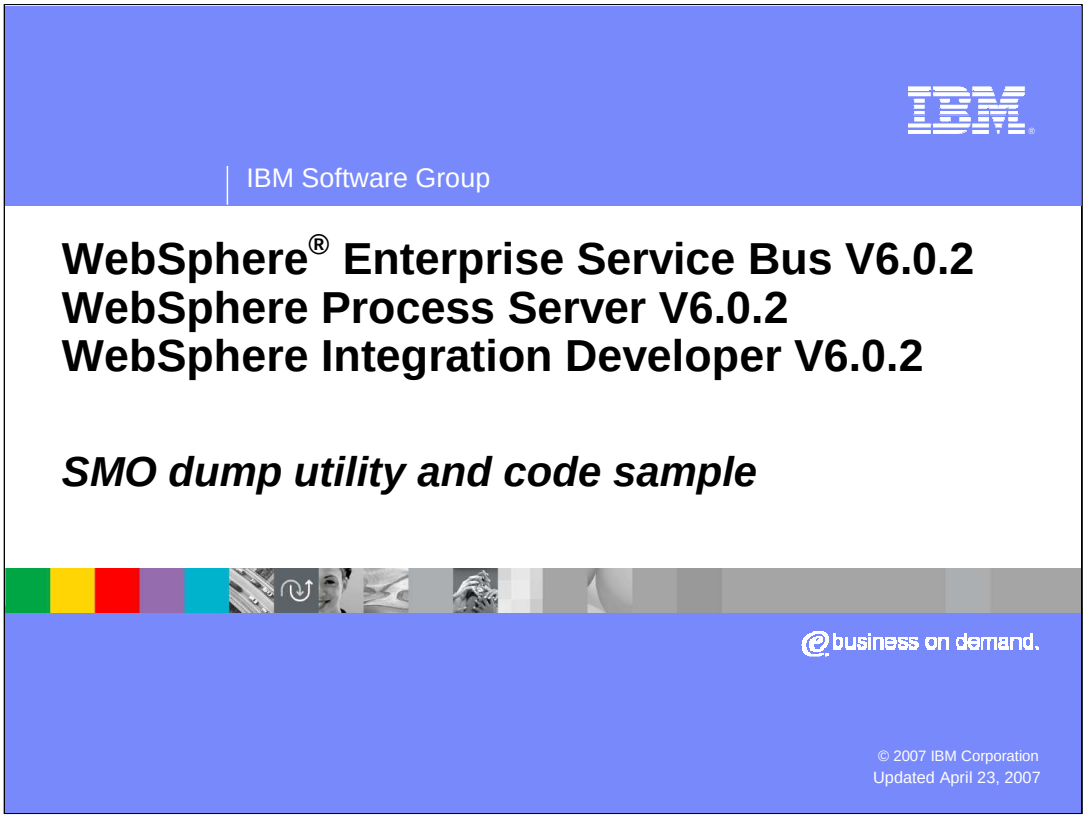


This presentation provides information about a Service Message Object (SMO) dump utility. It describes the usage of the utility and looks at the implementation, which serves as an example of SMO traversal.

Goals

- Introduce a DataObject dump utility
  - Useful for dumping the service message object (SMO)
  - Provide you with the code in a project interchange file
    - Can be imported into WebSphere Integration Developer
  - Describe how to use it
- Enhance your understanding of the SMO structure
  - Show sample dump output
  - Describe code from dump utility that traverses the SMO
- Enable you to examine the SMO in your own mediation flows

The goal of the presentation is to introduce you to the DataObject dump utility that is used for dumping the contents of the Service Message Object, or SMO. You are provided with a link to a project interchange file that you can import into WebSphere Integration Developer for your own use. How to instrument a mediation flow to call the dump utility is described.

The utility can be used to dump any DataObject. However, the focus of its use in this presentation is for the dumping of an SMO with the intention of enhancing your understanding of the SMO structure and contents. The presentation shows you a few sample output dumps of major sections of the SMO. Following that, the code for the dump utility is examined to show you have to write code that can traverse an SMO.

The project interchange file for this utility is being provided to you because the best way to enhance your understanding of the SMO is for you to use the dump utility in your own mediation flows. Being able to see what is really in the SMO will help you tremendously as you design, develop and debug your mediation flows.

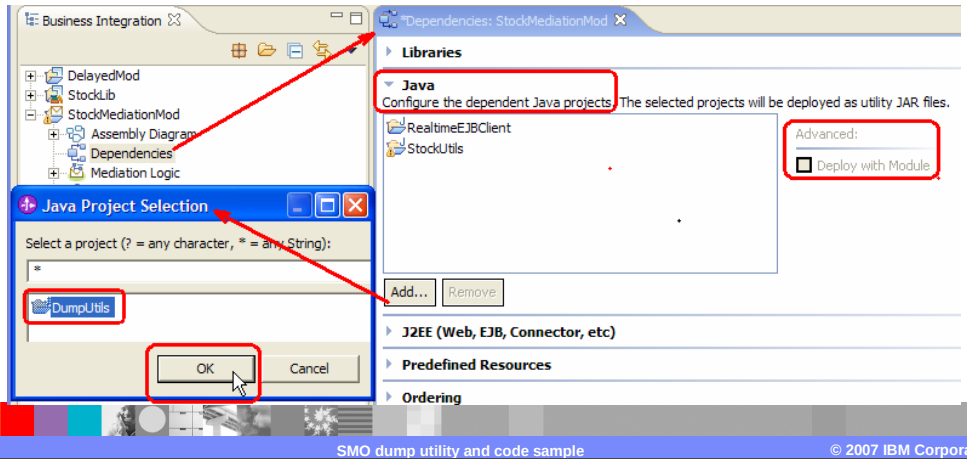
Section

How to use the dump utility

The first section of the presentation shows you how you can use the dump utility to dump the contents of the SMO.

Include dump utility in your mediation module

- To use the dump utility in your flow, you must:
  - **Download the DumpUtilsPI.zip** project interchange file
  - Import it into WebSphere Integration Developer
  - In your mediation module, add it as a Java™ project dependency



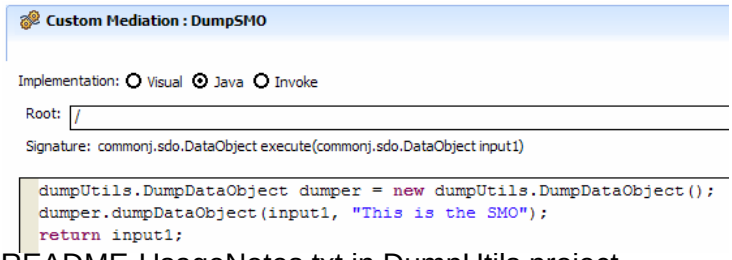
The dump utility is packaged in a WebSphere Integration Developer project interchange file named DumpUtilsPI.zip. You can download this using the link on this slide. There is also another link to it provided on the same page you used to launch this presentation.

Once you have downloaded the file, you need to import it as a project interchange file into your WebSphere Integration Developer workspace.

In order to use the utility in your Mediation Module a dependency to it must be created. The screen capture in the slide basically shows what you need to do. Start by opening the Dependencies editor for the Mediation Module and then open the Java section which allows you to add dependencies to Java projects. Click the Add… button to open the Java Project Selection dialog and then select the DumpUtils project and hit OK. This adds the project to the list of Java projects used by this module. Click on the DumpUtils project in the list and check to ensure that the Deploy with Module check box has been selected.

Call dump utility from a custom mediation

- Calling the dump utility
  - Create a custom mediation with java snippet
  - Select the root property as **/** or **/body**
  - Add the Java code to call the dump utility



- See README-UsageNotes.txt in DumpUtils project
  - Documents usage
  - Can cut and paste the code into the Java snippet
  - You need to change to the Java perspective to see DumpUtils

In your mediation flow, you can call the dump utility using a Custom Mediation primitive with a Java snippet implementation. Depending upon what part of the SMO you would like to dump, set the value for the Root property to either / (slash) or /body.

There are three lines of code that you need to add as is shown in the screen capture on the slide. The first line creates the object which provides the dump functionality and the second line uses the object to perform the dump of the SMO. The method to call is named dumpDataObject and it takes two parameters, the first being the DataObject to dump and the second is an optional comment that is included at the beginning of the dump.

The last line returns the SMO, a requirement of any Custom Mediation primitive implementation.

There are usage notes provided in the DumpUtils project in the read me file referenced in the slide. It contains full documentation for the utility, including code snippets which can be cut and pasted directly into your Custom Mediation. This project is not visible in the Business Integration perspective of WebSphere Integration Developer, so you have to switch to the Java perspective to open the read me file documentation.

Dumping selected elements of the SMO

- Utility can dump selected elements in the SMO
  - Element cannot be a leaf, must be a DataObject
  - Specification of Root determines starting point
- Examples:

```
// Dump all of the headers
dumper.dumpDataObject(input1.getDataObject("headers"), "Dump SMO headers");

// Dump just the JMS headers
dumper.dumpDataObject(input1.getDataObject("headers/JMSHeader"), "Dump SMO headers");

// Dump the failure info on flow from a fail terminal
dumper.dumpDataObject(input1.getDataObject("context/failInfo"), "Dump failure info");

// Dump the correlation context
dumper.dumpDataObject(input1.getDataObject("context/correlation"), "Dump correlation context");

// Dump the body when root is /
dumper.dumpDataObject(input1.getDataObject("body"), "Dump message body");

// Dump the body when root is /body
dumper.dumpDataObject(input1, "Dump message body");
```


The utility is able to dump any DataObject. Therefore, you do not need to dump the entire SMO when the Root property is / (slash) or the entire SMO body when the Root is /body. Any element of the SMO can be dumped provided it is a DataObject rather than a leaf element.

This slide shows several example invocations of the utility. To dump an element of the SMO you need to do a getDataObject method on input1, the parameter containing the SMO. The string passed to this operation is a path to the element to dump, starting from the location in the SMO specified by the root property.

The first example dumps all the headers in the SMO, and the second example dumps just the JMS header.

The third example would be useful in a flow wired to a fail terminal because it is dumping the failInfo element that gets populated when a mediation primitive raises an exception.

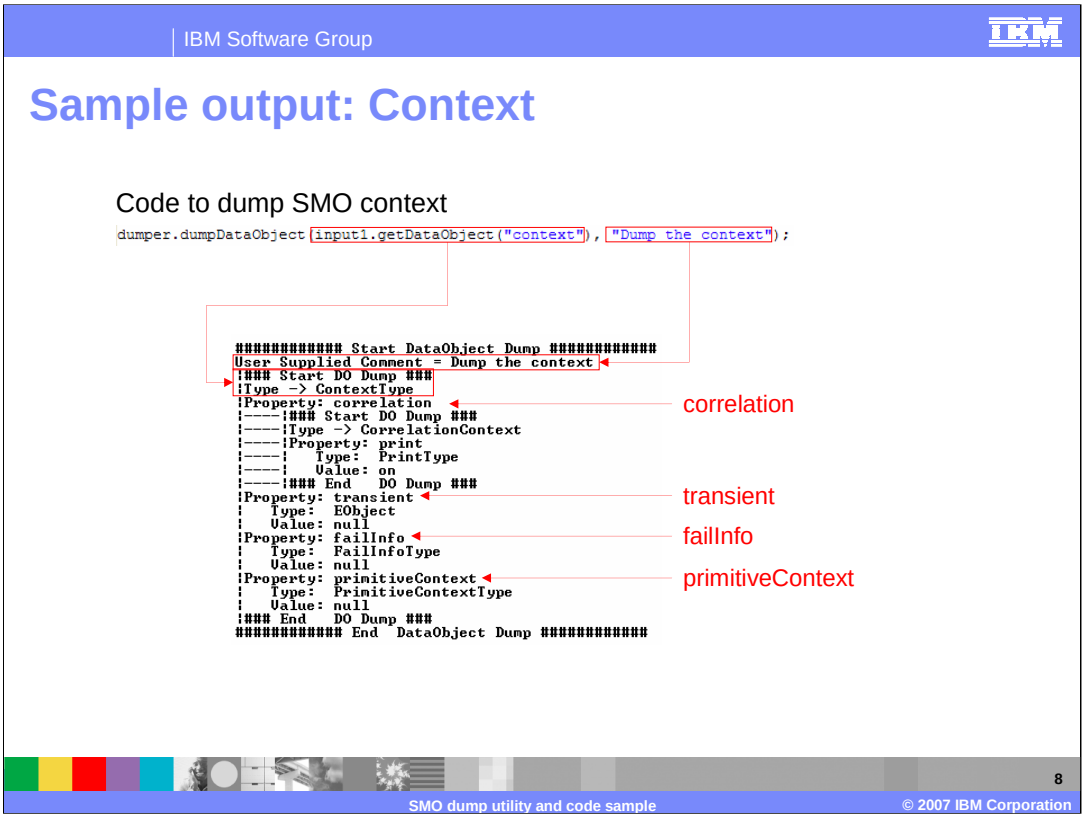
The fourth example dumps the correlation context.

The final two examples are really doing the same thing, dumping the entire contents of the SMO body. In the first case the root was specified as / (slash) and the method does a getDataObject for the element “body”. In the second case, the root was specified as /body, so input1 can be passed without doing a getDataObject method call.

Section

Sample output from the dump utility

In this section you are shown sample output from the dump utility. The dump output is directed to the SystemOut.log file for the server. It is also visible in the Console view of WebSphere Integration Developer.



This first example dumps the context section of the SMO, which requires that the root be specified as / (slash).

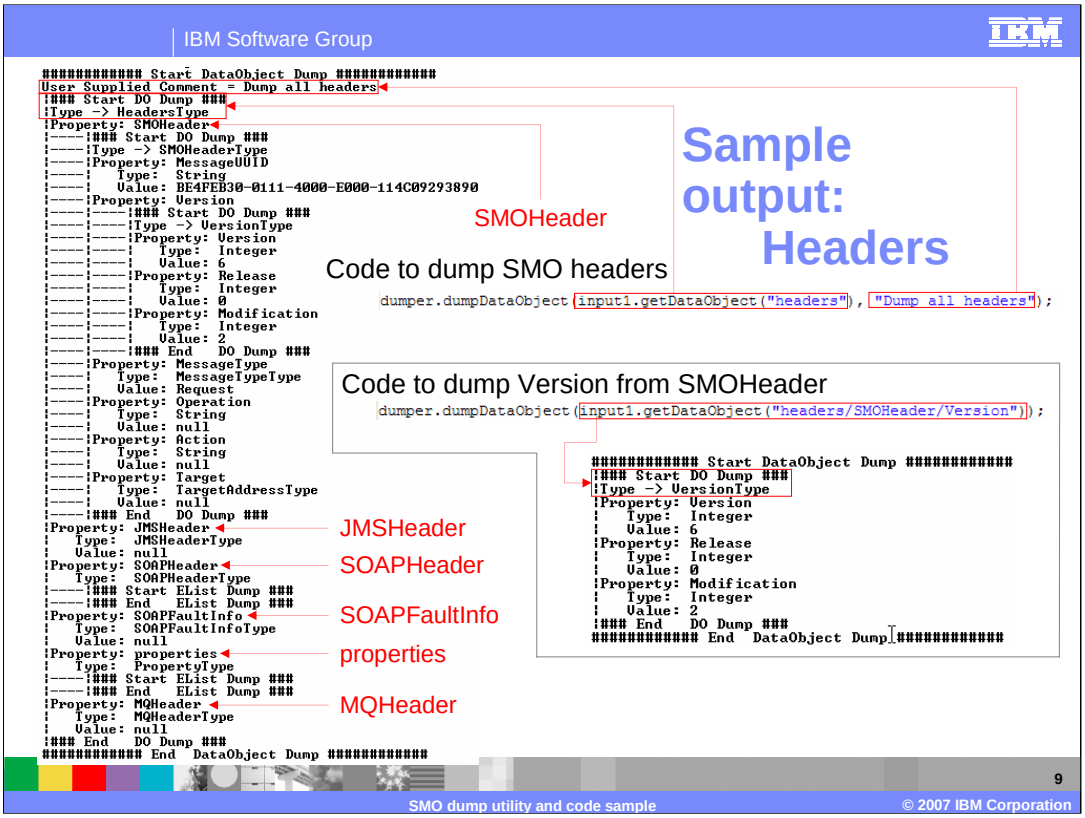
Looking at the example, notice that the second parameter supplies the comment “Dump the context”. This comment is placed at the beginning of the dump.

Notice the first parameter is the DataObject identified by the string “context” and that the DataObject shown in the dump is of type ContextType.

The context in the SMO has four properties. The first two are the correlation context and the transient context. Notice that the correlation context in this flow was configured but that the transient context was not.

The next property is the failInfo element which would be populated if this was a flow following a fail terminal.

The last property is the primitiveContext which is an area intended to be used by mediation primitive implementations. If would be populated if there was an Endpoint Lookup primitive in the flow.



There are two examples on this slide, one for the entire headers section of the SMO and the other for the Version contained within the SMOHeader. Both of these examples require that the root be specified as / (slash).

Similar to the example on the previous slide, both of these have a comment identifying what the dump is for. These will be placed at the beginning of the dumps.

In the top example, which is for all the headers, the first parameter is the DataObject identified by the string “headers”. The DataObject shown in the dump is of type HeadersType. In this particular example, only the SMOHeader has been populated, as it always is. The other headers include the JMSHeader, SOAPHeader, SOAPFaultInfo, properties and MQHeader. These would be populated with the appropriate information based on the type of SCA binding that was configured on the Export or Import that initiated this flow. Examination of what is contained in the headers is one of the very useful applications of the dump utility.

The second example illustrates that you can drill down into the SMO to dump a very specific section. In this case, the DataObject to dump was identified by the string “headers/SMOHeader/Version”, which is the full path to the Version information contained in the SMOHeader.

Sample output: Body

Code to dump body (Root was set to /body)

```
dumper.dumpDataObject(input1, "Dump body in response flow");
```

```
############# Start DataObject Dump #############
User Supplied Comment = Dump body in response flow
!### Start DO Dump ###
!Type -> getQuoteResponseMsg
!Property: getQuoteResponse
!----!### Start DO Dump ###
!----!Type -> getQuoteResponse_._type
!----!Property: quote
!----!----!### Start DO Dump ###
!----!----!Type -> Quote
!----!----!Property: symbol
!----!----!    Type:  String
!----!----!    Value: ibm
!----!----!Property: price
!----!----!    Type:  String
!----!----!    Value: 63.43
!----!----!Property: quotedTime
!----!----!    Type:  String
!----!----!    Value: Wed Apr 04 17:19:29 CDT 2007
!----!----!Property: comment
!----!----!    Type:  String
!----!----!    Value: quote by DelayedService
!----!----!### End   DO Dump ###
!----!### End   DO Dump ###
!### End   DO Dump ###
############# End  DataObject Dump #############
```

getQuoteResponseMsg
The type of the body is the message type of the SMO

Quote
Business object type of the output parameter

In this example, the entire body of the SMO is being dumped in a Custom Mediation primitive with the root property specified as /body. Therefore, the data object passed to the dump is the input parameter, input1, without any further qualification. Notice that the DataObject type is getQuoteResponseMsg which corresponds to the message type of the SMO as defined in the flow. Therefore, this dump was taken on the response flow for a getQuote operation. The output for the operation contains the Business Object named Quote which has several attributes, specifically symbol, price, quotedTime and comment.

Section

Dump utility code

Now that you are familiar with the output of the dump utility, this section examines the code. This provides some good sample code for showing how to traverse a DataObject.

Code for DumpDataObject – Public methods

- dumpDataObject(DataObject inputDO, String comment)
  - Public method to call when including a comment
  - Surrounds dump of the inputDO with starting and ending comments

```
public void dumpDataObject(DataObject inputDO, String comment) {
    System.out.println(" ");
    System.out.println("############ Start DataObject Dump ############");
    if (comment != null)
        System.out.println("User Supplied Comment = " + comment);
    dumpDO(inputDO, "|");
    System.out.println("############ End  DataObject Dump ############");
    System.out.println(" ");
}
```

dumpDO
internal method that does the actual work

- dumpDataObject(DataObject inputDO)
  - Public method to call without providing a comment
  - Calls other dumpDataObject method with a null comment

```
public void dumpDataObject(DataObject inputDO) {
    dumpDataObject(inputDO, null);
}
```


This slide shows the two public methods that can be called to do a DataObject dump. They are both called dumpDataObject and take a DataObject as input. The one on the top of the slide also takes a second parameter containing a comment to put into the dump.

The method on the top of the slide is very straight forward. It prints start and end of dump messages, in between which is a call to a private method named dumpDO which performs the actual dump of the DataObject.

The method on the bottom of the slide allows a user to call for a DataObject dump without supplying a comment. It is implemented by calling the other method and supplying a comment string of null.

Code for DumpDataObject – Private methods

- dumpDO(DataObject inputDO, String indent)
  - Recursive method, dumps the DataObject with indentation for nesting

```
private void dumpDO(DataObject inputDO, String indent) {
    if (inputDO != null) {
        System.out.println(indent + "### Start DO Dump ###");
        System.out.println(indent + "Type -> "
                + inputDO.getType().getName());
        List properties = inputDO.getType().getProperties();
        for (Iterator j = properties.iterator(); j.hasNext();) {
            Property p = (Property) j.next();
            System.out.println(indent + "Property: " + p.getName());
            String pName = p.getName();
            Object pValue = inputDO.get(pName);
            if (DataObject.class.isInstance(pValue)) {
                dumpDO((DataObject) pValue, "|----" + indent);
            } else  if (EObjectContainmentEList.class.isInstance(pValue)) {
                System.out.println(indent + "   Type:  " + p.getType().getName());
                dumpEList((EObjectContainmentEList) pValue, "|----" + indent);
            } else {
                System.out.println(indent + "   Type:  " + p.getType().getName());
                System.out.println(indent + "   Value: " + pValue);
            }
        }
        System.out.println(indent + "### End   DO Dump ###");
    }
    return;
}
```

indicate start of dump at this nesting level

print type of DataObject

get list of its properties

iterate, for each property

print property name

get property value

DataObject?, recursively dump

EList?, print type & call dumpEList to iteratively dump the list

all other properties, print type and value

indicate end of dump at this nesting level

This slide shows the dumpDO method which does the real work for dumping a DataObject. As input it takes the DataObject to dump and an indentation string that is added to the front of each line of output. This enables the visual nesting of DataObjects when the routine is called recursively to dump DataObjects within DataObjects.

The routine allows for the possibility that the DataObject passed in is null, in which case it does nothing.

When dumping a DataObject, it prints a line indicating that this is the start of the dump and then prints the type of the DataObject. Since a DataObject is composed of properties, a list of the contained properties is then obtained. An iterator is then used to iterate through the list of properties so that each property can be dumped. For each property, the name of the property is printed and the type of the property value is determined. If the property value is a DataObject, this operation is recursively called to dump it. If the property value is an EList, a method to dump the contents of the list is called. This method is examined on the next slide. If the property is neither a DataObject nor an EList, the type and value are printed. After all the properties are dumped an end of dump indicator is printed.

Code for DumpDataObject – private methods

- dumpEList(EObjectContainmentEList eList, String indent)
  - Recursive method, dumps the list with indentation for nesting
  - EObjectContainmentEList part of Eclipse Modeling Framework (EMF)
    - org.eclipse.emf.ecore.util.EObjectContainmentEList

```
private void dumpEList(EObjectContainmentEList eList, String indent) {
    if (eList != null) {
        System.out.println(indent + "### Start EList Dump ###");
        int count = 1;
        for (Iterator k = eList.iterator(); k.hasNext();) {
            Object item = k.next();
            System.out.println(indent + "EList Item " + count++);
            if (DataObject.class.isInstance(item)) {
                dumpDO((DataObject) item, "|----" + indent);
            } else if (EObjectContainmentEList.class.isInstance(item)) {
                dumpEList((EObjectContainmentEList) item, "|----" + indent);
            } else {
                System.out.println(indent + "   Item: " + item);
            }
        }
        System.out.println(indent + "### End   EList Dump ###");
    }
    return;
}
```

indicate start of dump at this nesting level

iterate, for each list item

print index for this item in the list

DataObject?, call dumpDO to iteratively dump the properties

EList?, recursively dump

print all other list items

indicate end of dump at this nesting level

The method on this slide is used to dump an EList, or more specifically an EObjectContainmentEList. Similar to the dumpDO routine, it can be called recursively to handle an EList within an EList. It also has support for the indentation string, allowing the output to have a visual indication of the nesting.

The EObjectContainmentEList class is part of the Eclipse Modeling Framework (EMF) and is defined in the package org.eclipse.emf.core.util.

If the EList passed in is null, the method returns immediately without any processing.

The method prints out start and end of EList dump messages around the dump of the list. The list is processed using an iterator to dump each item in the list, with each item being identified by an index number. If an item is a DataObject, the dumpDO method is called to dump it and if it is another EList, this method is called recursively. Otherwise, the item itself is printed.

Summary

- Introduced the DataObject dump utility
  - Provided the code in a project interchange file
  - Described how to use it
  - Showed sample dump output
  - Described the dump utility code
- Provided you with a tool you can use for:
  - Learning to gain a better understanding of the SMO
  - Examining the SMO during development and debug

In summary, this presentation provided you with a project interchange file that you can use to dump DataObjects. How to use the utility was described, along with sample output showing the results from dumping various sections of the SMO. The code used to perform the dump was then explained.

The dump utility has the potential to be useful to you. It can be used to gain a better understand of the structure and content of the SMO by dumping and examining the SMO for various different mediation flows. It can also be quite useful during the development and debugging of new mediation flows.

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

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

Trademarks, copyrights, and disclaimers

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM WebSphere

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

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.

Information is provided "AS IS" without warranty of any kind. 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.