRecord>Record3</DepositRecord>

</records>

<branchID>20643</branchID>

</BankingDepositLog>

An alternative to this is to use the [MessageHeaderArrayAttribute](http://msdn.microsoft.com/en-us/library/system.servicemodel.messageheaderarrayattribute.aspx). In this case, each array element is serialized independently and so that each array element has one header, similar to the following.

<numRecords>3</numRecords>

<records>Record1</records>

<records>Record2</records>

<records>Record3</records>

<branchID>20643</branchID>

The default name for array entries is the name of the member to which the **MessageHeaderArrayAttribute** attributes is applied.

The **MessageHeaderArrayAttribute** attribute inherits from the **MessageHeaderAttribute**. It has the same set of features as the non-array attributes, for example, it is possible to set the order, name, and namespace for an array of headers in the same way you set it for a single header. When you use the **Order** property on an array, it applies to the entire array.

You can apply the **MessageHeaderArrayAttribute** only to arrays, not collections.

**Using Byte Arrays in Message Contracts**

Byte arrays, when used with the non-array attributes (**MessageBodyMemberAttribute** and **MessageHeaderAttribute**), are not treated as arrays but as a special primitive type represented as Base64-encoded data in the resulting XML.

When you use byte arrays with the array attribute **MessageHeaderArrayAttribute**, the results depend on the serializer in use. With the default serializer, the array is represented as an individual entry for each byte. However, when the **XmlSerializer** is selected, (using the [XmlSerializerFormatAttribute](http://msdn.microsoft.com/en-us/library/system.servicemodel.xmlserializerformatattribute.aspx) on the service contract), byte arrays are treated as Base64 data regardless of whether the array or non-array attributes are used.

**Signing and Encrypting Parts of the Message**

A message contract can indicate whether the headers and/or body of the message should be digitally signed and encrypted.

This is done by setting the [System.ServiceModel.MessageContractMemberAttribute.ProtectionLevel](http://msdn.microsoft.com/en-us/library/system.servicemodel.messagecontractmemberattribute.protectionlevel.aspx) property on the **MessageHeaderAttribute** and **MessageBodyMemberAttribute** attributes. The property is an enumeration of the [System.Net.Security.ProtectionLevel](http://msdn.microsoft.com/en-us/library/system.net.security.protectionlevel.aspx) type and can be set to [None](http://msdn.microsoft.com/en-us/library/system.net.security.protectionlevel.aspx) (no encryption or signature), [Sign](http://msdn.microsoft.com/en-us/library/system.net.security.protectionlevel.aspx) (digital signature only), or [EncryptAndSign](http://msdn.microsoft.com/en-us/library/system.net.security.protectionlevel.aspx) (both encryption and a digital signature). The default is **EncryptAndSign**.

For these security features to work, you must properly configure the binding and behaviors. If you use these security features without the proper configuration (for example, attempting to sign a message without supplying your credentials), an exception is thrown at validation time.

For message headers, the protection level is determined individually for each header.

For message body parts, the protection level can be thought of as the "minimum protection level." The body has only one protection level, regardless of the number of body parts. The protection level of the body is determined by the highest **ProtectionLevel** property setting of all the body parts. However, you should set the protection level of each body part to the actual minimum protection level required.

Consider the class in the following code example.

C#

[MessageContract]

public class PatientRecord

{

[MessageHeader(ProtectionLevel=None)] public int recordID;

[MessageHeader(ProtectionLevel=Sign)] public string patientName;

[MessageHeader(ProtectionLevel=EncryptAndSign)] public string SSN;

[MessageBodyMember(ProtectionLevel=None)] public string comments;

[MessageBodyMember(ProtectionLevel=Sign)] public string diagnosis;

[MessageBodyMember(ProtectionLevel=EncryptAndSign)] public string medicalHistory;

}

In this example, the recordID header is not protected, patientName is signed, and SSN is encrypted and signed. At least one body part, medicalHistory, has **EncryptAndSign** applied, and thus the entire message body is encrypted and signed, even though the comments and diagnosis body parts specify lower protection levels.

**Controlling Header and Body Part Names and Namespaces**

In the SOAP representation of a message contract, each header and body part maps to an XML element that has a name and a namespace.

By default, the namespace is the same as the namespace of the service contract that the message is participating in, and the name is determined by the member name to which the **MessageHeaderAttribute** or the **MessageBodyMemberAttribute** attributes are applied.

You can change these defaults by manipulating the [System.ServiceModel.MessageContractMemberAttribute.Name](http://msdn.microsoft.com/en-us/library/system.servicemodel.messagecontractmemberattribute.name.aspx) and [System.ServiceModel.MessageContractMemberAttribute.Namespace](http://msdn.microsoft.com/en-us/library/system.servicemodel.messagecontractmemberattribute.namespace.aspx) (on the parent class of the **MessageHeaderAttribute** and **MessageBodyMemberAttribute** attributes).

Consider the class in the following code example.

C#

[MessageContract]

public class BankingTransaction

{

[MessageHeader] public Operation operation;

[MessageHeader(Namespace="http://schemas.contoso.com/auditing/2005")] public bool IsAudited;

[MessageBodyMember(Name="transactionData")] public BankingTransactionData theData;

}

In this example, the IsAudited header is in the namespace specified in the code, and the body part that represents the theData member is represented by an XML element with the name transactionData.

**Controlling Whether the SOAP Body Parts Are Wrapped**

By default, the SOAP body parts are serialized inside a wrapped element. For example, the following code shows the HelloGreetingMessage wrapper element generated from the name of the **MessageContractAttribute** type in the message contract for the HelloGreetingMessage message.

Visual Basic

<MessageContract> \_

Public Class HelloGreetingMessage

Private localGreeting As String

<MessageBodyMember(Name := "Salutations", Namespace := "http://www.examples.com")> \_

Public Property Greeting() As String

Get

Return localGreeting

End Get

Set(ByVal value As String)

localGreeting = value

End Set

End Property

End Class

'

' The following is the request message, edited for clarity.

'

' <s:Envelope>

' <s:Header>

' <!-- Note: Some header content has been removed for clarity.

' <a:Action>http://GreetingMessage/Action</a:Action>

' <a:To s:mustUnderstand="1"></a:To>

' </s:Header>

' <s:Body u:Id="\_0" xmlns:u="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">

' <HelloGreetingMessage xmlns="Microsoft.WCF.Documentation">

' <Salutations xmlns="http://www.examples.com">Hello.</Salutations>

' </HelloGreetingMessage>

' </s:Body>

' </s:Envelope>

'

C#

[MessageContract]

public class HelloGreetingMessage

{

private string localGreeting;

[MessageBodyMember(

Name = "Salutations",

Namespace = "http://www.examples.com"

)]

public string Greeting

{

get { return localGreeting; }

set { localGreeting = value; }

}

}

/\*

The following is the request message, edited for clarity.

<s:Envelope>

<s:Header>

<!-- Note: Some header content has been removed for clarity.

<a:Action>http://GreetingMessage/Action</a:Action>

<a:To s:mustUnderstand="1"></a:To>

</s:Header>

<s:Body u:Id="\_0" xmlns:u="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">

<HelloGreetingMessage xmlns="Microsoft.WCF.Documentation">

<Salutations xmlns="http://www.examples.com">Hello.</Salutations>

</HelloGreetingMessage>

</s:Body>

</s:Envelope>

\*/

To suppress the wrapper element, set the [IsWrapped](http://msdn.microsoft.com/en-us/library/system.servicemodel.messagecontractattribute.iswrapped.aspx) property to **false**. To control the name and the namespace of the wrapper element, use the [WrapperName](http://msdn.microsoft.com/en-us/library/system.servicemodel.messagecontractattribute.wrappername.aspx) and [WrapperNamespace](http://msdn.microsoft.com/en-us/library/system.servicemodel.messagecontractattribute.wrappernamespace.aspx) properties.

|  |
| --- |
| **Description: ms730255.note(en-us,VS.100).gifNote:** |
| Having more than one message body part in messages that are not wrapped is not compliant with WS-I Basic Profile 1.1 and is not recommended when designing new message contracts. However, it may be necessary to have more than one unwrapped message body part in certain specific interoperability scenarios. If you are going to transmit more than one piece of data in a message body, it is recommended to use the default (wrapped) mode. Having more than one message header in unwrapped messages is completely acceptable. |

**SOAP Action**

SOAP and related Web services standards define a property called **Action** that can be present for every SOAP message sent. The operation's [System.ServiceModel.OperationContractAttribute.Action](http://msdn.microsoft.com/en-us/library/system.servicemodel.operationcontractattribute.action.aspx) and [System.ServiceModel.OperationContractAttribute.ReplyAction](http://msdn.microsoft.com/en-us/library/system.servicemodel.operationcontractattribute.replyaction.aspx) properties control the value of this property.

**SOAP Header Attributes**

The SOAP standard defines the following attributes that may exist on a header:

* **Actor/Role** (**Actor** in SOAP 1.1, **Role** in SOAP 1.2)
* **MustUnderstand**
* **Relay**

The **Actor** or **Role** attribute specifies the Uniform Resource Identifier (URI) of the node for which a given header is intended. The **MustUnderstand** attribute specifies whether the node processing the header must understand it. The **Relay** attribute specifies whether the header is to be relayed to downstream nodes. WCF does not perform any processing of these attributes on incoming messages, except for the **MustUnderstand** attribute, as specified in the "Message Contract Versioning" section later in this topic. However, it allows you to read and write these attributes as necessary, as in the following description.

When sending a message, these attributes are not emitted by default. You can change this in two ways. First, you may statically set the attributes to any desired values by changing the [System.ServiceModel.MessageHeaderAttribute.Actor](http://msdn.microsoft.com/en-us/library/system.servicemodel.messageheaderattribute.actor.aspx), [System.ServiceModel.MessageHeaderAttribute.MustUnderstand](http://msdn.microsoft.com/en-us/library/system.servicemodel.messageheaderattribute.mustunderstand.aspx), and [System.ServiceModel.MessageHeaderAttribute.Relay](http://msdn.microsoft.com/en-us/library/system.servicemodel.messageheaderattribute.relay.aspx) properties, as shown in the following code example. (Note that there is no **Role** property; setting the **Actor** property emits the **Role** attribute if you are using SOAP 1.2).

C#

[MessageContract]

public class BankingTransaction

{

[MessageHeader(Actor="http://auditingservice.contoso.com", MustUnderstand=true)] public bool IsAudited;

[MessageHeader] public Operation operation;

[MessageBodyMember] public BankingTransactionData theData;

}

The second way to control these attributes is dynamically, through code. You can achieve this by wrapping the desired header type in the [MessageHeader](http://msdn.microsoft.com/en-us/library/ms586745.aspx) type (be sure not to confuse this type with the non-generic version) and by using the type together with the **MessageHeaderAttribute**. Then, you can use properties on the **MessageHeader** to set the SOAP attributes, as shown in the following code example.

C#

[MessageContract]

public class BankingTransaction

{

[MessageHeader] public MessageHeader<bool> IsAudited;

[MessageHeader] public Operation operation;

[MessageBodyMember] public BankingTransactionData theData;

}

// application code:

BankingTransaction bt = new BankingTransaction();

bt.IsAudited = new MessageHeader<bool>();

bt.IsAudited.Content = false; // Set IsAudited header value to "false"

bt.IsAudited.Actor="http://auditingservice.contoso.com";

bt.IsAudited.MustUnderstand=true;

If you use both the dynamic and the static control mechanisms, the static settings are used as a default but can later be overridden by using the dynamic mechanism, as shown in the following code.

[C#]

[MessageHeader(MustUnderstand=true)] public MessageHeader<Person> documentApprover;

// later on in the code:

BankingTransaction bt = new BankingTransaction();

bt.documentApprover = new MessageHeader<Person>();

bt.documentApprover.MustUnderstand = false; // override the static default of 'true'

Creating repeated headers with dynamic attribute control is allowed, as shown in the following code.

C#

[MessageHeaderArray] public MessageHeader<Person> documentApprovers[];

On the receiving side, reading these SOAP attributes can only be done if the **MessageHeader** class is used for the header in the type. Examine the **Actor**, **Relay**, or **MustUnderstand** properties of a header of the **MessageHeader** type to discover the attribute settings on the received message.

When a message is received and then sent back, the SOAP attribute settings only go round-trip for headers of the **MessageHeader** type.

**Order of SOAP Body Parts**

In some circumstances, you may need to control the order of the body parts. The order of the body elements is alphabetical by default, but can be controlled by the [System.ServiceModel.MessageBodyMemberAttribute.Order](http://msdn.microsoft.com/en-us/library/system.servicemodel.messagebodymemberattribute.order.aspx) property. This property has the same semantics as the [System.Runtime.Serialization.DataMemberAttribute.Order](http://msdn.microsoft.com/en-us/library/system.runtime.serialization.datamemberattribute.order.aspx) property, except for the behavior in inheritance scenarios (in message contracts, base type body members are not sorted before the derived type body members). For more information, see [Data Member Order](http://msdn.microsoft.com/en-us/library/ms729813.aspx).

In the following example, amount would normally come first because it is first alphabetically. However, the **Order** property puts it into the third position.

C#

[MessageContract]

public class BankingTransaction

{

[MessageHeader] public Operation operation;

[MessageBodyMember(Order=1)] public Account sourceAccount;

[MessageBodyMember(Order=2)] public Account targetAccount;

[MessageBodyMember(Order=3)] public int amount;

}

**Message Contract Versioning**

Occasionally, you may need to change message contracts. For example, a new version of your application may add an extra header to a message. Then, when sending from the new version to the old, the system must deal with an extra header, as well as a missing header when going in the other direction.

The following rules apply for versioning headers:

* WCF does not object to the missing headers—the corresponding members are left at their default values.
* WCF also ignores unexpected extra headers. The one exception to this rule is if the extra header has a **MustUnderstand** attribute set to **true** in the incoming SOAP message—in this case, an exception is thrown because a header that must be understood cannot be processed.

Message bodies have similar versioning rules—both missing and additional message body parts are ignored.

**Inheritance Considerations**

A message contract type can inherit from another type, as long as the base type also has a message contract.

When creating or accessing a message using a message contract type that inherits from other message contract types, the following rules apply:

* All of the message headers in the inheritance hierarchy are collected together to form the full set of headers for the message.
* All of the message body parts in the inheritance hierarchy are collected together to form the full message body. The body parts are ordered according to the usual ordering rules (by **System.ServiceModel.MessageBodyMemberAttribute.Order** property and then alphabetical), with no relevance to their place in the inheritance hierarchy. Using message contract inheritance where message body parts occur at multiple levels of the inheritance tree is strongly discouraged. If a base class and a derived class define a header or a body part with the same name, the member from the base-most class is used to store the value of that header or body part.

Consider the classes in the following code example.

C#

[MessageContract]

public class PersonRecord

{

[MessageHeader(Name="ID")] public int personID;

[MessageBodyMember] public string patientName;

}

[MessageContract]

public class PatientRecord : PersonRecord

{

[MessageHeader(Name="ID")] public int patientID;

[MessageBodyMember] public string diagnosis;

}

The PatientRecord class describes a message with one header called ID. The header corresponds to the personID and not the patientID member, because the base-most member is chosen. Thus, the patientID field is useless in this case. The body of the message contains the diagnosis element followed by the patientName element, because that is the alphabetical order. Notice that the example shows a pattern that is strongly discouraged: both the base and the derived message contracts have message body parts.

**WSDL Considerations**

When generating a Web Services Description Language (WSDL) contract from a service that uses message contracts, it is important to remember that not all message contract features are reflected in the resulting WSDL. Consider the following points:

* WSDL cannot express the concept of an array of headers. When creating messages with an array of headers using the **MessageHeaderArrayAttribute**, the resulting WSDL reflects only one header instead of the array.
* The resulting WSDL document may not reflect some protection-level information.
* The message type generated in the WSDL has the same name as the class name of the message contract type.
* When using the same message contract in multiple operations, multiple message types are generated in the WSDL document. The names are made unique by adding the numbers "2", "3", and so on, for subsequent uses. When importing back the WSDL, multiple message contract types are created and are identical except for their names.

**SOAP Encoding Considerations**

WCF allows you to use the legacy SOAP encoding style of XML, however, its use is not recommended. When using this style (by setting the **Use** property to **Encoded** on the **System.ServiceModel.XmlSerializerFormatAttribute** applied to the service contract), the following additional considerations apply:

* The message headers are not supported; this means that the attribute **MessageHeaderAttribute** and the array attribute **MessageHeaderArrayAttribute** are incompatible with SOAP encoding.
* If the message contract is not wrapped, that is, if the property **IsWrapped** is set to **false**, the message contract can have only one body part.
* The name of the wrapper element for the request message contract must match the operation name. Use the **WrapperName** property of the message contract for this.
* The name of the wrapper element for the response message contract must be the same as the name of the operation suffixed by 'Response'. Use the **WrapperName** property of the message contract for this.
* SOAP encoding preserves object references. For example, consider the following code.

C#

[MessageContract(WrapperName="updateChangeRecord")]

public class ChangeRecordRequest

{

[MessageBodyMember] Person changedBy;

[MessageBodyMember] Person changedFrom;

[MessageBodyMember] Person changedTo;

}

[MessageContract(WrapperName="updateChangeRecordResponse")]

public class ChangeRecordResponse

{

[MessageBodyMember] Person changedBy;

[MessageBodyMember] Person changedFrom;

[MessageBodyMember] Person changedTo;

}

// application code:

ChangeRecordRequest cr = new ChangeRecordRequest();

Person p = new Person("John Doe");

cr.changedBy=p;

cr.changedFrom=p;

cr.changedTo=p;

After serializing the message using SOAP encoding, changedFrom and changedTo do not contain their own copies of p, but instead point to the copy inside the changedBy element.

**Performance Considerations**

Every message header and message body part is serialized independently of the others. Therefore, the same namespaces can be declared again for each header and body part. To improve performance, especially in terms of the size of the message on the wire, consolidate multiple headers and body parts into a single header or body part. For example, instead of the following code:

C#

[MessageContract]

public class BankingTransaction

{

[MessageHeader] public Operation operation;

[MessageBodyMember] public Account sourceAccount;

[MessageBodyMember] public Account targetAccount;

[MessageBodyMember] public int amount;

}

Use this code.

C#

[MessageContract]

public class BankingTransaction

{

[MessageHeader] public Operation operation;

[MessageBodyMember] public OperationDetails details;

}

[DataContract]

public class OperationDetails

{

[DataMember] public Account sourceAccount;

[DataMember] public Account targetAccount;

[DataMember] public int amount;

}

**Event-based Asynchronous and Message Contracts**

The design guidelines for the event-based asynchronous model state that if more than one value is returned, one value is returned as the **Result** property and the others are returned as properties on the [EventArgs](http://msdn.microsoft.com/en-us/library/system.eventargs.aspx) object. One result of this is that if a client imports metadata using the event-based asynchronous command options and the operation returns more than one value, the default **EventArgs** object returns one value as the **Result** property and the remainder are properties of the **EventArgs** object.

If you want to receive the message object as the **Result** property and have the returned values as properties on that object, use the **/messageContract** command option. This generates a signature that returns the response message as the **Result** property on the **EventArgs** object. All internal return values are then properties of the response message object.