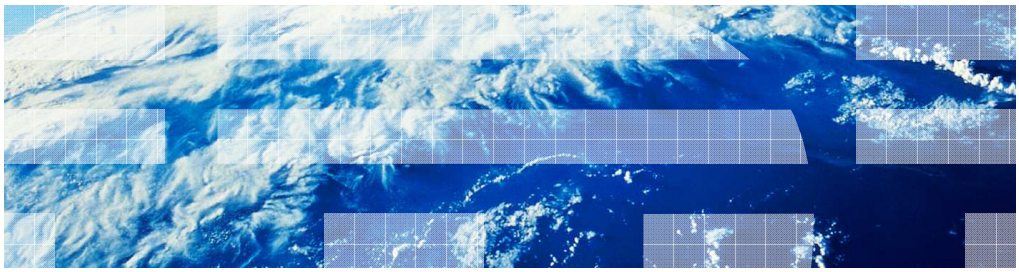
WebSphere Business Process Management

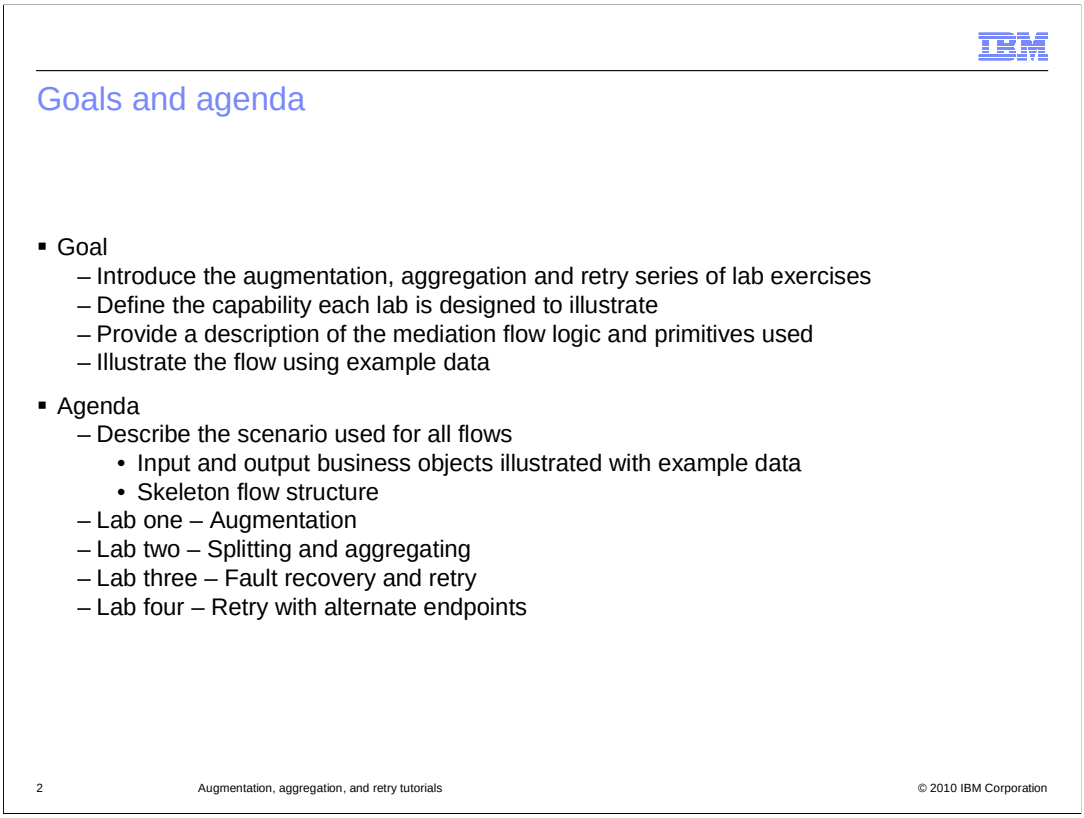
WebSphere Integration Developer
WebSphere Enterprise Service Bus
WebSphere Process Server

Augmentation, aggregation, and retry tutorials



WebSphere. software

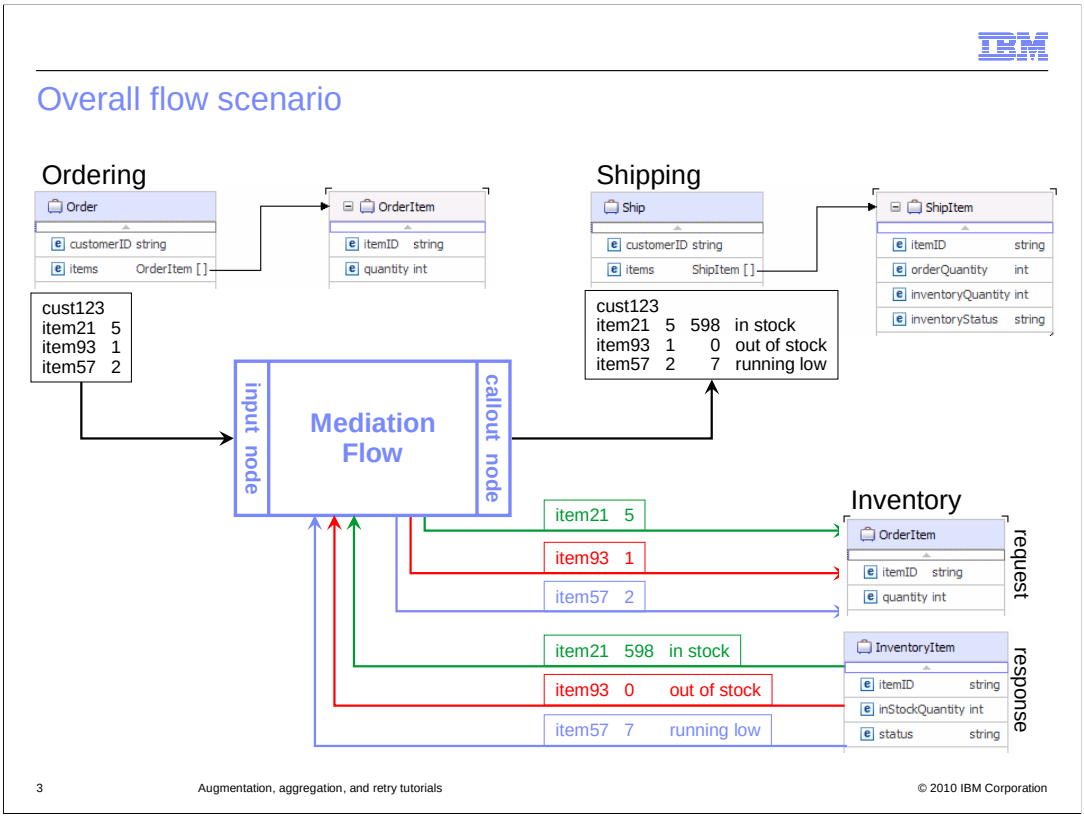
This presentation introduces a series of tutorials, or lab exercises, designed to illustrate the mediation flow programming model for message augmentation, message splitting and aggregation, and the use of service invoke retry.



The goal of this presentation is to introduce to you a series of lab exercises designed to illustrate elements of the mediation flow programming model. These include the use of the service invoke primitive to perform augmentation of data in a message. Also, the use of the fan out and fan in primitives to do message splitting and aggregation, enabling the processing of an array of elements within a message. Finally, the use of service invoke retry, which performs automatic retry logic when a service call returns a fault. In the presentation, there is a description of the capability that each of the lab exercises is designed to illustrate and how that is realized in the mediation flow logic. Example data is used to illustrate what the flow actually does.

The presentation starts out by describing the overall scenario that is used in all the lab exercises. This is done by looking at the business objects used on input and output along with example data. The basic flow, or skeleton, used in the construction of each lab exercise is described. Therefore, it does not have to be repeated in the description of each of the lab exercises.

There are a total of four labs, the first addressing message augmentation for a message with a single element. The next lab looks at how splitting and aggregating is done for augmenting a message with an array of repeating elements. The third looks at recovery from a fault, both in flow logic and with the use of service call retry. Finally, the fourth lab addresses the use of alternate endpoints when performing service call retry.

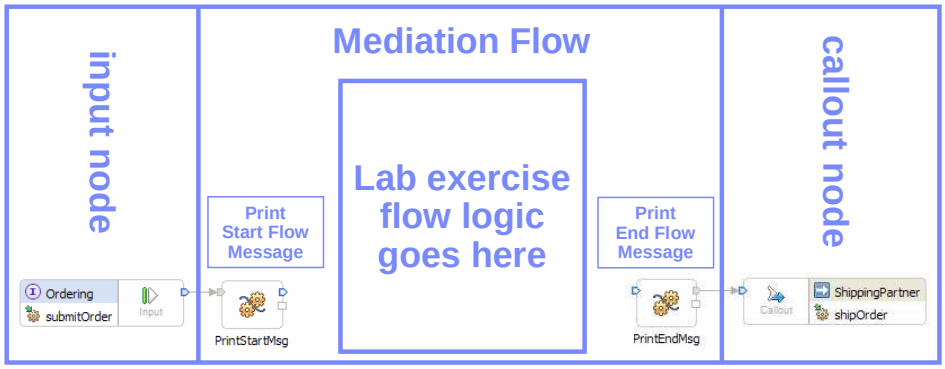


This slide provides you with a description of the overall scenario used in this series of labs. There is a mediation flow which takes as input a one way operation from an ordering system as shown in the upper left. It receives an Order business object instance containing a customerID field and an array of OrderItem business objects, each containing an itemID and quantity. You can see the business object definitions and example data on the slide. During the mediation, calls are made to an inventory service, one call for each item. The request passes an OrderItem as input and the response is an InventoryItem containing the itemID, the inStockQuantity and inventory status. This can be seen on the bottom of the slide. Finally, the mediation calls out to a shipping service in a one way operation, passing a Ship business object, as seen in the upper right of the slide. The Ship object is composed of a customerID and an array of ShipItem business objects, each containing the itemID, the orderQuantity, the inventoryQuantity and the inventoryStatus.

Each of the labs in this series uses a variation of this basic scenario.

Skeleton structure of the flow

- Lab exercises use the same input node and callout node
- The flows start and end with primitives that write to the console (SystemOut.log)
  - The flow logic for all lab exercises goes between these primitives
  - Screen captures in this presentation only show the lab exercise flow logic



Before looking at the individual labs, this slide introduces to you the basic skeleton flow. The flow logic for each lab exercise is added within this skeleton. The ordering and shipping services are the same throughout the labs, so that the endpoints of the flow are always the same and behave the same. Therefore, there are no changes to the input node or callout node. The lab exercises make use of writing to the console view containing the SystemOut.log file as a way of illustrating what is happening in the overall processing. The skeleton flow contains a primitive immediately following the input node which writes to the console indicating the start of the flow. Likewise, there is a primitive just before the callout node which writes the end of flow message. The flow logic for all of the labs is contained between the two primitives used to write to the console. It is only this intervening logic that is shown in the illustrations that follow in this presentation.

Lab one – Augmentation

- Key illustration
  - Use of service invoke primitive for message augmentation
  - Requires use of XSL transformation and transient context
- Primitive usage
  - Order2Inventory XSL transformation
    - Modify body from order request message to inventory request message
    - Save order data in transient context
  - CheckInventory service invoke
    - Call inventory service to check item status
    - The results returned as an inventory response message
  - Inventory2Ship XSL transformation
    - Modify body from inventory response message to ship request message
    - Inventory data moved from body to body
    - Order data moved from transient context to body

5 Augmentation, aggregation, and retry tutorials

The key purpose of lab one is to illustrate the use of the service invoke primitive. Calling out to a service from within a mediation flow can have many useful purposes. In this example it is used to obtain additional data that is then used to augment the contents of the message as it flows through the mediation. To perform a message augmentation scenario using a service invoke primitive, there is a need to transform the message to and from the message type used by the service invoke. This requires a primitive, such as the XSL transformation, that can modify the message type. It also typically requires use of the transient context to save information from the input message body that is needed by the callout message.

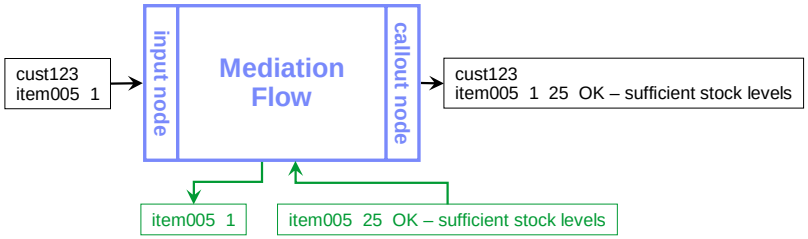
The logic for this flow requires three primitives as can be seen in the screen capture. The first is an XSL transformation, called Order2Inventory, whose primary purpose is to transform the message body from the order request message received by the mediation to an inventory request message needed to call the inventory service. It also places data into the transient context that needs to be saved across the call to the inventory service.

The second primitive is a service invoke, called CheckInventory, which makes a call to the inventory service which is external to the mediation flow. It passes information about the item being ordered and returns inventory information about that item.

The last primitive is another XSL transformation, called Inventory2Ship. Its purpose is to modify the message body from the inventory response message, returned by CheckInventory, into a ship request message, which is required by the callout. The body of the ship request message is created with data from two sources. Inventory information is taken from the body of the inventory response message and order information is taken from the saved data in the transient context.

Lab one – Augmentation

- Key assumption
  - Input array contains exactly one item
  - Array references are all indexed → [1]

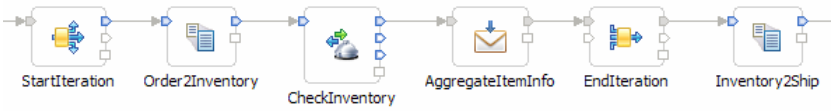


6 Augmentation, aggregation, and retry tutorials

This lab is designed to illustrate the augmentation of a message containing a single item. However, in order to maintain the identical input and output definitions throughout this series of lab exercises, there are arrays of items defined in the input and output business objects. Therefore, for this lab, there is a key assumption made in the construction of the flow. Even though the input from order and the output to shipping are defined to have an array of items, this flow assumes that there is exactly one item in the input array containing the order. This requires, throughout the flow, using an index of one when dealing with the item arrays.

The illustration in the lower part of the slide shows the data flow associated with this mediation. The data coming from the order system is seen on the left. On the bottom of the illustration you see the data for the request and response with the inventory service. Finally, on the right, is the message to the shipping service. You can see that only one item is being processed and you can also see the augmentation of the data sent to the shipping service.

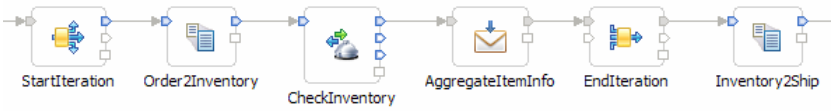
Lab two – Splitting and aggregating



- Key illustration
  - Use of fan out and fan in for splitting and aggregating
    - Enabling augmentation of multiple array elements
  - Requires:
    - Augmentation scenario elements - service invoke, XSL transformation and transient context
    - Use of fan out context containing array element for each iteration
    - Use of shared context for aggregation of results
    - Use of message element setter append option for building up array

The key purpose of lab two is to illustrate the use of the fan out and fan in primitives to perform a splitting and aggregating scenario. Basically, this takes the previous augmentation scenario and enables the augmentation of multiple items in the message. Looking at the screen capture, you can see that the Order2Inventory, CheckInventory and Inventory2Ship primitives used in the previous lab are still part of the flow. To accomplish this scenario there are some additional requirements. The fan out context, which is initialized by the fan out primitive, is used to obtain the current item during each iteration. The shared context, which is a shared memory area, must be used to accumulate the results of each iteration. The message element setter append option is needed to build up the array in the shared context.

Lab two – Splitting and aggregating



- Primitive usage
  - StartIteration fan out
    - Configured to iterate over item array in input message
    - SMO cloned at the start of each iteration
    - Fan out context set with current element at the start of each iteration
  - Order2Inventory XSL transformation (different map from lab one)
    - Modify body from order request message to inventory request message
    - Save order data in transient context
      - Customer data moved from the body
      - Item information moved from the fan out context
  - CheckInventory service invoke (same as lab one)
    - Call inventory service to check item status
    - The results returned as an inventory response message

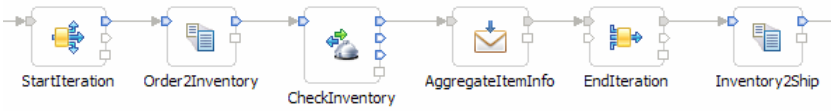
8 Augmentation, aggregation, and retry tutorials

The best way to understand this flow is to walk through it to see what each primitive does in the flow. The first primitive is the StartIteration fan out primitive which is configured to iterate over the item array in the order request message. This primitive serves as the top of a loop, which is used to process the individual items in the flow. As each iteration is begun, the fan out primitive clones the SMO and sets the value of the current item into the fan out context. As a result of this, the SMO used to start each iteration is almost the same as the SMO in every other iteration. The only differences are that each SMO is in a separate memory area

The next primitive is the Order2Inventory XSL transformation, which has the same purpose as its counterpart in the previous lab, but contains a different mapping. As in the previous lab, it modifies the body of the message from an order request message to an inventory request message, in preparation for calling the inventory service. Also, information about the order is saved in the transient context. The customer data comes from the body of the order message, however the item information comes from the fan out context. The order information saved in the transient context is placed into a ShipItem structure in preparation for being combined with inventory information.

The next primitive, CheckInventory, is unchanged from the previous lab and is used to call the inventory service.

Lab two – Splitting and aggregating



- Primitive usage (continue)
  - AggregateItemInfo message element setter
    - Inventory data moved from body to transient context (combining it with order data)
    - Combined order and inventory data for this item taken from transient context and appended to array in shared context
  - EndIteration fan in
    - Configured to complete when input array fully processed
    - Flow returns to StartIteration fan out if not complete
  - Inventory2Ship XSL transformation (different map from lab one)
    - Modify body from inventory response message to ship request message
    - Customer information moved from transient context to body
    - Completed item array with order and inventory information moved from shared context to body

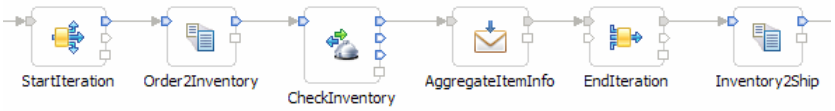
9 Augmentation, aggregation, and retry tutorials

The next primitive in the flow is a message element setter named AggregateItemInfo. It takes inventory information about the item and moves it from the body of the inventory response message to the transient context. It is placed into the ShipItem structure in the transient context, thus combining it with the order data for the item which was previously placed there. The primitive is then used to move the completed ShipItem from the transient context and append it to the array of ShipItem elements in the shared context. The shared context is in a memory area shared by all of the SMO clones, and therefore this provides a mechanism for accumulating the results of each iteration.

The next primitive is the fan in, called EndIteration. This primitive serves as the end of the iterative loop. It has configuration information defining its completion criteria, indicating that it completes when an entire item array has been processed. If the item array processing is not complete, the flow returns to the StartIteration fan out for the next iteration. If the item array processing is complete, the flow continues following the fan in.

When the fan in completes, the flow proceeds to the Inventory2Ship XSL transformation. Similar to the previous lab, this primitive modifies the message so that it can be sent to the shipping service. However, the actual mapping required is different. The customer information is moved to the body from the transient context, where it was saved at the beginning of the flow. The array of ship items, which as been built up in the shared context during the iterative flow, is moved from the shared context to the body.

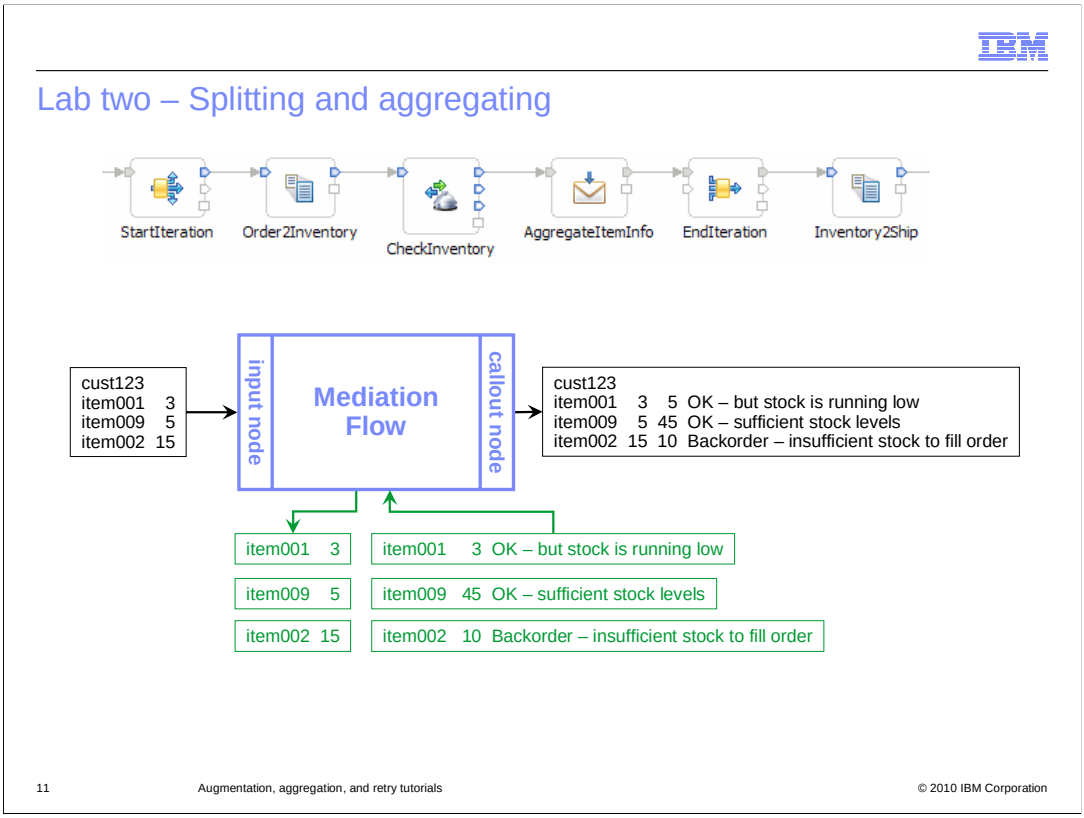
Lab two – Splitting and aggregating – Best practices



- Best practices for handling of array elements in aggregation
  - Target array element type is used throughout the flow
    - Use this as a guide to design the transient and shared contexts you use
  - Array type in shared context is identical to the target array type
  - Have single element in the transient context
  - Build up single element during an iteration
    - Use whatever primitives are needed for your scenario
    - For example, in this scenario an XSL transformation and message element setter
  - Append single element to shared context array at end of an iteration
  - Move shared context array to message body array when all iterations complete

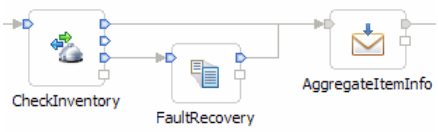
This slide is a tangent to the specific discussion of the lab exercises. However, it is inserted here to highlight key factors about the processing done in this lab exercise. These points are worth mentioning as they are considered best practices for implementation of an aggregation scenario.

The scenario assumes there is a target array of repeating elements that is being constructed during the aggregation. You should use the array element type of the target array when defining your shared and transient contexts for use in the flow. The array defined in the shared context should be the same type as the target array, composed of the same element types. In the transient context, define a single element of the type contained in the array. During the iterative flow, build up the contents of the single element in the transient context, using whatever primitives are needed for your particular scenario. For example, in this scenario, an XSL transformation was used to move in the order information and a message element setter was used to move in the inventory information. At the end of each iteration, use a message element setter to append the single element built in the transient context to the end of the array being built up in the shared context. When the entire input array has been processed, move the array built up in the shared context to the message body.



This slide shows an example of the processing that occurs as a result of this flow, assuming the input order contains three items. This is shown on the left. The three inventory service requests and responses, one for each item, can be seen on the bottom. The callout to the shipping service, with augmented data for all three items, is on the right.

Lab three (first section) – Fault recovery



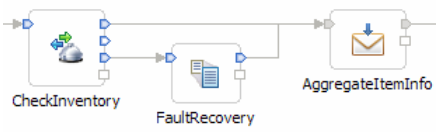
- Key illustration
  - Recover from a fault returned to the flow
    - Allows full array to be processed regardless of the fault returned from the inventory service for one item
- Primitive usage
  - Same flow as lab two with addition of one primitive
  - FaultRecovery XSL transformation
    - Wired between the CheckInventory service invoke primitive's terminal for InventoryFault and the AggregateItemInfo message element setter
    - Converts message from inventory fault message to inventory response message
    - Sets inventory quantity to 0 and inventory status to indicate an error occurred
    - Other than actual inventory data values the entry to AggregateItemInfo is the same as if the call worked

The third lab is broken into two sections, the first for fault recovery and the second illustrates service call retry.

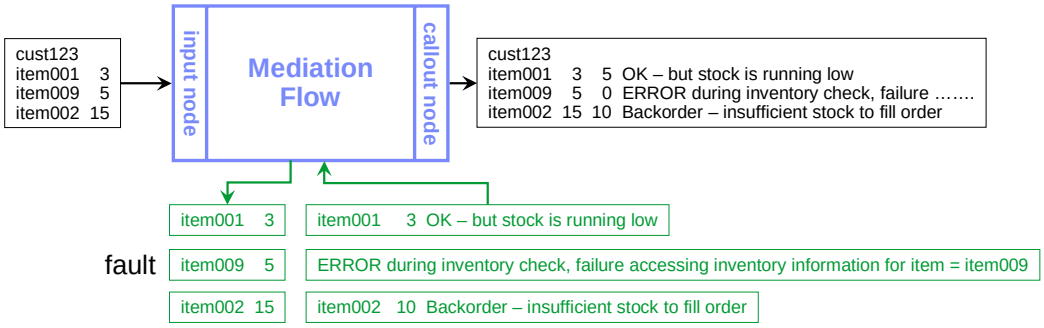
This first section describes fault recovery in the flow, allowing the flow to continue rather than terminate when a fault is returned from the inventory service. This enables the entire list of items to be passed to the shipping service, but without inventory information for the item being processed when the fault occurred.

To do this the same flow is used as lab two, with the addition of one primitive, the FaultRecovery XSL transformation. The CheckInventory service invoke primitive has an InventoryFault terminal that is wired to this XSL transformation. It converts the SMO from an inventory fault message to an inventory response message. The inventory quantity is explicitly set to zero and the inventory status is set to indicate that an error occurred calling the inventory service. By performing this transformation, the SMO content is the same as if the inventory service call worked, with the exception of the actual data values for inventory quantity and status. The FaultRecovery XSL transformation is then wired to the AggregateItemInfo and processing proceeds as if no fault occurred.

Lab three (first section) – Fault recovery



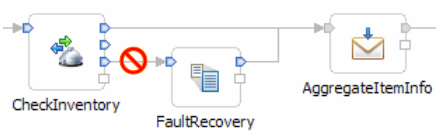
- Inventory service
  - Implementation randomly works or returns a fault
  - Because random, your output will vary



13 Augmentation, aggregation, and retry tutorials

When doing this part of the lab exercise, the endpoint for the inventory service is changed to an implementation that is designed so that it sometimes works and sometimes returns a fault. Because the results are random, the output you see when testing can be different than what is illustrated in this example. In the example, the input is the same three items used in the previous lab. Notice that the call to the inventory service for the second item, item009, returned a fault. The output sent to the shipping service shows all the items in the list. However, the inventory information for item009 indicates that there was a problem calling the inventory service and the inventory quantity is zero.

Lab three (second section) – Retry

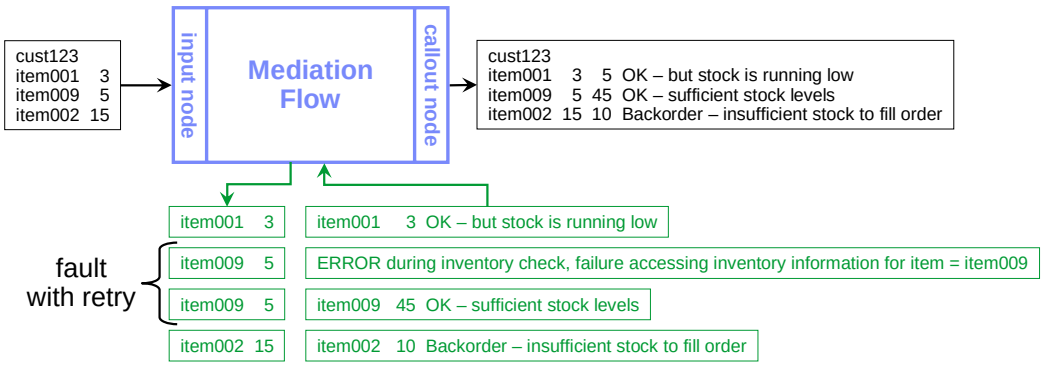


- Key illustration
  - Service invoke automatically retries when fault returned
    - Reduces chance of fault actually being returned to the flow
- Primitive usage
  - No changes to flow
  - CheckInventory service invoke primitive
    - Configured for retry on modeled fault with retry up to three times

The second section of lab three illustrates the use of the service call retry capabilities of the service invoke primitive. By performing a retry automatically, the likelihood of the service invoke primitive having to return a fault is reduced because the retry might be successful. To do this, there are no changes made to the flow other than the configuration for the CheckInventory service invoke primitive. It is modified so that it performs a retry on the occurrence of a modeled fault, performing up to three retries. If any retry is successful, the flow continues. If not, the fault returned on the third retry is passed to the flow.

Lab three (second section) – Retry

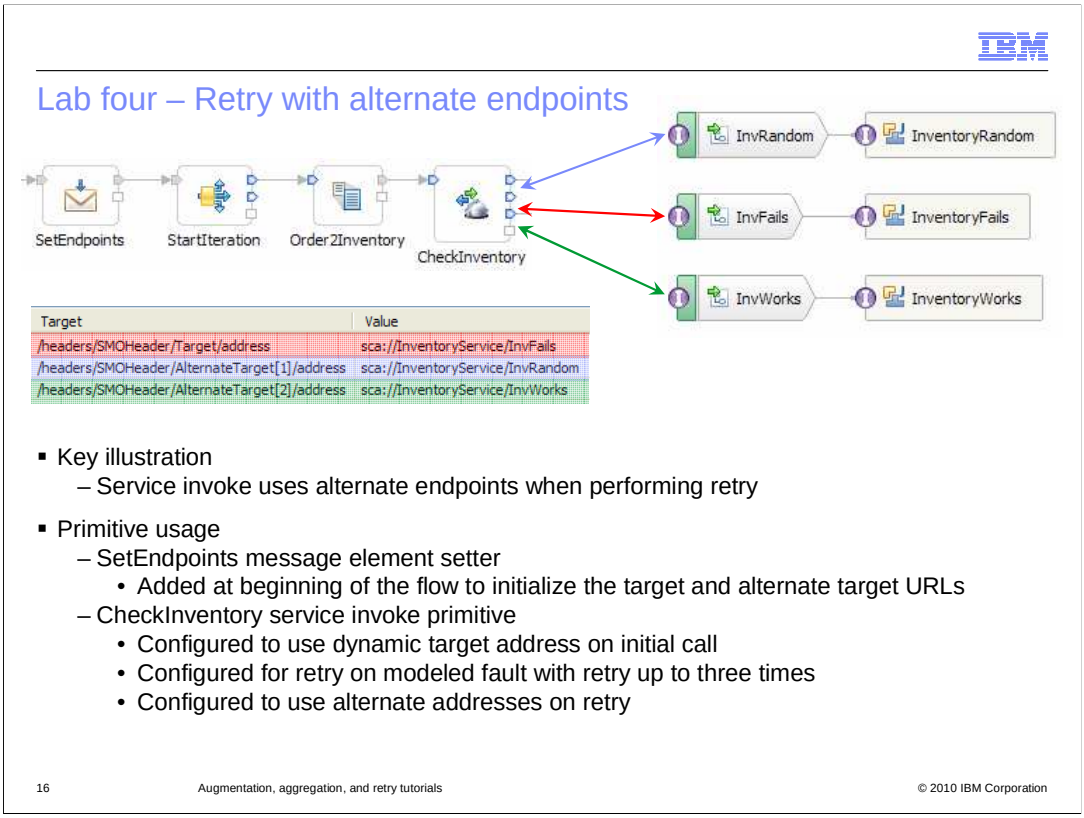
- Expected behavior in this lab exercise
  - Inventory service randomly works or returns a fault
  - Inventory service will not return more than two sequential faults
  - Inventory service will not work more than two time sequentially
  - Retry count on service invoke is set to three (result is the fault terminal is never used in this lab)



15 Augmentation, aggregation, and retry tutorials

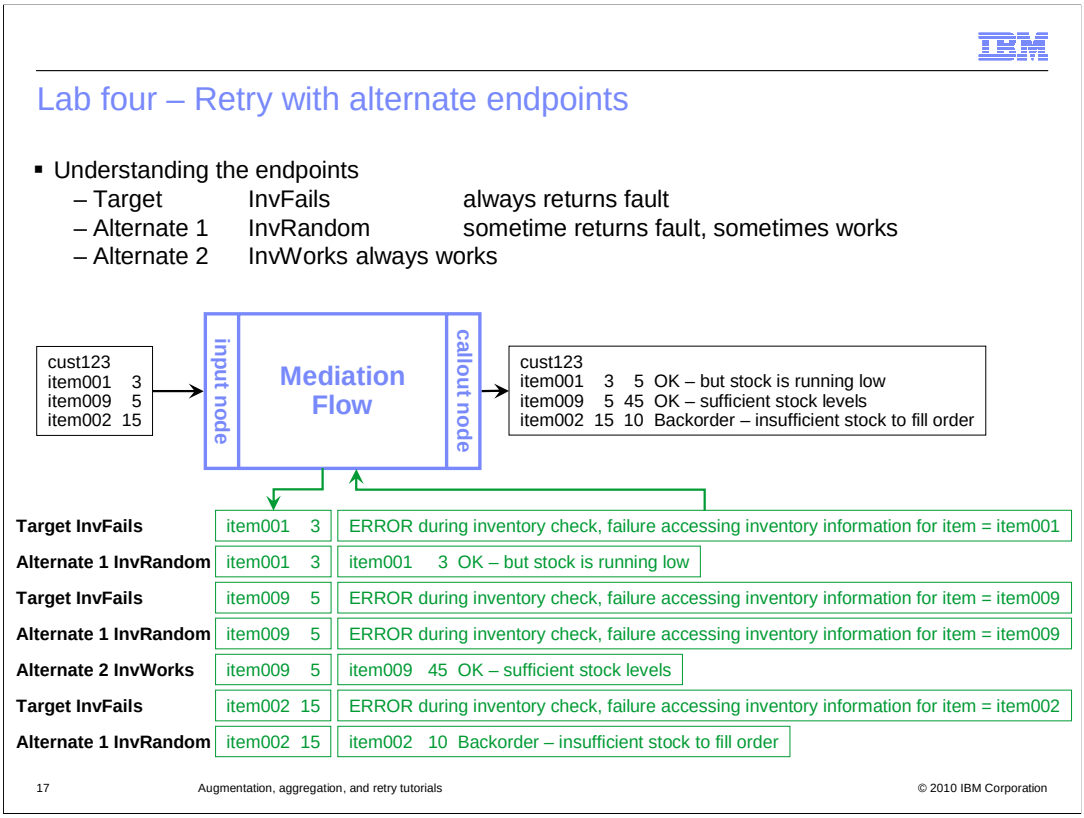
In this section of the lab, the inventory service that randomly works is still being used. It is implemented so that it will never return more than two sequential faults, nor will it work more than two times in a row. Therefore, the results are not entirely random. Consequently, when you are running the test, at least one fault will occur. Also, the retry limit is never reached, so the inventory calls for every item will always eventually be successful.

In the example data for the flow, the same three order items previously used are being passed into the mediation. Notice that there are actually four calls made to the inventory service rather than three. The call for the second item, item009, returned a fault. This resulted in the service invoke primitive performing a retry which was successful. In the output data sent to the shipping service you can see that inventory information has been included for all three items. The only evidence that a fault occurred is the extra call to the inventory service, which was only seen by the CheckInventory service invoke primitive.



Lab four illustrates the use of alternate endpoints when doing a service call retry. To do this, the target address field and alternate target address array in the SMO must be initialized with appropriate endpoint URLs. The initialization of these fields is typically accomplished using the endpoint lookup primitive which calls the WebSphere Service Registry and Repository. However, in this lab the fields are set using a message element setter.

The SetEndpoints message element setter is added to the flow before the StartIteration fan out primitive. It sets the SMO target address field and two alternate target addresses using URLs that point to exports in the module containing the inventory service. The CheckInventory service invoke is modified so that it uses the target address URL for the initial call, and then uses the alternate target addresses for the retries. In the screen capture, you can see the URLs for the targets set by the message element setter. You can also see the CheckInventory service invoke and the assembly diagram for the inventory service containing the three exports which are the service endpoints.



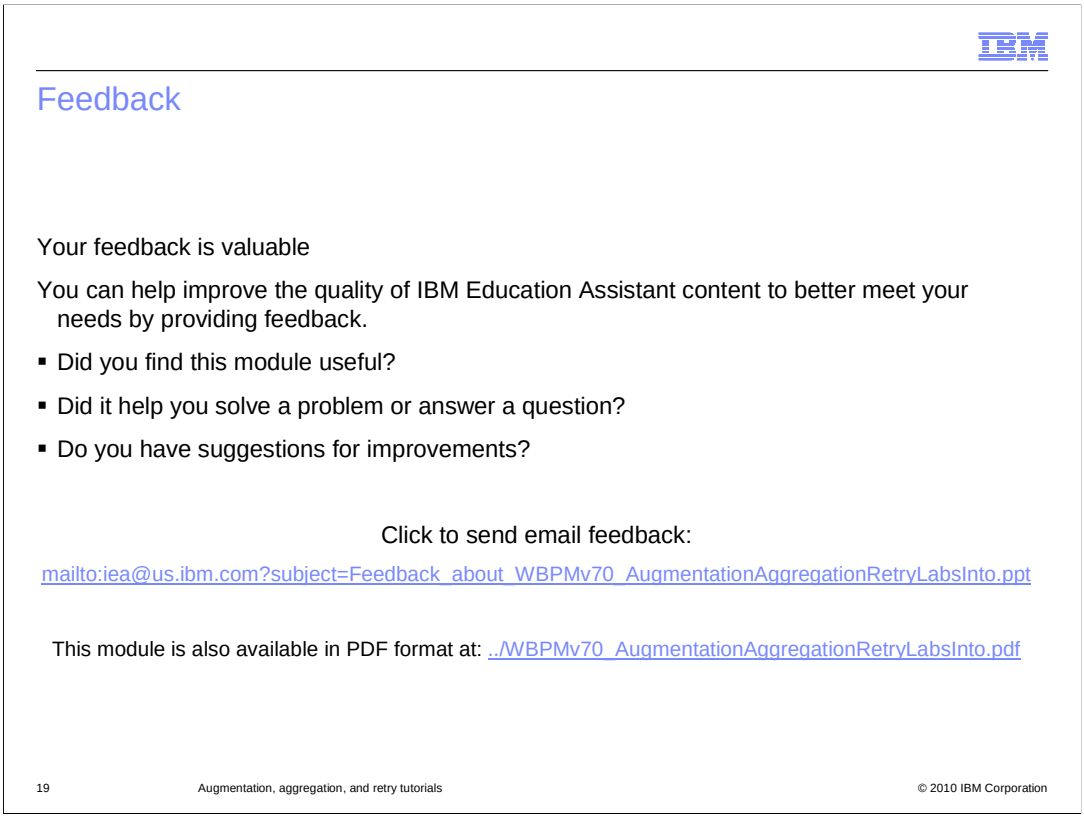
The three endpoints used in this lab exercise are implemented so that one will always return a fault, one will randomly work or return a fault and the third will always work. To aid in illustration of this function, the target address is set to the endpoint that will always fail. The first alternate target is set to the endpoint that will randomly work and the second alternate target is set to the endpoint that always works. As a result, the first call to the inventory service for each item will always result in a fault. The first retry might be successful or return a fault. If the first retry returns a fault, the second retry is performed and it will always work.

Looking at the data, the same three order items are passed in. You can see seven calls to the inventory service. For item001 and item002, there is one failing call followed by success on the first retry. For item009, the first call fails as does the first retry, but the second retry works. Looking at the output going to the shipping service, you can see that the item array is fully populated with inventory information. The only evidence that faults occurred are the extra calls to the inventory service, which were only seen by the CheckInventory service invoke primitive.

Summary

- Introduced the augmentation, aggregation and retry lab exercises
- Defined the capability each lab exercise is designed to illustrate
- Provided a description of the mediation flow logic and the primitives used
- Illustrated the flows using example data

In summary, this presentation examined a series of lab exercises which are designed to illustrate message augmentation, message aggregation and service call retry. Each of the four labs was described in terms of the capability it is designed to illustrate. The mediation flow logic and primitives used were described. Finally, for each of the labs, example data was used to illustrate the behavior of the scenario.



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

Trademarks, disclaimer, and copyright information

IBM, the IBM logo, ibm.com, and WebSphere are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. 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

THE INFORMATION CONTAINED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY. WHILE EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION CONTAINED IN THIS PRESENTATION, IT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED. IN ADDITION, THIS INFORMATION IS BASED ON IBM'S CURRENT PRODUCT PLANS AND STRATEGY, WHICH ARE SUBJECT TO CHANGE BY IBM WITHOUT NOTICE. IBM SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, THIS PRESENTATION OR ANY OTHER DOCUMENTATION. NOTHING CONTAINED IN THIS PRESENTATION IS INTENDED TO, NOR SHALL HAVE THE EFFECT OF, CREATING ANY WARRANTIES OR REPRESENTATIONS FROM IBM (OR ITS SUPPLIERS OR LICENSORS), OR ALTERING THE TERMS AND CONDITIONS OF ANY AGREEMENT OR LICENSE GOVERNING THE USE OF IBM PRODUCTS OR SOFTWARE.