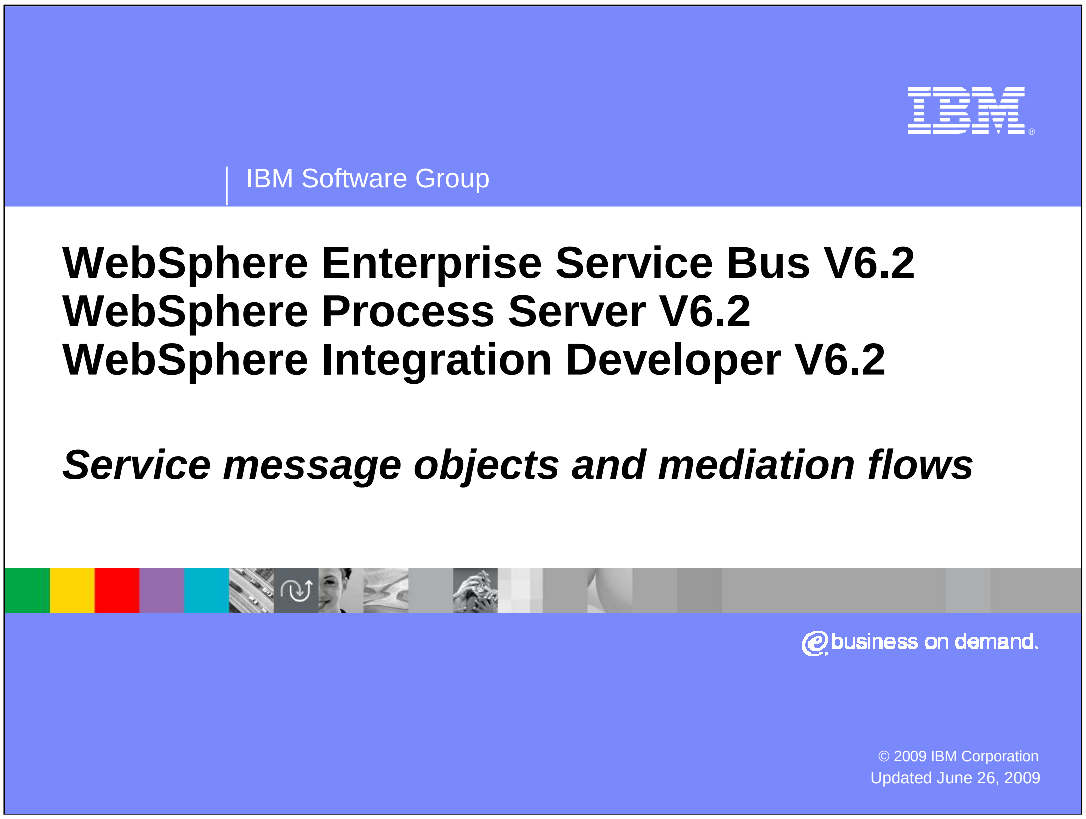


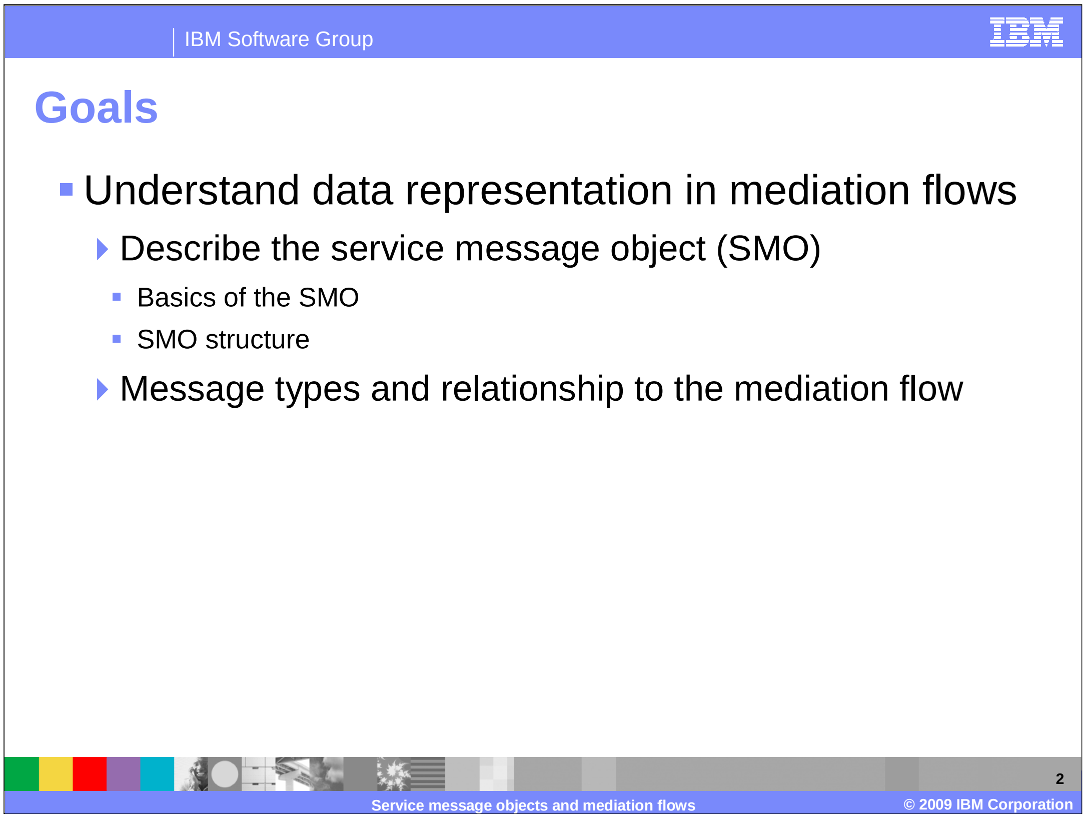
IBM Software Group

WebSphere Enterprise Service Bus V6.2
WebSphere Process Server V6.2
WebSphere Integration Developer V6.2

Service message objects and mediation flows

Updated June 26, 2009

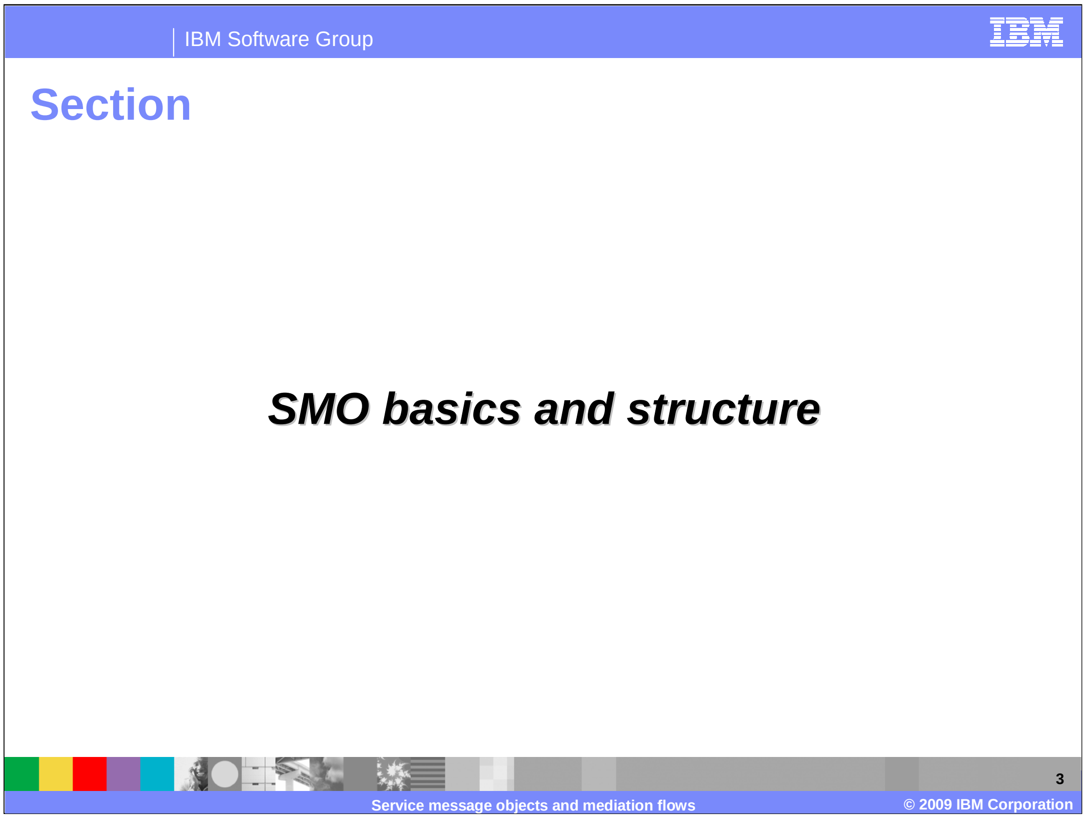
This presentation describes Service Message Objects and how they relate to Mediation Flows.



Goals

- Understand data representation in mediation flows
  - Describe the service message object (SMO)
    - Basics of the SMO
    - SMO structure
  - Message types and relationship to the mediation flow

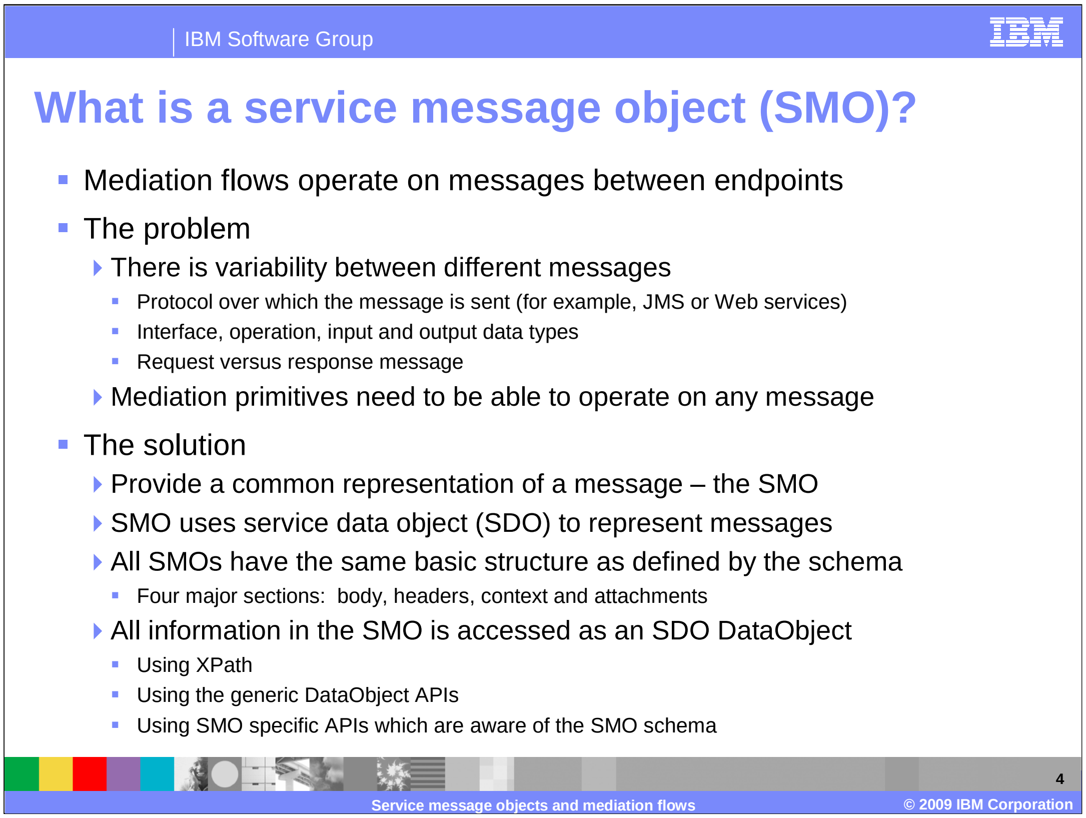
The goal is to provide an understanding of how data is represented in a mediation flow. The data in a flow is described using a Service Message Object (SMO) and this presentation explains some basic characteristics of an SMO and then describes its overall structure. The structure of the application portion of the SMO, referred to as the body or payload, defines a message type. Message types are an important element when considering the logic and flow of a mediation. This presentation describes the relationship between message types and mediation flows.



Section

SMO basics and structure

This section considers the basic characteristics and structure of an SMO.



What is a service message object (SMO)?

- Mediation flows operate on messages between endpoints
- The problem
  - There is variability between different messages
    - Protocol over which the message is sent (for example, JMS or Web services)
    - Interface, operation, input and output data types
    - Request versus response message
  - Mediation primitives need to be able to operate on any message
- The solution
  - Provide a common representation of a message – the SMO
  - SMO uses service data object (SDO) to represent messages
  - All SMOs have the same basic structure as defined by the schema
    - Four major sections: body, headers, context and attachments
  - All information in the SMO is accessed as an SDO DataObject
    - Using XPath
    - Using the generic DataObject APIs
    - Using SMO specific APIs which are aware of the SMO schema

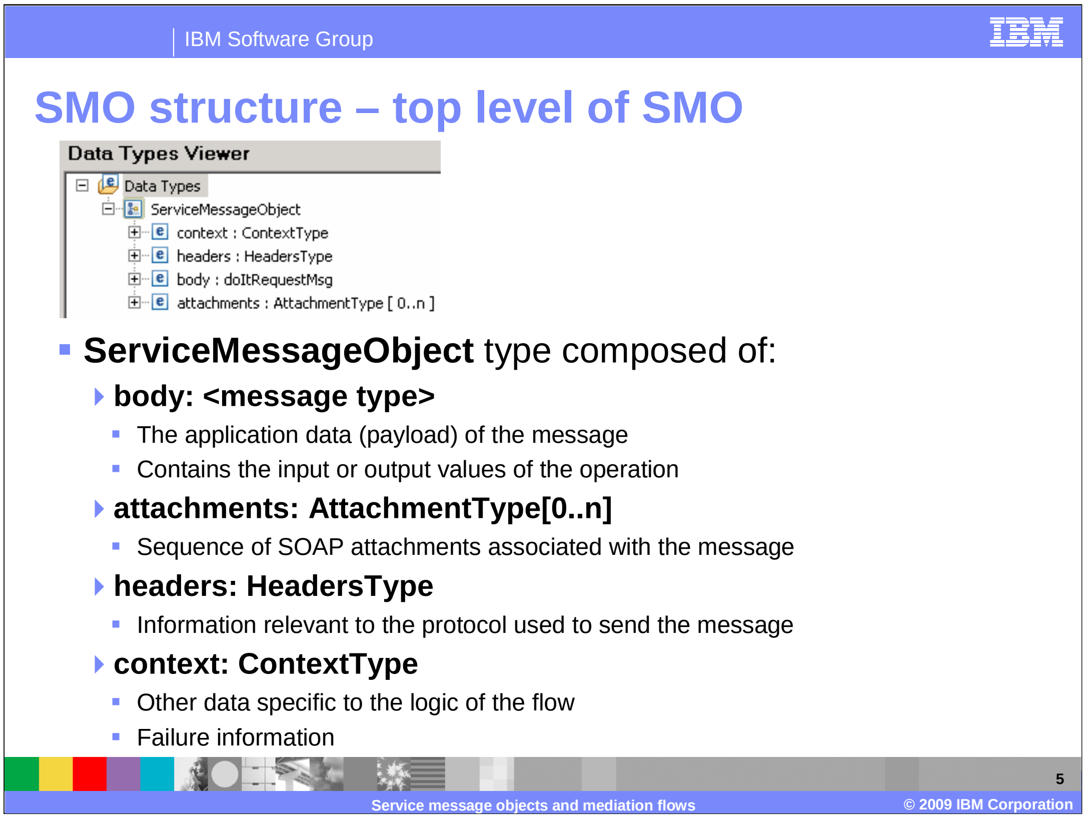
In order to understand what a Service Message Object is, you must first understand some characteristics of a mediation flow. The primary function of a mediation flow is to operate on a message between endpoints, where a service requestor and a service provider are those endpoints. However, this presents a problem.

The first point is that a message can take on many different forms, because the protocol used to send a message, whether JMS or Web services, can vary. Also, each message is different depending upon the interface and operation associated with the message and whether this is the request side or response side of the interaction between the requestor and provider.

The next point to understand is that within the mediation flow, mediation primitives are used to operate on the message. Mediation primitives examine and update the message contents and therefore must understand what is contained in the message. The solution is to provide mediation primitives with some kind of a common representation of a message, and that is what a Service Message Object does. SMOs provide a common representation of a message that accounts for differing protocols and differing interfaces, operations and parameters that the message represents.

SMOs are built using Service Data Object (SDO) technology. SDO uses a schema that describes the basic structure of an SMO which is composed of four major sections. The body of the message represents the specific interface, operation and parameters relevant to this message. The headers section of the message represents information about the protocol over which the message was sent. The context section represents data that is important to the internal logic of the flow itself. The attachments section contains a list of attachments that are associated with a SOAP message. Each of these major sections of the SMO is examined in more detail in subsequent slides.

The data within an SMO is accessed using SDO, specifically the SDO DataObject, which enables access using XPath, the generic DataObject APIs, and some SMO specific APIs that are aware of the SMO schema.



IBM Software Group

SMO structure – top level of SMO

Data Types Viewer

```
Data Types
  ServiceMessageObject
    context : ContextType
    headers : HeadersType
    body : doItRequestMsg
    attachments : AttachmentType [ 0..n ]
```

- **ServiceMessageObject** type composed of:
  - **body: <message type>**
    - The application data (payload) of the message
    - Contains the input or output values of the operation
  - **attachments: AttachmentType[0..n]**
    - Sequence of SOAP attachments associated with the message
  - **headers: HeadersType**
    - Information relevant to the protocol used to send the message
  - **context: ContextType**
    - Other data specific to the logic of the flow
    - Failure information

5

Service message objects and mediation flows

© 2009 IBM Corporation

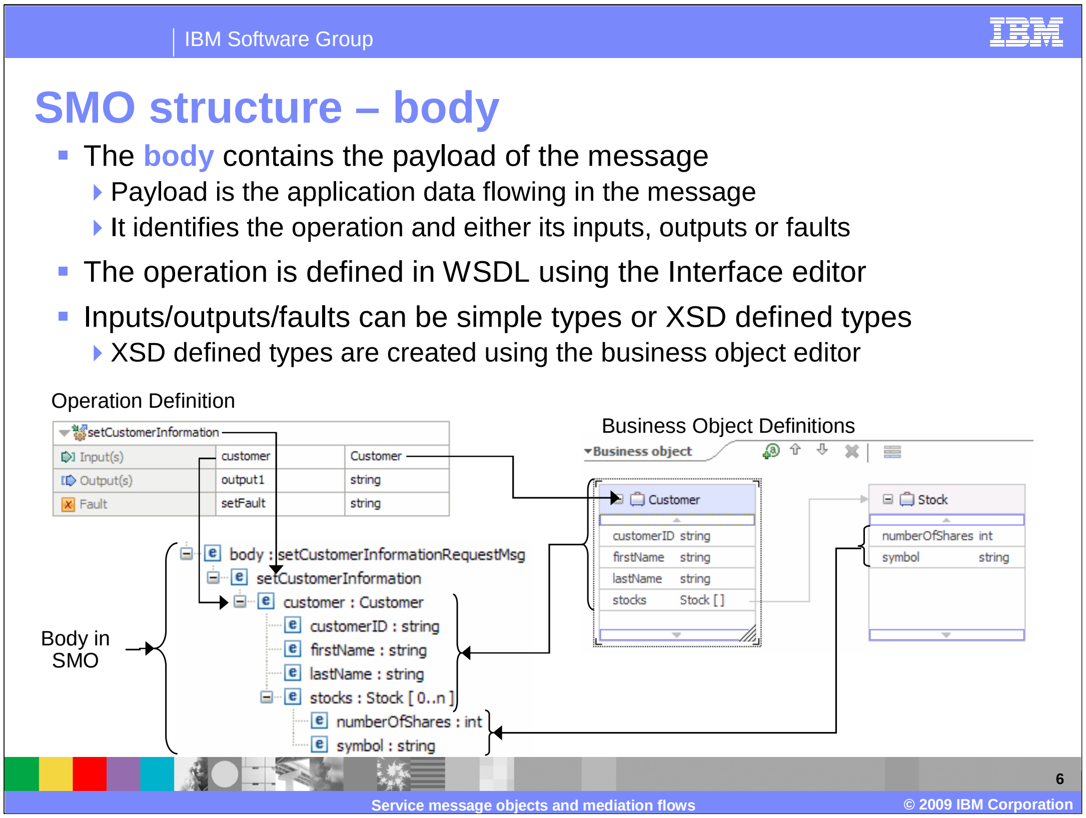
Shown here is an illustration of the four major sections of an SMO. The screen capture at the top is from the data types viewer of the XPath expression builder tool, which is used to drill down into the structure of the SMO. The top level of the SMO is defined by the type ServiceMessageObject which contains four elements, the body, the attachments, the headers, and the context.

The body of the SMO contains the application data, sometimes referred to as the payload. This is the data that is relevant to the endpoints, the inbound coming from the service requestor and the outbound going to the service provider. The body describes the operation being performed and the inputs or outputs of that operation. Also carrying application data is the attachments section that contains a sequence of attachments for SOAP messages, if there are any attachments associated with the message.

The headers of the SMO contain protocol specific information associated with the protocol over which the message is being sent.

The context of the SMO contains data required by the logic of the flow. This data exists within the flow itself but is not passed to or from the requestor or provider. Under certain conditions, error information is also added to the context.

Each of these sections of the SMO is examined more closely in the subsequent slides.



SMO structure – body

- The **body** contains the payload of the message
  - Payload is the application data flowing in the message
  - It identifies the operation and either its inputs, outputs or faults
- The operation is defined in WSDL using the Interface editor
- Inputs/outputs/faults can be simple types or XSD defined types
  - XSD defined types are created using the business object editor

Operation Definition

setCustomerInformation		
Input(s)	customer	Customer
Output(s)	output1	string
Fault	setFault	string

Business Object Definitions

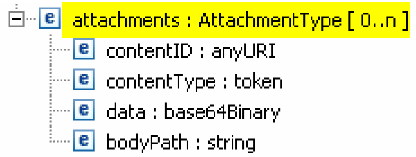
Customer	
customerID	string
firstName	string
lastName	string
stocks	Stock [ ]

Stock	
numberOfShares	int
symbol	string

```
body : setCustomerInformationRequestMsg
  setCustomerInformation
    customer : Customer
      customerID : string
      firstName : string
      lastName : string
      stocks : Stock [ 0..n ]
        numberOfShares : int
        symbol : string
```

The body of the SMO contains the payload, which is the application data that flows between a service requestor and service provider. The body represents a specific operation on a specific interface. The data associated with that operation is also contained in the body and will be either the inputs, the outputs or the faults defined for the operation. The interface is a WSDL defined interface, and the Interface Editor in WebSphere® Integration Developer can be used to define it. The inputs, outputs and faults can be simple types or they can be XSD defined types. The Business Object Editor in WebSphere Integration Developer can be used to define these types. The illustration at the bottom of this slide shows the relationship between an interface defined in the Interface Editor, a business object defined in the Business Object Editor and the contents of the body of an SMO. In the lower left section is an SMO body expanded to show the individual elements. Starting at the upper left in the Interface Editor, is an operation definition for an operation called setCustomerInformation. The body contains a section called setCustomerInformation as its top level. This operation has an input called customer, defined by a Customer Business Object. Since this SMO body represents the request flow, it contains the inputs. Within the SMO, the setCustomerInformation section contains a customer section. To understand what is contained in the customer section, look at the Business Object Editor in the upper right where the Customer business object is defined. It is composed of four fields, a customerID, firstName and lastName, which are all strings, and a stocks field, which is an array of Stock business objects. A Stock business object is composed of two fields, numberOfShares, which is an int, and symbol, which is a string. The SMO body contains the same elements as the Customer business object defined in the Business Object Editor. The body of the SMO truly is a representation of an operation and the data associated with that operation.

SMO structure – attachments



```
attachments : AttachmentType [ 0..n ]
    contentID : anyURI
    contentType : token
    data : base64Binary
    bodyPath : string
```

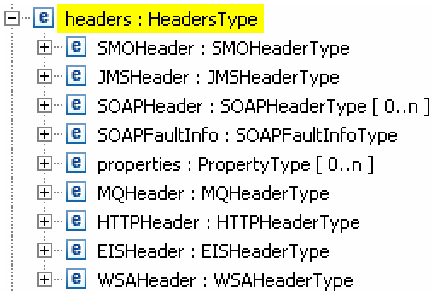
- The attachments contain SOAP attachments
  - Only unreferenced attachments supported
  - Only applicable to JAX-WS and default SCA bindings
  - Sequence of AttachmentType
- AttachmentType
  - contentID – value of Content-ID header of the MIME part
  - contentType – value of Content-Type header of the MIME part
  - data – the attachment data
  - bodyPath – not used in this release

The attachments section of the SMO contains the SOAP attachments associated with the message. In this release, only unreferenced attachments are supported. Unreferenced attachments are MIME parts included in a SOAP message, but they are not defined in the WSDL portType of the message. Attachments can only be used with mediation flows that are wired to JAX-WS Web service bindings or default SCA bindings.

The attachments section is a sequence of AttachmentType, with one AttachmentType for each of the attachments in the message. The AttachmentType contains a contentID and contentType element, which carry the values of the Content-ID and Content-Type from the MIME header. The attachment data is carried in the data element. In the current release, which only supports unreferenced attachments, the bodyPath element is not used.

SMO structure – headers

- The **headers** carry protocol information based on binding type
  - Header elements are populated by inbound message
    - Binding type of the export for request flows
    - Binding type of the import for response flows
  - Outbound message constructed using header elements
    - Binding type of the import for request flows
    - Binding type of the export for response flows
    - What actually applies depends upon the binding type
- Next slides introduce the various header types

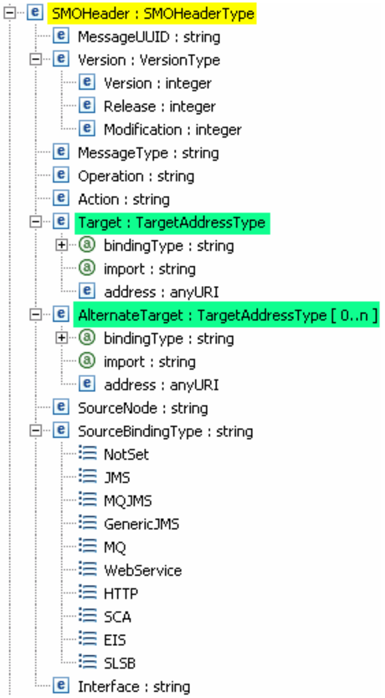


```
headers : HeadersType
  SMOHeader : SMOHeaderType
  JMSHeader : JMSHeaderType
  SOAPHeader : SOAPHeaderType [ 0..n ]
  SOAPFaultInfo : SOAPFaultInfoType
  properties : PropertyType [ 0..n ]
  MQHeader : MQHeaderType
  HTTPHeader : HTTPHeaderType
  EISHeader : EISHeaderType
  WSAHeader : WSAHeaderType
```

The headers section of the SMO contains information associated with the protocol over which the message is sent or received. The binding type of the export or import determines which of these header types apply. For inbound messages, the header elements are populated, and for outbound messages, the header elements are considered when constructing the message. Inbound messages apply to exports for request flows and imports for response flows. Conversely, outbound messages apply to imports for request flows and exports for response flows. The degree to which the header elements apply to construction of an outbound message varies by binding type. The next few slides provide a brief introduction to the header types contained in the SMO.

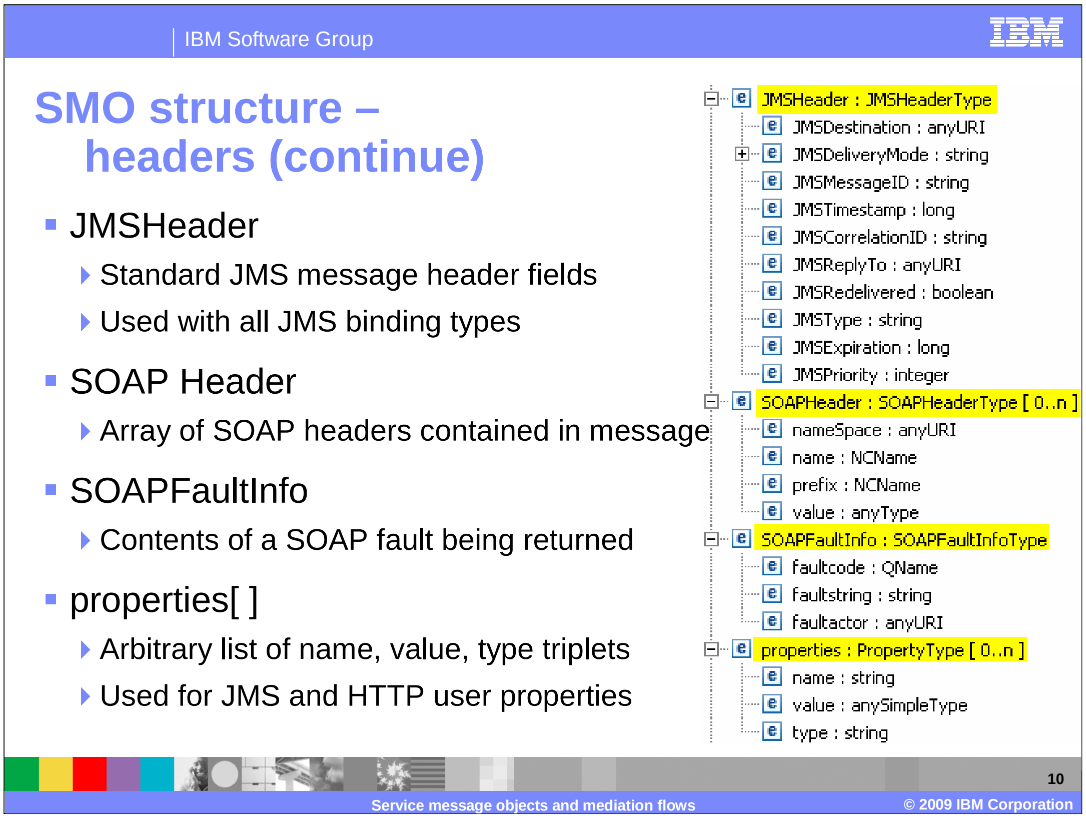
SMO structure – headers (continue)

- SMOHeader
  - Protocol independent information about the message
  - Always contained in SMO
  - Contains the target information for dynamic addressing:
    - Target
    - AlternateTarget sequence



```
SMOHeader : SMOHeaderType
  MessageUUID : string
  Version : VersionType
    Version : integer
    Release : integer
    Modification : integer
  MessageType : string
  Operation : string
  Action : string
  Target : TargetAddressType
    bindingType : string
    import : string
    address : anyURI
  AlternateTarget : TargetAddressType [ 0..n ]
    bindingType : string
    import : string
    address : anyURI
  SourceNode : string
  SourceBindingType : string
    NotSet
    JMS
    MQJMS
    GenericJMS
    MQ
    WebService
    HTTP
    SCA
    EIS
    SLSB
  Interface : string
```

The first of the header types, the SMOHeader, contains protocol independent information that defines the message. This includes elements such as a unique message ID, the version number of the SMO schema, the binding type, and the interface used to initiate the flow. The SMOHeader is always present in the service message object. One of the key functions of the SMOHeader is the support for dynamic addressing by callouts and service invoke primitives. The Target element contains a TargetAddressType used for dynamic addressing, and the AlternateTarget contains a sequence of TargetAddressType which can be used for retry processing.



IBM Software Group

SMO structure – headers (continue)

- JMSHeader
  - Standard JMS message header fields
  - Used with all JMS binding types
- SOAP Header
  - Array of SOAP headers contained in message
- SOAPFaultInfo
  - Contents of a SOAP fault being returned
- properties[ ]
  - Arbitrary list of name, value, type triplets
  - Used for JMS and HTTP user properties

- JMSHeader : JMSHeaderType
  - JMSDestination : anyURI
  - JMSDeliveryMode : string
  - JMSMessageID : string
  - JMSTimestamp : long
  - JMSCorrelationID : string
  - JMSReplyTo : anyURI
  - JMSRedelivered : boolean
  - JMSType : string
  - JMSExpiration : long
  - JMSPriority : integer
- SOAPHeader : SOAPHeaderType [0..n]
  - nameSpace : anyURI
  - name : NCName
  - prefix : NCName
  - value : anyType
- SOAPFaultInfo : SOAPFaultInfoType
  - faultcode : QName
  - faultstring : string
  - faultactor : anyURI
- properties : PropertyType [0..n]
  - name : string
  - value : anySimpleType
  - type : string

10

Service message objects and mediation flows

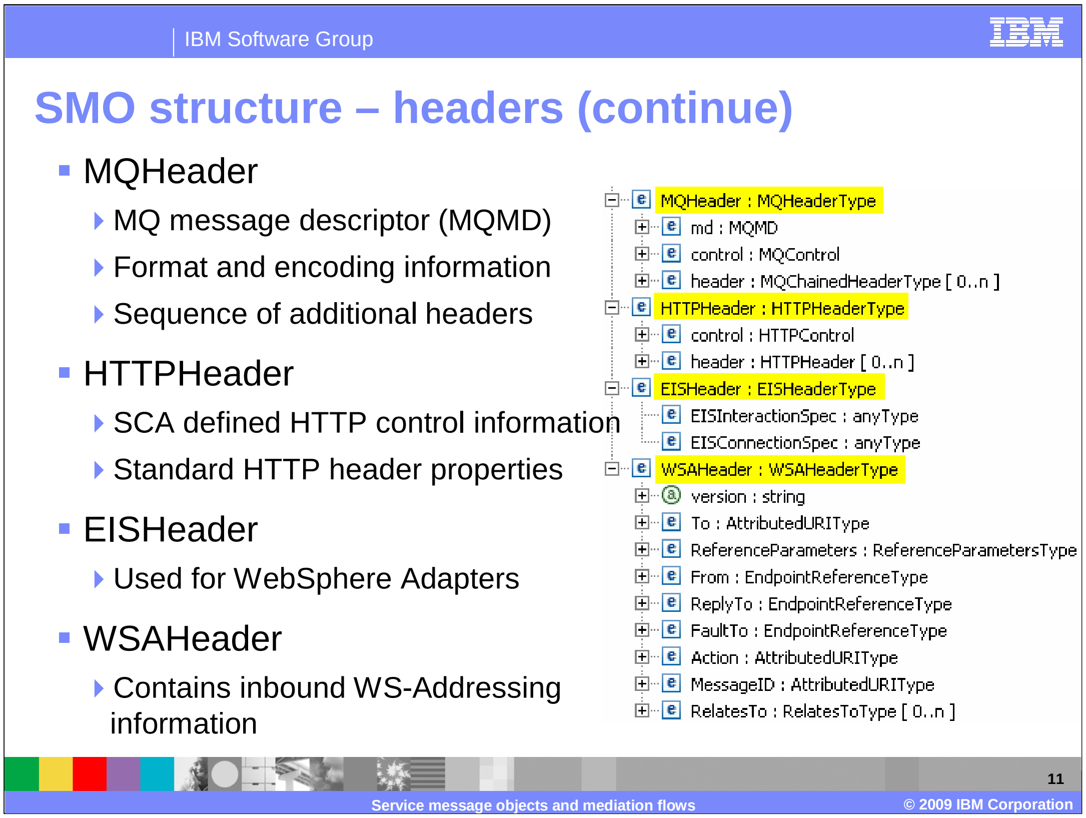
© 2009 IBM Corporation

The JMSHeader type contains the standard JMS message header properties, which are sent with all JMS messages. This header applies to all of the JMS binding types. Note that JMS user properties are not placed into this header, but are contained in the properties header described later in this slide.

The SOAPHeader type contains an array of the SOAP headers from the SOAP message. SOAP headers are defined by XSDs, so the actual value of a header is contained in an anyType element.

The SOAPFaultInfo type contains information about SOAP faults that are being returned.

The properties contain a sequence of PropertyType. It provides the ability to include an arbitrary list of name, value, type triplets that can be used to represent any information. When JMS and HTTP protocols are being used, this array contains the user defined properties associated with the JMS or HTTP message.



IBM Software Group

SMO structure – headers (continue)

- MQHeader
  - MQ message descriptor (MQMD)
  - Format and encoding information
  - Sequence of additional headers
- HTTPHeader
  - SCA defined HTTP control information
  - Standard HTTP header properties
- EISHeader
  - Used for WebSphere Adapters
- WSAHeader
  - Contains inbound WS-Addressing information

```
MQHeader : MQHeaderType
    md : MQMD
    control : MQControl
    header : MQChainedHeaderType [ 0..n ]
HTTPHeader : HTTPHeaderType
    control : HTTPControl
    header : HTTPHeader [ 0..n ]
EISHeader : EISHeaderType
    EISInteractionSpec : anyType
    EISConnectionSpec : anyType
WSAHeader : WSAHeaderType
    version : string
    To : AttributedURIType
    ReferenceParameters : ReferenceParametersType
    From : EndpointReferenceType
    ReplyTo : EndpointReferenceType
    FaultTo : EndpointReferenceType
    Action : AttributedURIType
    MessageID : AttributedURIType
    RelatesTo : RelatesToType [ 0..n ]
```

11

Service message objects and mediation flows

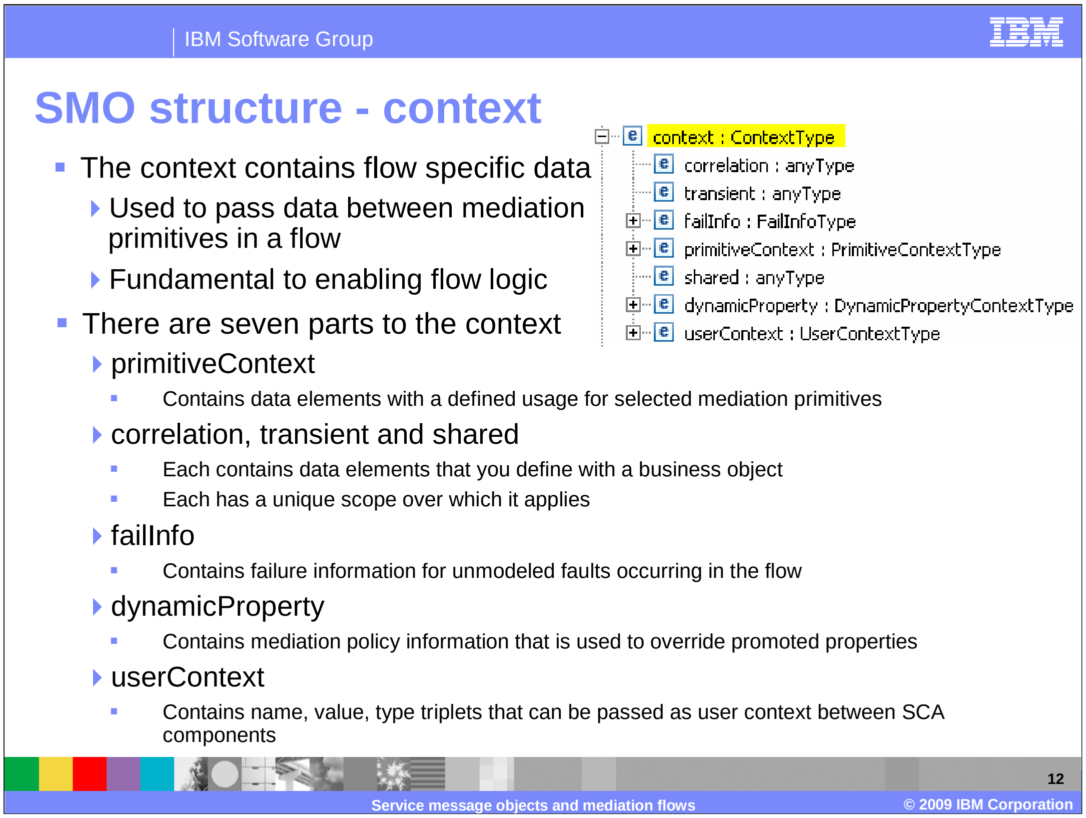
© 2009 IBM Corporation

The MQHeader type contains header information from an MQ message. It contains the MQ message descriptor which is contained in all MQ messages. The format and encoding information for the MQMD is contained in the control element. Following that is a sequence of other MQ headers contained in the message. The structure of the MQ headers in the SMO is slightly simplified from the structure contained in the actual MQ message, eliminating the need to walk a chain of format and encoding information when traversing the headers.

The HTTPHeader type contains the HTTP control elements that are defined for HTTP by SCA. It also contains a sequence of standard HTTP properties, which are name value pairs defined by the HTTP specification. Note that HTTP user properties are not placed in this header, but are found in the properties header described on the previous slide.

The EISHeader type is used for WebSphere adapters which follow the J2EE™ Connector architecture (JCA). It contains the data used in the JCA defined ConnectionSpec and InterationSpec.

The WSAHeader type is used for WS-Addressing related information. It holds WS-Addressing header information from an inbound message for use by the mediation flow. The values in this header are not used for outbound messages.



SMO structure - context

- context : ContextType
  - correlation : anyType
  - transient : anyType
  - failInfo : FailInfoType
  - primitiveContext : PrimitiveContextType
  - shared : anyType
  - dynamicProperty : DynamicPropertyContextType
  - userContext : UserContextType

- The context contains flow specific data
  - Used to pass data between mediation primitives in a flow
  - Fundamental to enabling flow logic
- There are seven parts to the context
  - primitiveContext
    - Contains data elements with a defined usage for selected mediation primitives
  - correlation, transient and shared
    - Each contains data elements that you define with a business object
    - Each has a unique scope over which it applies
  - failInfo
    - Contains failure information for unmodeled faults occurring in the flow
  - dynamicProperty
    - Contains mediation policy information that is used to override promoted properties
  - userContext
    - Contains name, value, type triplets that can be passed as user context between SCA components

The context section of the SMO is used for passing data that is required internally by the mediation flow logic. Mediation primitives can place data into the context so that it can be accessed by subsequent mediation primitives in the flow. The context is divided into seven sections.

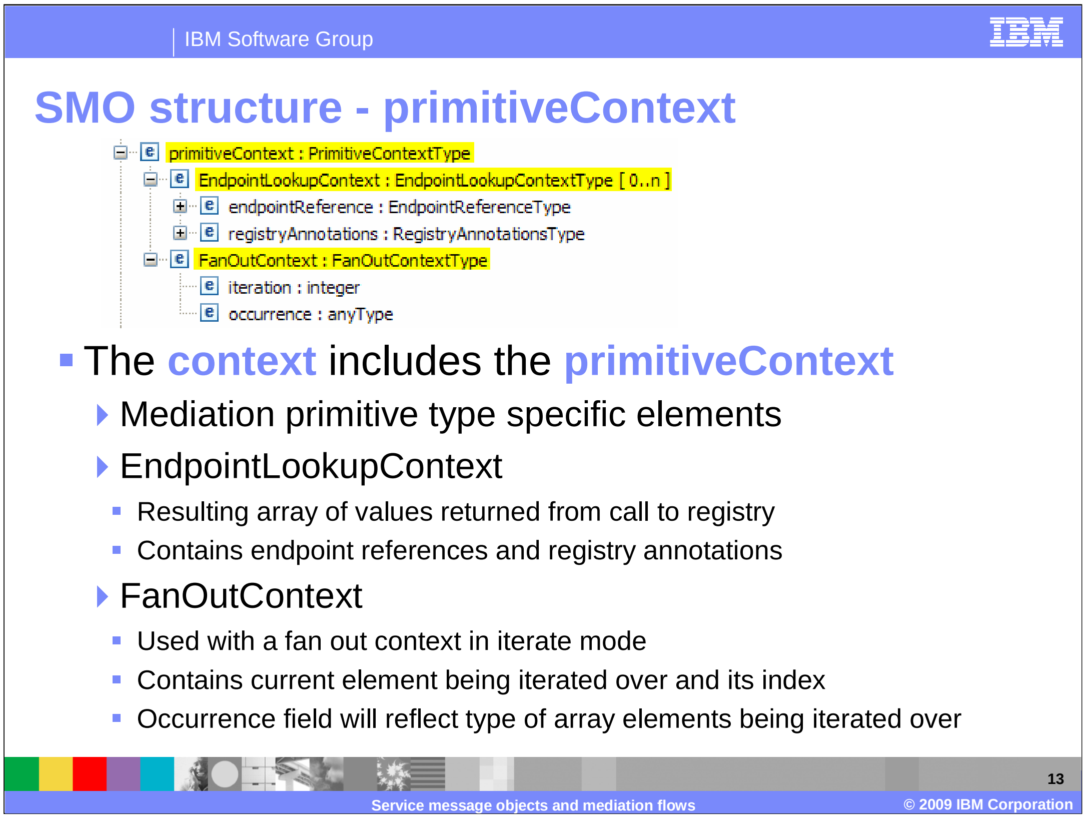
The primitiveContext contains data whose definition and usage is specific to particular types of mediation primitives.

The correlation, transient and shared contexts contain data that is defined by you to meet specific requirements in your flow logic. The scope over which each of these contexts is used is different, addressing different needs in passing data between primitives.

The failInfo context is used to carry failure information when an unmodeled fault occurs during the flow.

The dynamicProperty context contains mediation policy settings which override the values for promoted properties in a flow instance.

The userContext is somewhat different in scope than the other contexts because the data it contains is not just for internal use within the flow. The userContext represents the user context that is propagated between SCA modules. It is composed of name, value, type triplets.



SMO structure - primitiveContext

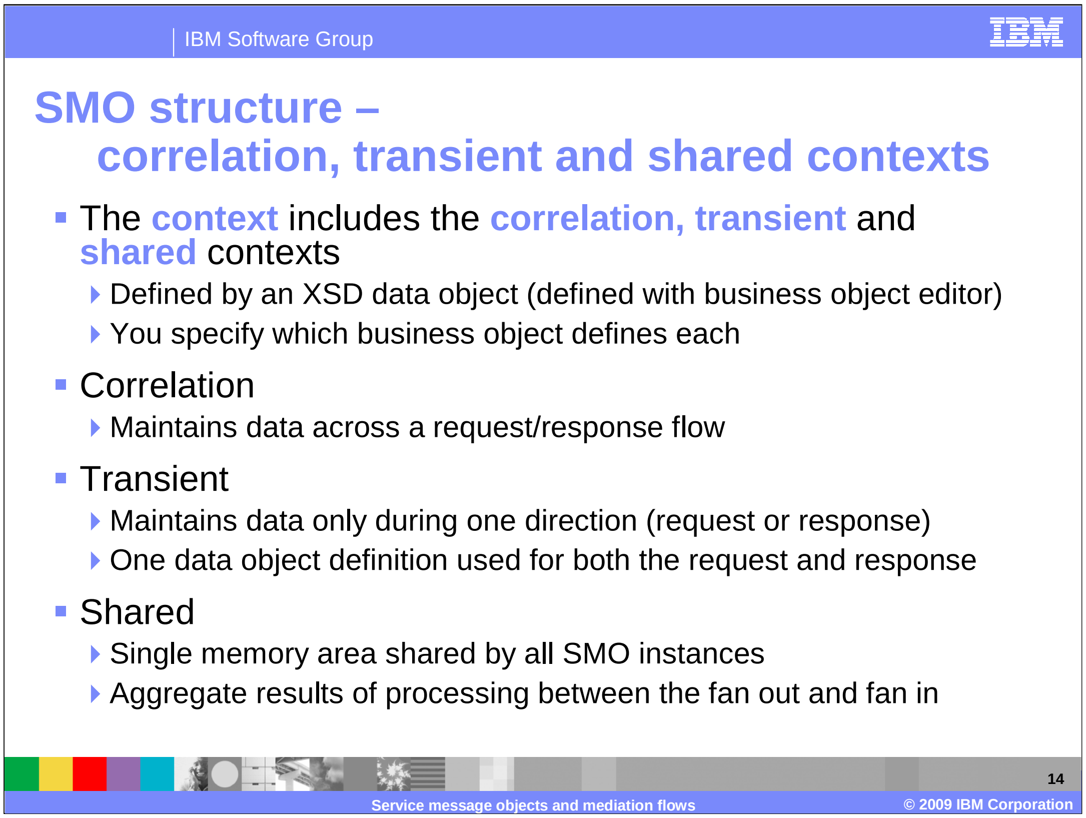
```
primitiveContext : PrimitiveContextType
  EndpointLookupContext : EndpointLookupContextType [ 0..n ]
    endpointReference : EndpointReferenceType
    registryAnnotations : RegistryAnnotationsType
  FanOutContext : FanOutContextType
    iteration : integer
    occurrence : anyType
```

- The **context** includes the **primitiveContext**
  - Mediation primitive type specific elements
  - EndpointLookupContext
    - Resulting array of values returned from call to registry
    - Contains endpoint references and registry annotations
  - FanOutContext
    - Used with a fan out context in iterate mode
    - Contains current element being iterated over and its index
    - Occurrence field will reflect type of array elements being iterated over

The primitiveContext is used in those cases where the function delivered by a primitive needs to provide data for your use with other primitives in the flow. There are two primitives that fit into this category, the endpoint lookup primitive and fan out primitive.

The EndpointLookupContext is used to store the information returned from a query made to the WebSphere Service Registry and Repository. It contains an array of endpoint references and registry annotations for all the service endpoints that satisfied the query. You can then make use of this information when your flow logic is trying to determine which endpoint should be used for a service call.

The FanOutContext is used when a fan out primitive has been configured in iterate mode. The current array element and its index are stored in this context, allowing your flow logic to know which element is being worked on during this iteration of the flow. You can see that the occurrence element shows as an anyType in this screen capture. However, in a flow containing a fan out, it is defined as the specific element type of the array elements.



IBM Software Group

SMO structure – correlation, transient and shared contexts

- The **context** includes the **correlation, transient** and **shared** contexts
  - Defined by an XSD data object (defined with business object editor)
  - You specify which business object defines each
- Correlation
  - Maintains data across a request/response flow
- Transient
  - Maintains data only during one direction (request or response)
  - One data object definition used for both the request and response
- Shared
  - Single memory area shared by all SMO instances
  - Aggregate results of processing between the fan out and fan in

14

Service message objects and mediation flows

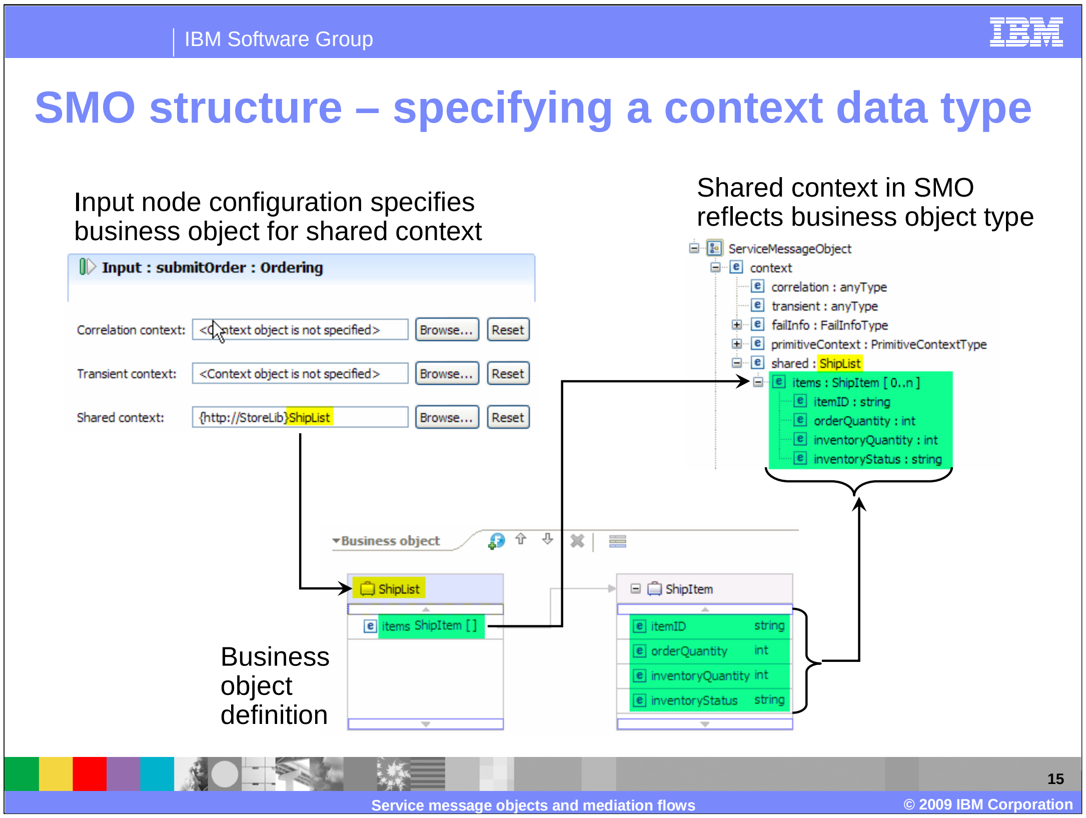
© 2009 IBM Corporation

The context section of an SMO contains the correlation, transient and shared contexts, which have several things in common. For instance, they are each used to pass flow specific information between mediation primitives. An XSD defined data object, such as one you created using the business object editor, is used to define the elements of a correlation, transient or shared context. You associate these with a flow by specifying the appropriate business object on the input node of the mediation flow. Correlation, transient and shared contexts differ, however, in the scope over which they maintain data.

A correlation context retains data across a request/response flow and therefore can be used to pass data from a mediation primitive on the request flow side to a mediation primitive on the response flow side.

A transient context can be used during either the request or response flow but does not retain the data set in the request for access by the response. Only one business object is used to define the transient context. Therefore, if you want to use it on both the request and response flows, the business object definition must contain the fields required for both sides of the flow. This is true even though the values set in the request flow are not available for the response flow.

A shared context provides a single memory area shared by all SMO instances. This is needed when doing a splitting and aggregating scenario using the fan out and fan in primitives. The SMO is cloned at the start of each iteration between the fan out and fan in. Therefore, the shared context is needed to provide a location where the aggregated information can be built up during the iterative processing.



IBM Software Group

SMO structure – specifying a context data type

Input node configuration specifies business object for shared context

Input : submitOrder : Ordering

Correlation context:	<Context object is not specified>	Browse...	Reset
Transient context:	<Context object is not specified>	Browse...	Reset
Shared context:	{http://StoreLib}ShipList	Browse...	Reset

Shared context in SMO reflects business object type

```
ServiceMessageObject
  context
    correlation : anyType
    transient : anyType
    failInfo : FailInfoType
    primitiveContext : PrimitiveContextType
    shared : ShipList
      items : ShipItem [ 0..n ]
        itemID : string
        orderQuantity : int
        inventoryQuantity : int
        inventoryStatus : string
```

Business object definition

Business object

ShipList

items	ShipItem [ ]

ShipItem

itemID	string
orderQuantity	int
inventoryQuantity	int
inventoryStatus	string

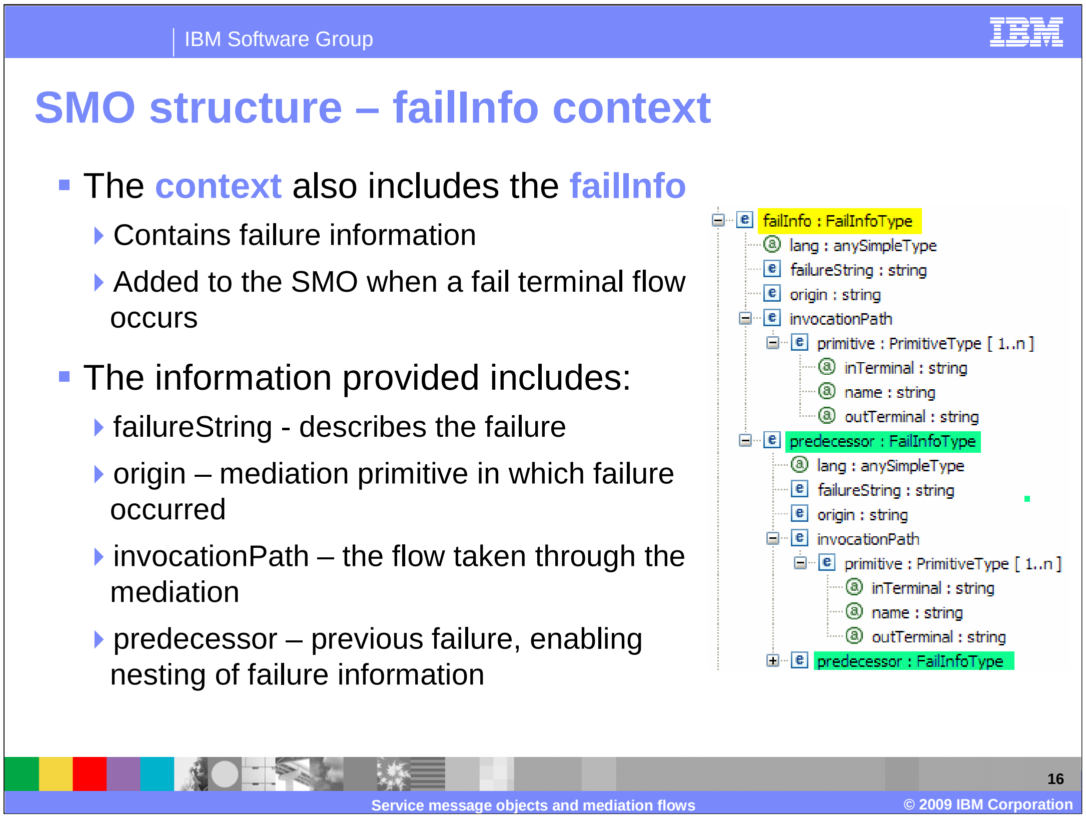
15

Service message objects and mediation flows

This slide illustrates the specification of a business object for the shared context and the resulting affect on the definition of the SMO.

Starting on the upper left side you can see the properties panel for the input node of a flow. The shared context has been configured to be defined by a business object type called ShipList. On the lower part of the slide you can see the business object editor showing that ShipList is an array of ShipItem, where a ShipItem is composed of the fields itemID, orderQuantity, inventoryQuantity and inventoryStatus. Looking at the upper right of the slide you can see that the SMO definition for the shared context reflects the definition of the ShipList business object.

Notice that in this illustration the transient and correlation contexts are not configured and therefore show as anyType in the SMO definition. Either or both of these might also be configured to a business object type with similar results on the SMO definition.



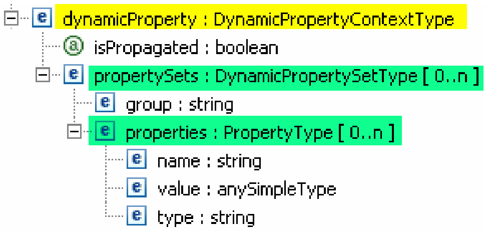
SMO structure – failInfo context

- The **context** also includes the **failInfo**
  - Contains failure information
  - Added to the SMO when a fail terminal flow occurs
- The information provided includes:
  - failureString - describes the failure
  - origin – mediation primitive in which failure occurred
  - invocationPath – the flow taken through the mediation
  - predecessor – previous failure, enabling nesting of failure information

```
failInfo : FailInfoType
    lang : anySimpleType
    failureString : string
    origin : string
    invocationPath
        primitive : PrimitiveType [ 1..n ]
            inTerminal : string
            name : string
            outTerminal : string
    predecessor : FailInfoType
        lang : anySimpleType
        failureString : string
        origin : string
        invocationPath
            primitive : PrimitiveType [ 1..n ]
                inTerminal : string
                name : string
                outTerminal : string
        predecessor : FailInfoType
```

Shown on the right is an expanded view of the failInfo portion of the context section, which is used to contain information about a failure that occurred during the flow. It is only populated when a failure occurs in a mediation primitive and the mediation primitive has its fail terminal wired to another primitive or node. This allows a mediation flow to examine a failure and determine how the failure should be handled. The failInfo contains a string that describes the failure, the name of the mediation primitive in which the failure occurred and information about the path taken through the flow before the failure. In the event that a second failure occurs while processing the first failure, the predecessor section is used to retain the information about the original failure. Therefore, you are provided with a nesting of failure information when multiple failures occur in the flow.

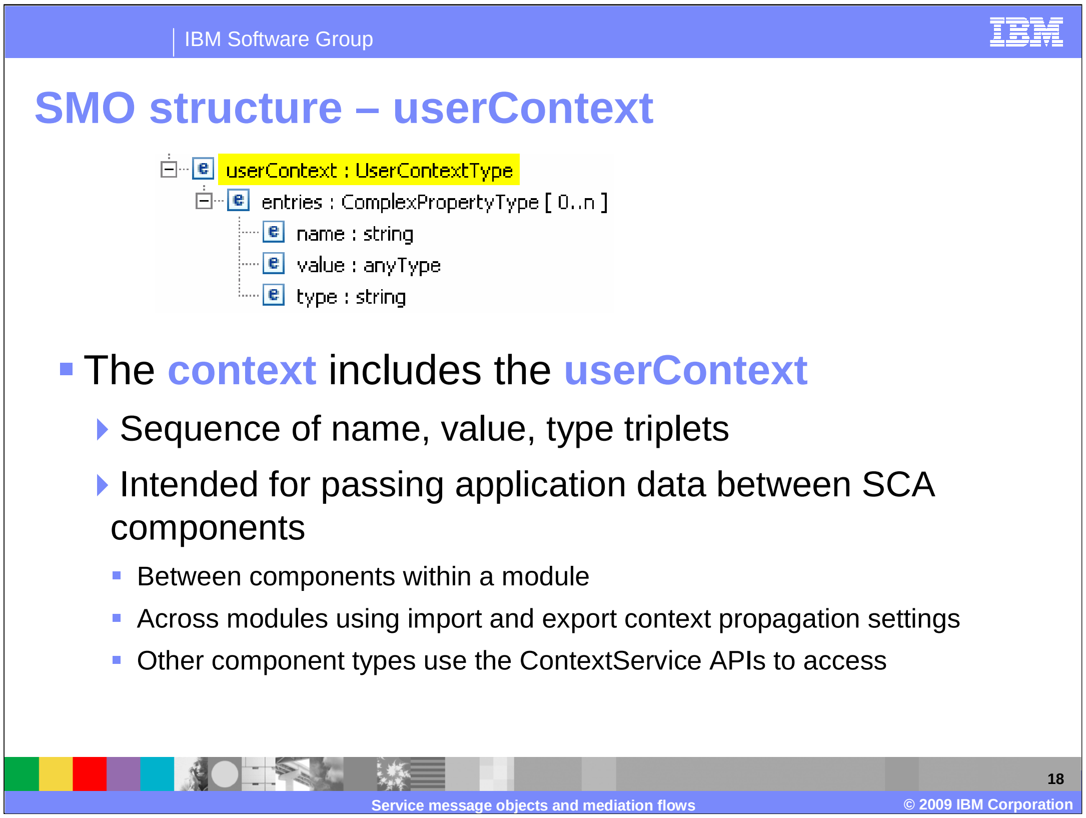
SMO structure – dynamicProperty context



```
dynamicProperty : DynamicPropertyContextType
  isPropagated : boolean
  propertySets : DynamicPropertySetType [ 0..n ]
    group : string
    properties : PropertyType [ 0..n ]
      name : string
      value : anySimpleType
      type : string
```

- The **context** also includes the **dynamicProperty** context
  - Contains policy values set by the policy resolution primitive
    - Lookup from WebSphere Service Registry and Repository
  - Overrides values for promoted properties
  - Grouped into policy sets corresponding to promoted property group names
  - Each property is a name, value, type triplet
  - Can be propagated from request flow to response flow

The dynamicProperty context is shown here. This context is provided as a place to store policy values. The policy values are looked up from WebSphere Service Registry and Repository by the policy resolution primitive. They provide dynamic flow instance overrides to promoted properties in the flow. In the context, they are organized by groups, where each group has a name and a sequence of properties. The properties are represented as name, value, type triplets. The name corresponds to alias names given to promoted properties. The property values in this context are applied to promoted properties in the flow that have the same group name and alias name. This context can optionally be propagated from the request flow to the response flow if needed, thus preventing the requirement for a subsequent lookup with the policy resolution primitive.



IBM Software Group

SMO structure – userContext

```
userContext : UserContextType
  entries : ComplexPropertyType [ 0..n ]
    name : string
    value : anyType
    type : string
```

- The context includes the userContext
  - Sequence of name, value, type triplets
  - Intended for passing application data between SCA components
    - Between components within a module
    - Across modules using import and export context propagation settings
    - Other component types use the ContextService APIs to access

18

Service message objects and mediation flows

The last context to describe is the userContext. It contains a sequence of name, value, type triplets. The userContext is not intended for passing data between mediation primitives in a flow as all the other contexts are, but is instead intended for passing data between SCA components. The user context information is passed between components in the same module. Also, imports and exports can be configured so that the user context information is propagated between modules. For SCA components other than mediation flow components, there is a ContextService API that enables access to the user context data.

IBM Software Group

Section

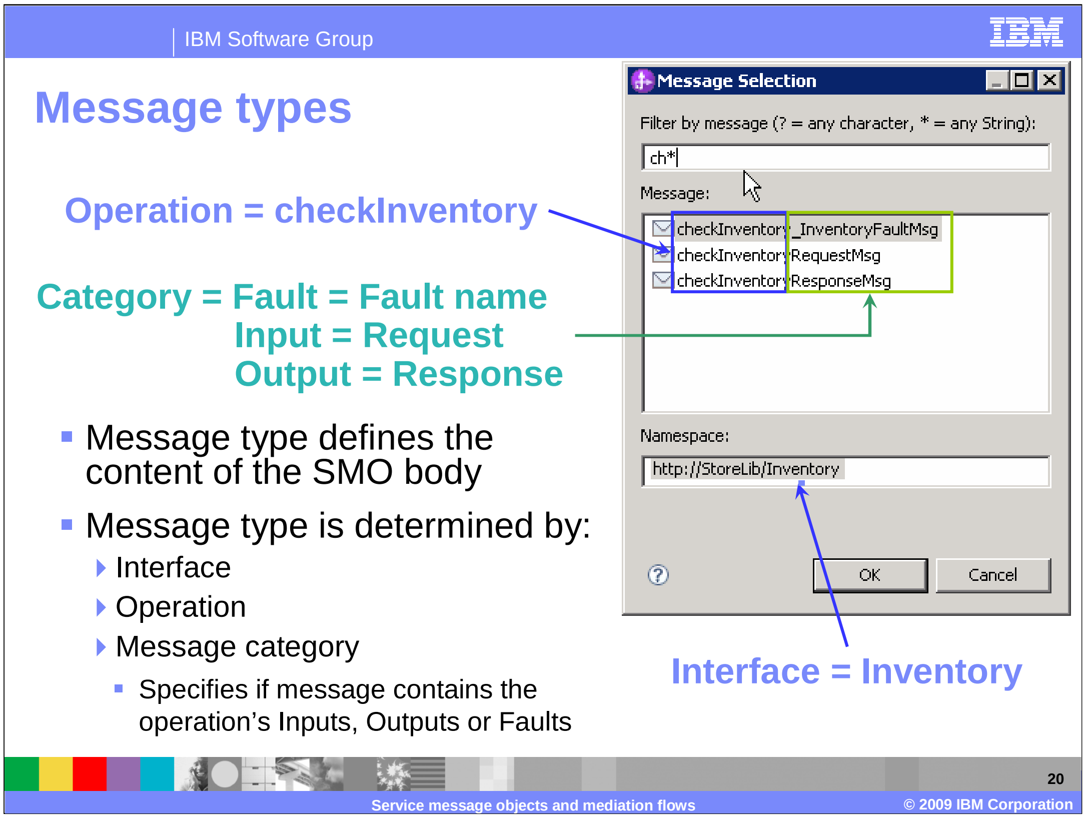
***Mediation flows
and the
service message object***

19

Service message objects and mediation flows

© 2009 IBM Corporation

This section describes the relationship between SMOs and mediation flows, in particular how message type plays a major role when defining a mediation flow.



Message types

Operation = checkInventory

Category = Fault = Fault name
Input = Request
Output = Response

- Message type defines the content of the SMO body
- Message type is determined by:
  - Interface
  - Operation
  - Message category
    - Specifies if message contains the operation’s Inputs, Outputs or Faults

Message Selection

Filter by message (? = any character, * = any String): ch*

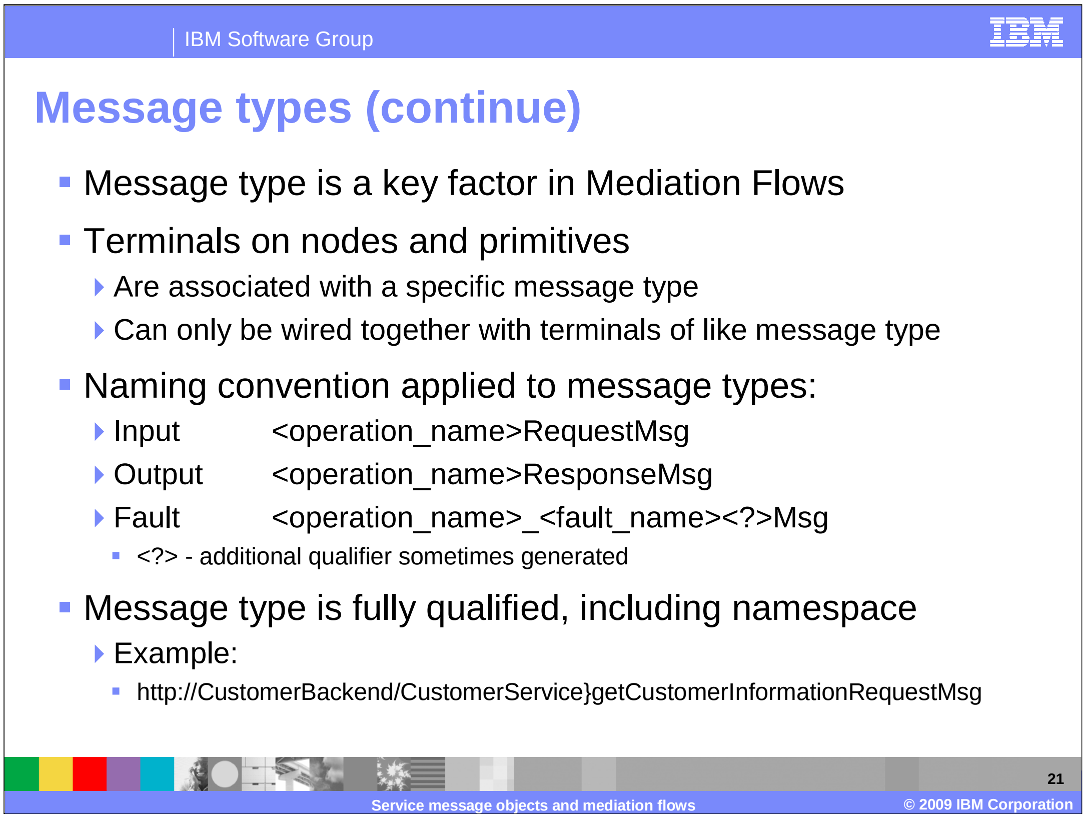
Message:

- checkInventory_InventoryFaultMsg
- checkInventoryRequestMsg
- checkInventoryResponseMsg

Namespace: http://StoreLib/Inventory

Interface = Inventory

A message type defines the structure of the body portion of the SMO. It is defined by the interface and the operation associated with the message, along with the message category. The message category indicates if the message contains the operation’s inputs, outputs or faults. The screen capture in this slide shows the Message Selection dialog, which you use to select the message type. The message type is composed of the operation name and category. It is qualified by the fully qualified interface name. This dialog displays all the available message types for the flow, and can be filtered to reduce the size of the list. For example, in the screen capture the filter is ch*, which reduced the list to only those message types starting with ch.



IBM Software Group

Message types (continue)

- Message type is a key factor in Mediation Flows
- Terminals on nodes and primitives
  - Are associated with a specific message type
  - Can only be wired together with terminals of like message type
- Naming convention applied to message types:
  - Input <operation_name>RequestMsg
  - Output <operation_name>ResponseMsg
  - Fault <operation_name>_<fault_name><?>Msg
    - <?> - additional qualifier sometimes generated
- Message type is fully qualified, including namespace
  - Example:
    - http://CustomerBackend/CustomerService}getCustomerInformationRequestMsg

21

Service message objects and mediation flows

© 2009 IBM Corporation

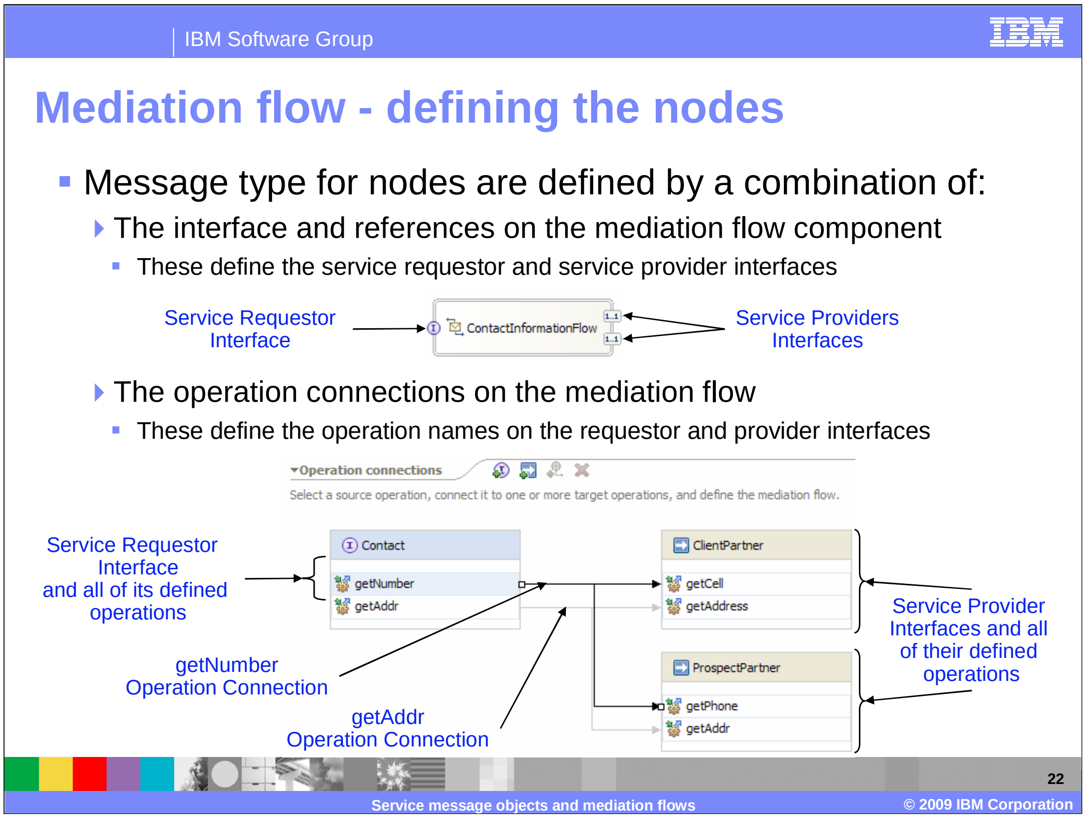
Message types are a key factor when defining a mediation flow. In a mediation flow the nodes and mediation primitives have terminals and each terminal is associated with a specific message type. When wiring a flow, only terminals of like message type can be wired together. There is a naming convention that is used for the definition of message types.

For an input message the convention is: *operation name*, RequestMsg.

For an output message the convention is: *operation name*, ResponseMsg.

The convention for a fault is: *operation name*, _*faultname*Msg. In this case, there sometimes is also a generated qualifier placed in between fault name and Msg. When a qualifier is generated. It can have one or more characters.

These naming conventions are actually the shortened form of the message type that appears in the mediation flow editor, whereas the real message type appears in the properties view. The real message type is a fully qualified name and includes both the namespace and interface as shown in the example above. This example shows a namespace of http://CustomerBackend, an interface of CustomerService, an operation of getCustomerInformation and it ends in RequestMsg to indicate this is for a request flow and contains the inputs.



IBM Software Group

Mediation flow - defining the nodes

- Message type for nodes are defined by a combination of:
  - The interface and references on the mediation flow component
    - These define the service requestor and service provider interfaces

  - The operation connections on the mediation flow
    - These define the operation names on the requestor and provider interfaces

Operation connections
Select a source operation, connect it to one or more target operations, and define the mediation flow.
Service Requestor Interface and all of its defined operations
Contact
getNumber
getAddr
ClientPartner
getCell
getAddress
ProspectPartner
getPhone
getAddr
Service Provider Interfaces and all of their defined operations
getNumber Operation Connection
getAddr Operation Connection

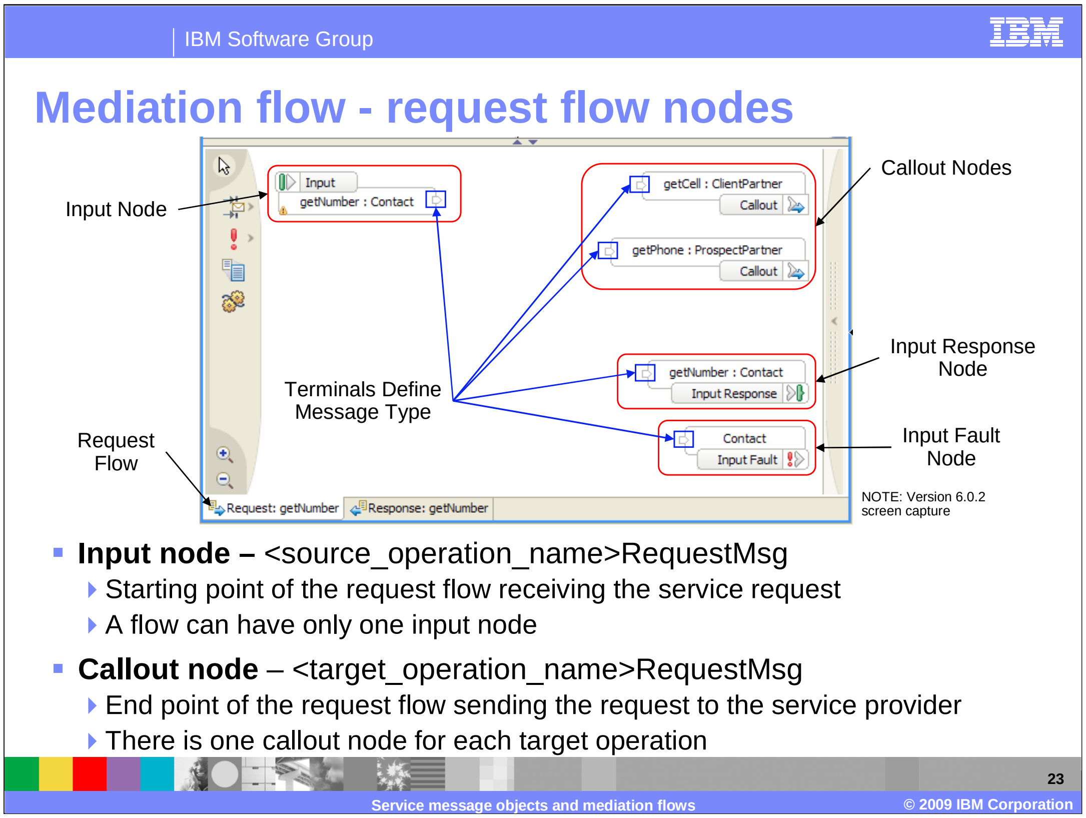
22

Service message objects and mediation flows

© 2009 IBM Corporation

The next several slides are used to show how a mediation flow is defined. The specific focus is on the message types associated with the terminals of the nodes and mediation primitives that make up the flow. Every mediation flow has nodes that represent the entry and exit points for the flow and the nodes have terminals that have fixed message types. The interfaces and operations associated with the flow determine which nodes are present in the flow and the message types associated with their terminals. It starts with the definition of the Mediation Flow Component in the assembly diagram as shown in the top of this slide. The Mediation Flow Component contains an interface that is used by a requestor and it also has references defining the interfaces used for calling providers. In the lower portion of the slide, the Operation Connections panel of the Mediation Flow Editor shows all of the operations associated with the defined interfaces. Using this panel, the operations on the input interface are connected to operations on the interfaces used to call providers.

Doing this provides sufficient information for any input operation to define the nodes for the flow, including the message types associated with the terminals for the nodes. This will be examined in detail on subsequent slides.



IBM Software Group

Mediation flow - request flow nodes

NOTE: Version 6.0.2 screen capture

- **Input node –** <source_operation_name>RequestMsg
  - Starting point of the request flow receiving the service request
  - A flow can have only one input node
- **Callout node** – <target_operation_name>RequestMsg
  - End point of the request flow sending the request to the service provider
  - There is one callout node for each target operation

23

Service message objects and mediation flows

© 2009 IBM Corporation

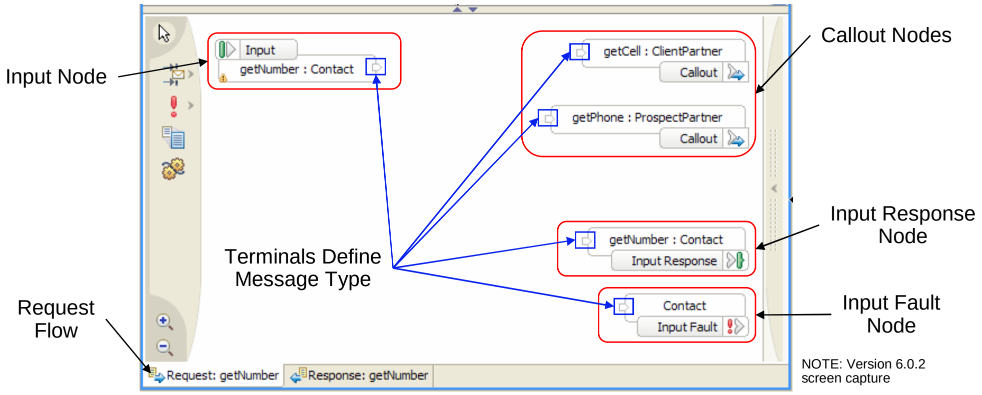
On this slide, and several of the next slides, there are screen captures taken from the mediation flow editor using WebSphere Integration Developer version 6.0.2. You might notice some differences between these screen captures and visual appearance when using the current version, but all the technical information is the same between the two versions.

This slide shows the canvas of the mediation flow editor for the request flow before the addition of any mediation primitives. Shown at the upper left of the canvas is the Input Node, which is the starting point for the request flow. There is only one input node for a mediation request flow and it has an output terminal with a message type of: source operation name, RequestMsg.

On the right side of the canvas, the top two nodes are the Callout Nodes. These are the end points for the request flow where a call is made to a service provider. There is one callout node for every target operation defined in the Operations Connections panel. The callout nodes each have an input terminal with a message type of: target operation name, RequestMsg.

The remaining nodes are described on the next slide.

Mediation flow - request flow nodes



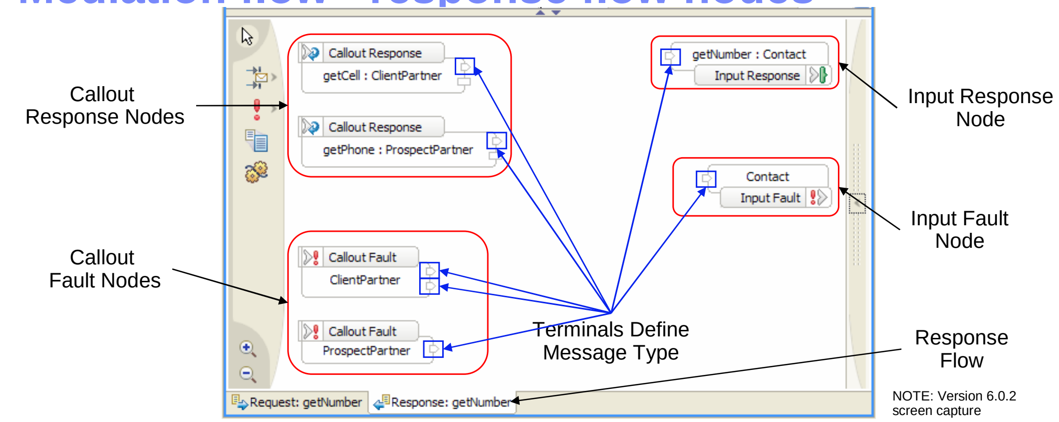
NOTE: Version 6.0.2 screen capture

- **Input response node –** <source_operation_name>ResponseMsg
  - Enables mediation flow to reply to requestor without calling a service provider
- **Input fault node –** <source_operation_name>_<fault_name><?>Msg
  - Enables mediation flow to return a WSDL fault message to the requestor without calling a service provider
  - Each fault defined for the source operation has its own terminal on this node

The third node on the lower right side is the Input Response Node. The Input Response Node enables the mediation flow to return directly to the requestor without calling a service provider and can be used where the mediation flow can satisfy the request. The input response node has an input terminal with a message type of: source operation name, ResponseMsg.

The bottom node on the right is the Input Fault Node, which enables the mediation flow to return a WSDL fault to the requestor and can be used when some error has been detected within the mediation flow. This node can have multiple input terminals, one for each of the faults defined on the source operation. The message type associated with each terminal is: source operation name, underbar, fault name, optional qualifier, Msg. If there are no faults defined for the source operation, the input fault node will not be present on the canvas.

Mediation flow - response flow nodes



Callout Response Nodes

Input Response Node

Input Fault Node

Callout Fault Nodes

Terminals Define Message Type

Response Flow

NOTE: Version 6.0.2 screen capture

- **Callout response node –** <target_operation_name>ResponseMsg
  - Starting point of the response flow receiving the response from the service provider
  - There is one callout response node for each target operation
- **Callout fault node** – <target_operation_name>_<fault_name><?>Msg
  - Starting point of the response flow receiving a WSDL fault message from the provider
  - Each fault defined for the target operation has its own terminal on this node

This slide shows the canvas of the mediation flow editor for the response flow before the addition of any mediation primitives.

Starting at the upper left of the canvas, there are two Callout Response Nodes, which are the starting points for the response flow where the return from the service provider is received. There is one callout response node for every target operation defined in the Operations Connections panel and they each have one output terminal with a message type of: target operation name, ResponseMsg. A Callout Response Node has another terminal which is used for unmodeled fault handling, the terminal type of which is beyond the scope of this discussion.

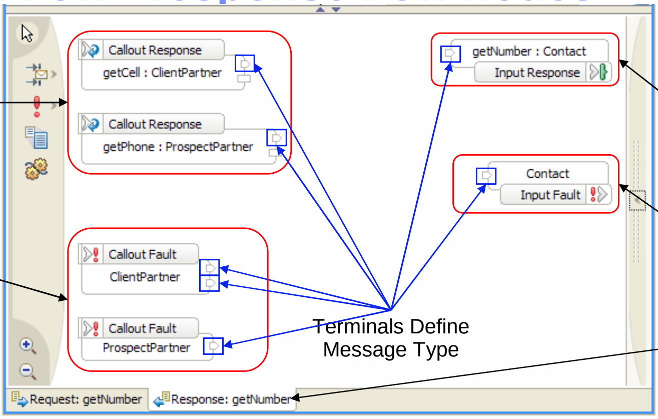
The lower two nodes on the left side are the Callout Fault Nodes, which are the starting points for the response flow when a service provider returns a fault. There is one callout fault node for every target operation that has one or more faults defined. These nodes may have multiple output terminals, one for each defined fault on the target operation. The message type for the terminals is: target operation name, underbar, fault name, optional qualifier, Msg.

The remaining nodes are described on the next slide.

IBM Software Group

IBM

Mediation flow - response flow nodes



Callout Response Nodes

Callout Fault Nodes

Terminals Define Message Type

Input Response Node

Input Fault Node

Response Flow

NOTE: Version 6.0.2 screen capture

- **Input response node –** <source_operation_name>ResponseMsg
  - End point of the response flow returning a response to the original requestor
  - A flow can have only one input response node
- **Input fault node –** <source_operation_name>_<fault_name><?>Msg
  - End point of the response flow returning a WSDL fault message to the original requestor
  - Each fault defined for the source operation has its own terminal on this node

26

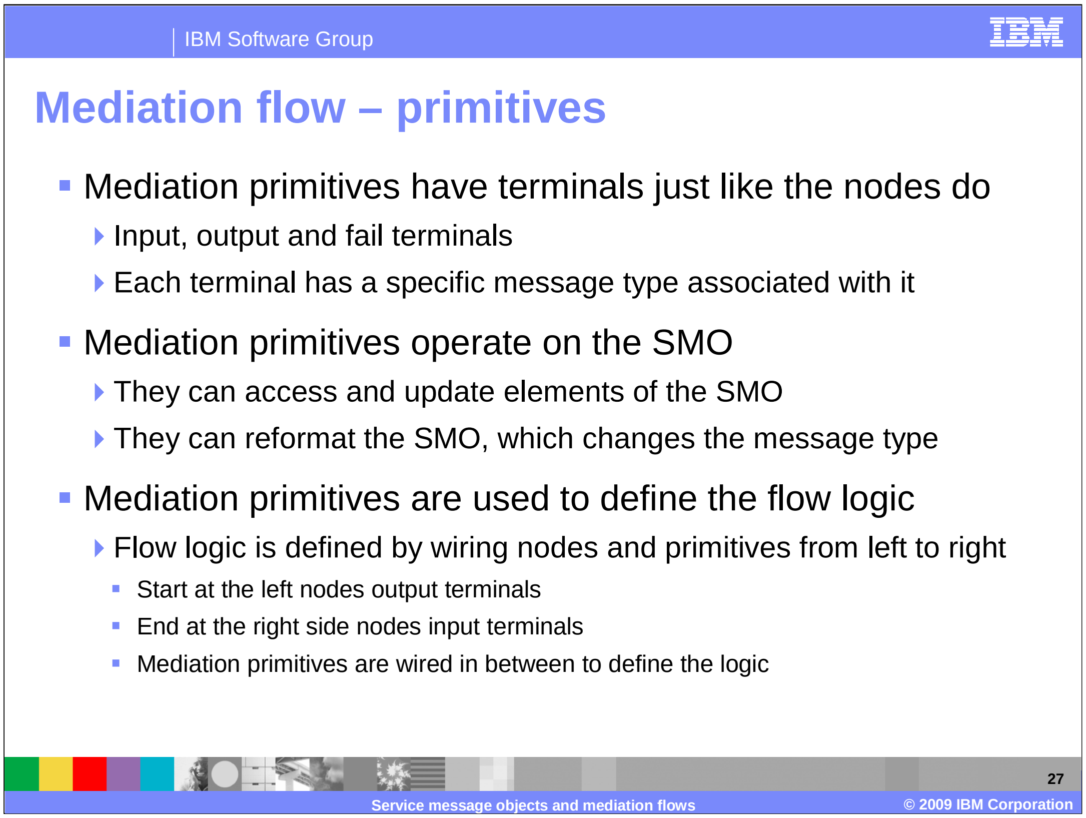
Service message objects and mediation flows

© 2009 IBM Corporation

On the right side of the canvas, the top node is the Input Response Node, the end point for the response flow, which returns to the original service requestor. There will be only one input response node in a response flow and the input terminal of this node has a message type of: source operation name, ResponseMsg.

The bottom node on the right is the Input Fault Node, which is used to return a WSDL fault to the original service requestor. There can be multiple input terminals on this node, one for each of the faults defined on the source operation. The message type associated with each terminal is: source operation name, underbar, fault name, optional qualifier, Msg. If there are no faults defined for the source operation, the input fault node will not be present on the canvas.

This completes an examination of the nodes, their terminals and associated message types.



IBM Software Group

Mediation flow – primitives

- Mediation primitives have terminals just like the nodes do
  - Input, output and fail terminals
  - Each terminal has a specific message type associated with it
- Mediation primitives operate on the SMO
  - They can access and update elements of the SMO
  - They can reformat the SMO, which changes the message type
- Mediation primitives are used to define the flow logic
  - Flow logic is defined by wiring nodes and primitives from left to right
    - Start at the left nodes output terminals
    - End at the right side nodes input terminals
    - Mediation primitives are wired in between to define the logic

27

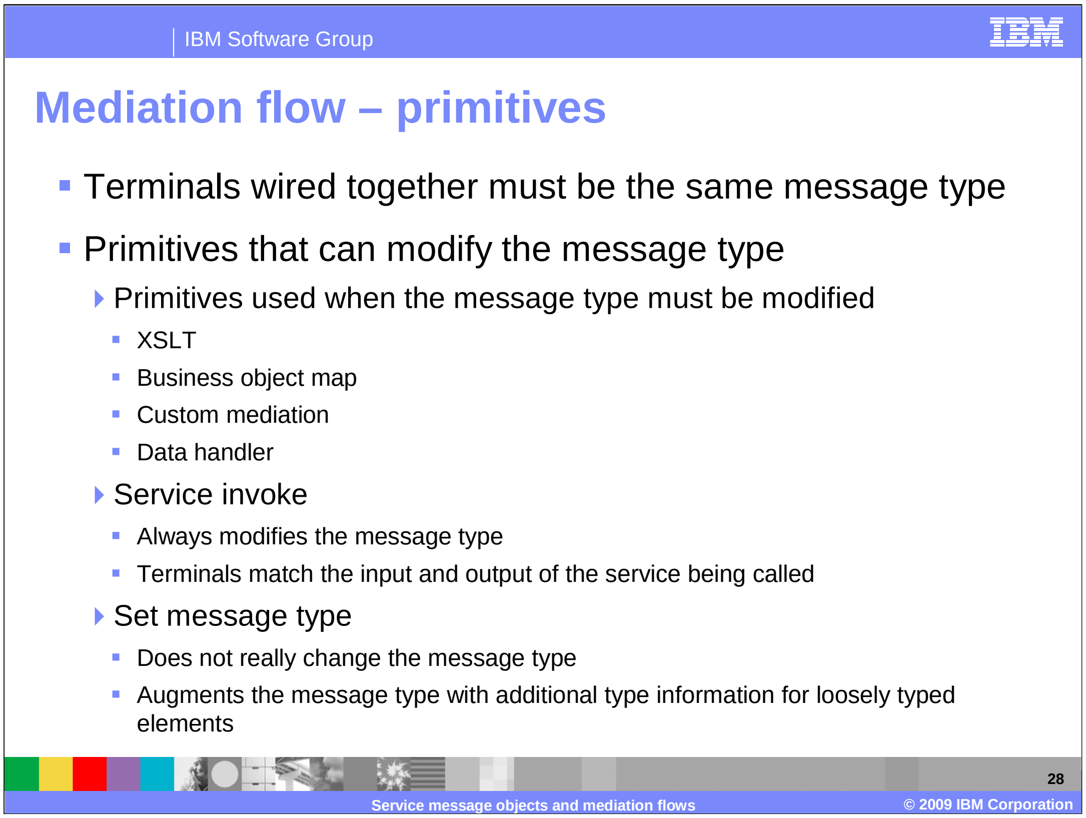
Service message objects and mediation flows

© 2009 IBM Corporation

Now that nodes have been covered, including their terminals and associated message types, it is time to consider mediation primitives and how they are used to define the logic of a mediation flow. Similar to nodes, the mediation primitives also have terminals. An SMO is passed to a mediation primitive through the input terminal and is passed out of the primitive through an output or fail terminal. Each of these terminals has a specific message type associated with it.

The mediation primitives operate on the SMO. Some primitives can only access element values from the SMO, others can access and update values and some can also reformat the SMO. Reformatting of the SMO changes the message type.

The flow logic of the mediation is defined by adding mediation primitives to the canvas between the nodes and then wiring the nodes and mediation primitives together. The flow is defined from left to right, starting with the left side nodes, wiring through some combination of mediation primitives and ending with the right side nodes.



IBM Software Group

Mediation flow – primitives

- Terminals wired together must be the same message type
- Primitives that can modify the message type
  - Primitives used when the message type must be modified
    - XSLT
    - Business object map
    - Custom mediation
    - Data handler
  - Service invoke
    - Always modifies the message type
    - Terminals match the input and output of the service being called
  - Set message type
    - Does not really change the message type
    - Augments the message type with additional type information for loosely typed elements

28

Service message objects and mediation flows

© 2009 IBM Corporation

When wiring together the terminals for the nodes and primitives, any terminals that are wired together must be for the same message type. This is because the format of the SMO as it flows out of one terminal is unchanged as it flows into the terminal it is wired to.

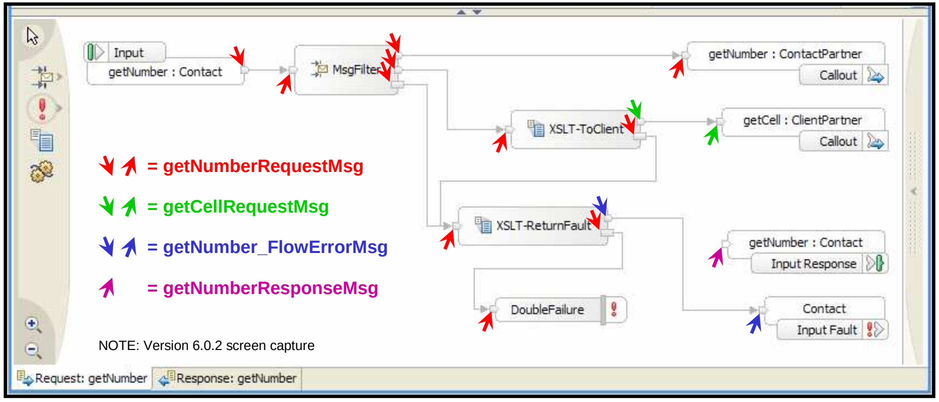
When there is a need to connect nodes or primitives having terminals with differing message types, the XSLT, business object map, custom mediation, or data handler primitive must be used between them. This allows the SMO to be reformatted to the other message type based on a mapping you define.

Another primitive that modifies the message type is the service invoke. The input terminal for the service invoke is the request message type for the operation being invoked on the service. The output terminal is the response message type.

Finally, there is the set message type primitive. The name of this primitive is somewhat misleading in that it does not really change the message type of the SMO that flows through this primitive at runtime. It is used to declare a more specific type for a loosely typed field in the SMO, such as an anyType. This allows development time tools to understand the more specific type, making it easier for you to use the XPath expression builder and the mapping tools in WebSphere Integration Developer. A good analogy for this functionality is a cast operation in a programming language. This message type augmentation does affect the wiring capabilities, which is explained in detail in the presentation for this primitive.

The next slide will provide an example mediation flow and explain the various terminal message types.

Mediation flow definition - example



↓↑ = getNumberRequestMsg

↓↑ = getCellRequestMsg

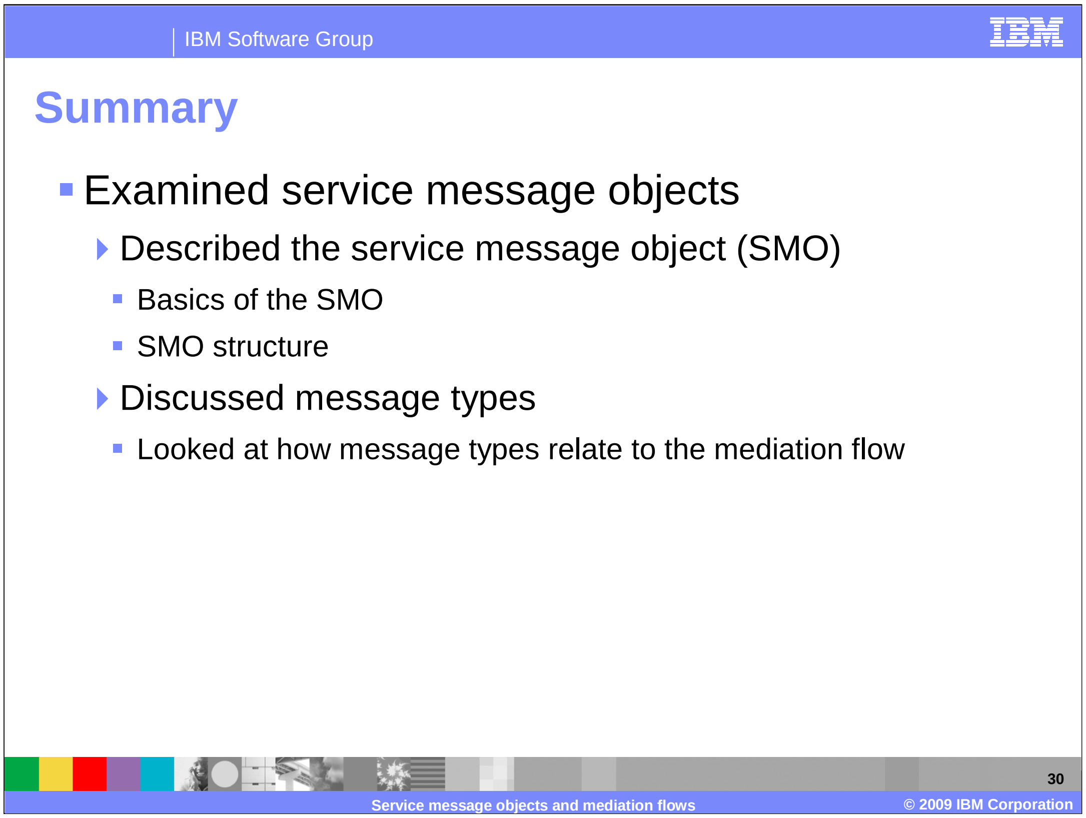
↓↑ = getNumber_FlowErrorMsg

↑ = getNumberResponseMsg

NOTE: Version 6.0.2 screen capture

- Two possible providers, one with a different interface than the requestor
- Errors in the flow result in a fault being returned to the requestor
- XSLT primitives used to modify message type when required
- Message types of terminal identified with color coded arrows

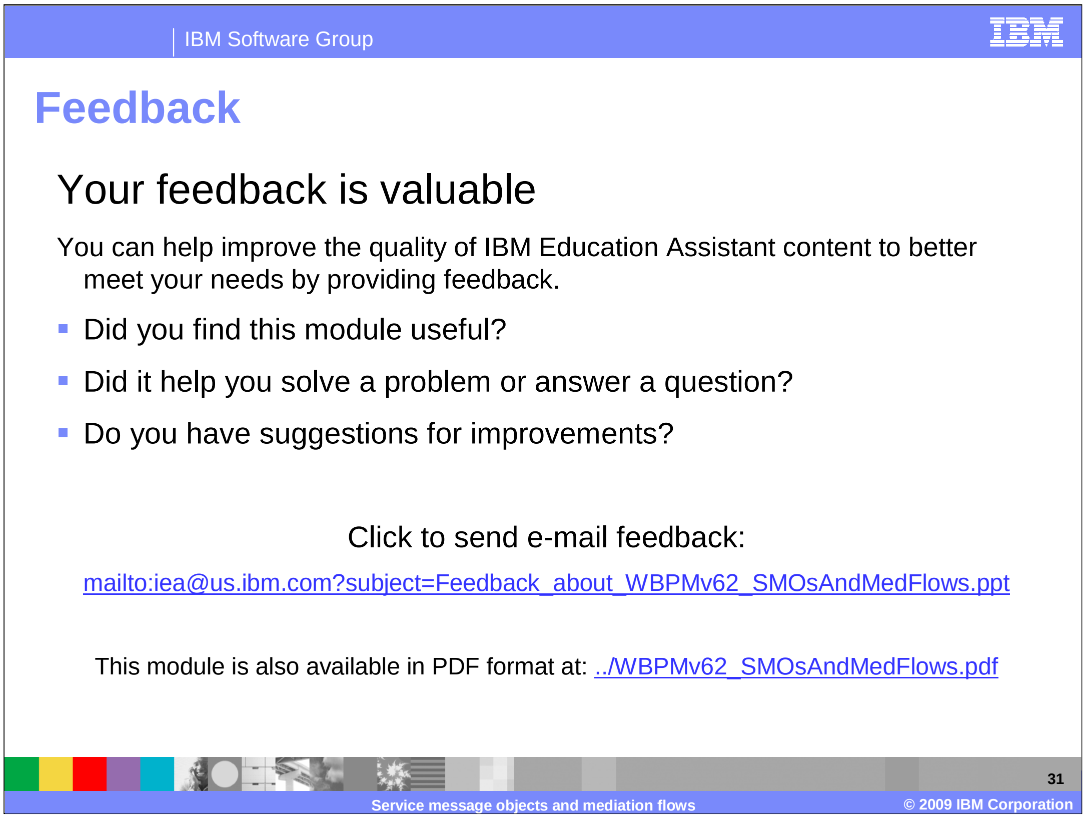
This slide contains a realistic example of a mediation flow, illustrating the wiring of nodes and mediation primitives together, while taking into account the constraint of only being able to wire terminals of like message type. This example shows two possible target service providers, one of which has a different interface than the service requestor. The flow also handles errors and returns a fault to the requestor if there is a failure in the flow. The flow in the upper left shows that the Input node is for the getNumber operation of the Contact interface. Therefore, it has an output terminal with a message type of getNumberRequestMsg. Looking at the flow in the upper right, the top Callout node is for the same interface and operation as the Input node and therefore has an input terminal of the same message type. The other Callout node is for the getCell operation of the Client interface and it therefore has an input terminal with a message type of getCellRequestMsg. Continuing down the right side, the Input Response node's input terminal will be for message type getNumberResponseMsg. Finally, on the lower right is an Input Fault node with an input terminal for message type getNumber_FlowErrorMsg, corresponding to the FlowError fault defined in the getNumber operation of the requestor. Before examining the flow, notice that each terminal in the flow is marked with an arrow of a different color, with each color representing the message type of the terminal it is pointing to. In the flow, the Input Node is wired to a Message Filter mediation primitive. This primitive contains some logic that differentiates between requests that should be passed to the provider with the Contact interface versus requests that should be passed to the provider with the Client interface. As you can see, all terminal message types for this primitive are for the getNumberRequestMsg. In the case where the request goes to the provider with the Contact interface, the wire can go directly to the Callout node. In the case where the request goes to the provider with the Client interface, the message type must be changed from a getNumberRequestMsg to a getCellRequestMsg. This is done using the XSLT primitive that is labeled XSLT-ToClient and which is then wired to the Callout. The description of the non-error paths through the flow is now complete and the error paths can now be examined. Coming out of the Message Filter and XSLT primitives are Fail terminals which are used when the primitive raises some kind of an error. Fail terminals always have the same message type as the input terminal, so both of these have a type of getNumberRequestMsg. The flow logic is designed to return a fault to the requestor when either of these primitives fails. In order to do this, both Fail terminals are wired to the primitive labeled XSLT-ReturnFault, which changes the message type from a getNumberRequestMsg to a getNumber_FlowErrorMsg. It is then wired to the Input Fault node which returns the fault to the requestor. Finally, there is a possibility that the XSLT used to modify the SMO to a fault message can fail. In this case, the Fail terminal of the XSLT-ReturnFault primitive is wired to a Fail primitive. This results in the mediation flow ending in an exception with no response returned to the requestor.



Summary

- Examined service message objects
  - Described the service message object (SMO)
    - Basics of the SMO
    - SMO structure
  - Discussed message types
    - Looked at how message types relate to the mediation flow

In summary, this presentation examined the use of Service Message Objects by first describing what an SMO is and how it is structured. The concept of message types was explained and a detailed description of how message types affect the construction of a mediation flow was given. Finally, an example of a mediation flow was provided, illustrating how the message type affects the wiring of the flow.



Feedback

Your feedback is valuable

You can help improve the quality of IBM Education Assistant content to better meet your needs by providing feedback.

- Did you find this module useful?
- Did it help you solve a problem or answer a question?
- Do you have suggestions for improvements?

Click to send e-mail feedback:

mailto:iea@us.ibm.com?subject=Feedback_about_WBPMv62_SMOsAndMedFlows.ppt

This module is also available in PDF format at: ../WBPMv62_SMOsAndMedFlows.pdf

You can help improve the quality of IBM Education Assistant content by providing feedback on this module.

Trademarks, copyrights, and disclaimers

IBM, the IBM logo, ibm.com, and the following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

WebSphere

If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of other IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml

J2EE, and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Product data has been reviewed for accuracy as of the date of initial publication. Product data is subject to change without notice. This document could include technical inaccuracies or typographical errors. IBM may make improvements or changes in the products or programs described herein at any time without notice. Any statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business. Any reference to an IBM Program Product in this document is not intended to state or imply that only that program product may be used. Any functionally equivalent program, that does not infringe IBM's intellectual property rights, may be used instead.

THE INFORMATION PROVIDED IN THIS DOCUMENT IS DISTRIBUTED "AS IS" WITHOUT ANY WARRANTY, EITHER EXPRESS OR IMPLIED. IBM EXPRESSLY DISCLAIMS ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT. IBM shall have no responsibility to update this information. IBM products are warranted, if at all, according to the terms and conditions of the agreements (for example, IBM Customer Agreement, Statement of Limited Warranty, International Program License Agreement, etc.) under which they are provided. Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products in connection with this publication and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products.

IBM makes no representations or warranties, express or implied, regarding non-IBM products and services.

The provision of the information contained herein is not intended to, and does not, grant any right or license under any IBM patents or copyrights. Inquiries regarding patent or copyright licenses should be made, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

Note to U.S. Government Users - Documentation related to restricted rights-Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract and IBM Corp.