

IMS



IMS Connector for Java 2.2 and 9.1.0.1 Online Documentation for WebSphere Studio Application Developer Integration Edition 5.1.1

IMS



IMS Connector for Java 2.2 and 9.1.0.1 Online Documentation for WebSphere Studio Application Developer Integration Edition 5.1.1

Note

Before using this information and the product it supports, read the information in Notices at the end of this book.

(Second Edition 2005)

© Copyright International Business Machines Corporation 2000, 2005. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Chapter 1. What is the IMS resource adapter? 1

Prerequisites for using the IMS resource adapter	3
Platform configurations and communication protocol considerations	5
Preparing to use the IMS resource adapter	6

Chapter 2. Developing your application 7

Overview of the Common Client Interface (CCI) record helper class	7
---	---

Chapter 3. Configuring your application 9

Execution timeout	9
Valid execution timeout values	9
Setting execution timeout values	11
Socket timeout	12
Setting the Socket Timeout Value	13
Connection properties	14
Operation binding properties	17

Chapter 4. Security 27

IMS resource adapter security	27
Component-managed EIS sign-on	28
Configuring component-managed EIS sign-on	29
Container-managed EIS sign-on	30
Configuring container-managed EIS sign-on	31
Overview of secure socket layer (SSL)	32
Using secure socket layer (SSL) support	34

Chapter 5. Commit mode processing 39

Overview of commit mode processing	39
SYNC_SEND_RECEIVE programming model	44
Retrieving asynchronous output	46
Displaying output message counts.	48
SYNC_SEND programming model	49
Creating an application to run a Commit mode 0 transaction.	51
Displaying output message counts.	55

Chapter 6. Transaction processing. . . 59

Global transaction support with two-phase commit	59
Two-phase commit prerequisites	62

Using global transaction support in your application	63
Two-phase commit environment considerations	64

Chapter 7. Diagnosing problems. . . . 65

Diagnosing problems when using the IMS resource adapter.	65
Logging and tracing with the IMS resource adapter	66
J2CA0056I, WLTC0017E, HWSP1445E, and HWSSL00E Error Messages	67
IMS resource adapter messages and exceptions	69

Chapter 8. Migration and coexistence 95

Migration and coexistence considerations for the IMS resource adapter	95
Compatibility of existing applications with IMS Connector for Java Version 2.2.1	95

Chapter 9. Samples 99

Sample: Creating an enterprise service for an IMS transaction.	99
Sample: Deploying an IMS enterprise service to a production server	121
Sample: Running an enterprise service for an IMS transaction	126
Sample: Building a service that submits commands to IMS.	131
Sample: Building container-managed and component-managed transactional EJBs to run IMS transactions	138
Sample: Building input and output records using the CCI record helper class.	165
Sample: Creating an Enterprise Java Bean to communicate with a conversational IMS application	182
Sample: Building an Application to Process Variable Length and Multiple Segment IMS Transaction Output Messages	199
Sample: Building an Application to Process IMS Transaction Input and Output Messages Containing Arrays.	208

Notices 219

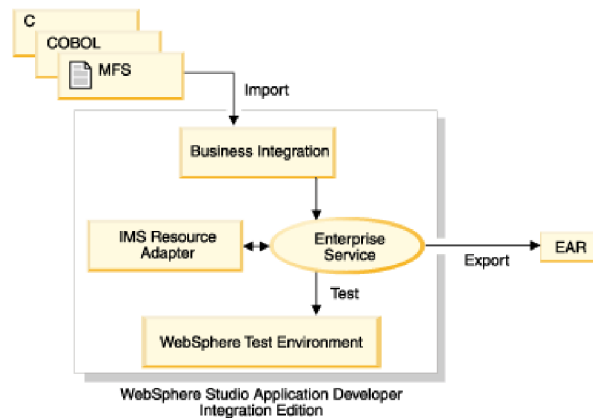
Chapter 1. What is the IMS resource adapter?

WebSphere® Studio Application Developer Integration Edition is a service-based development environment and the IMS resource adapter is one of the service providers included in it. The IMS resource adapter is used by Java™ applications to access IMS™ transactions running on host IMS systems and is used during development and at runtime. The IMS resource adapter is also called IMS Connector for Java.

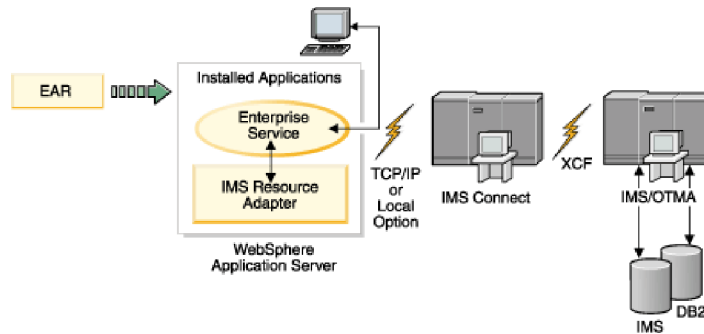
The process of building a Java application that runs an IMS transaction is summarized by the following steps:

1. In the Business Integration perspective, import C, COBOL, or MFS definitions of the IMS transaction input and output messages.
2. Generate an enterprise service and Java application using the imported definitions.
3. Test the service-based Java application using the WebSphere Test Environment.
4. When the Java application is tested, export it as an Enterprise Application Archive (EAR) file, to be deployed in WebSphere Application Server.

The following figure illustrates the use of the IMS resource adapter during development:



At run time, the IMS resource adapter is used with IBM WebSphere Application Server. When a Java application runs, it submits a transaction request to IMS through the host product, IMS Connect. The IMS resource adapter communicates with IMS Connect using TCP/IP or Local Option. IMS Connect then sends the transaction request to IMS OTMA using XCF (Cross-system Coupling Facility), and the transaction runs in IMS. The response is returned to the Java application using the same path. The following figure illustrates the run-time process:



There are two versions of the IMS resource adapter included in WebSphere Studio. Both are based on Version 1.0 of the Java 2 Platform, Enterprise Edition (J2EE) Connector (J2C) architecture. This information does not describe the J2EE Connector architecture in general. For information on the J2C architecture and its concepts, see the J2EE Connector Architecture Specification at <http://java.sun.com/j2ee/download.html>.

IMS Connector for Java V2.2.x is a Data Management tool offering shipped as a component of IMS Connect V2.2. IMS Connector for Java V9.1.0.1.x is a repackaging of IMS Connector for Java V2.2.x in IMS V9.1 and is functionally equivalent to IMS Connector for Java V2.2.x. A license for IMS V9.1 is required to run an application that uses IMS Connector for Java V9.1.0.1.x.

The IMS resource adapter:

- Provides global transaction and two-phase-commit support
- Provides run as thread identity support
- Supports component-managed and container-managed security
- Supports pooling and reuse of connections
- Supports SSL communication between IMS Connector for Java and IMS Connect
- Supports both commit mode 1 and commit mode 0 IMS transactions
- Supports the retrieval of output messages queued as the result of a failed commit mode 0 interaction or by insertion to an alternate PCB
- Supports conversational processing
- Provides control of whether undelivered output for commit mode 0 interactions on shareable persistent socket connections is queued or discarded. This function is controlled by the **purgeAsyncOutput** property.
- Supports specification of the name of a destination for undelivered output for commit mode 0 interactions on shareable persistent socket connections. This function is controlled by the **reRoute** flag and **reRouteName** properties.
- Provides enhanced control of the retrieval of undelivered output with the introduction of two new interaction verbs:
SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT and
SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT.
- Supports the use of RACF keyrings as SSL keystores and truststores.

See IMS resource adapter APIs for additional information on the IMS resource adapter's J2C classes and interfaces.

The IMS resource adapter is included in WebSphere Studio for use in the development of service-based Java applications that access IMS transactions. The run-time component of the IMS resource adapter is packaged as a component of

IMS Connect V2.2 (5655-K52). The run-time component of the IMS resource adapter V9.1.0.1.x is part of IMS Connect V9.1 (5655-J38). Both are deployed as Resource Adapter Archive (RAR) files to WebSphere Application Server and can be used to run service-based Java applications developed using WebSphere Studio.

The IMS resource adapter is primarily intended for use by services that submit transactions to IMS. However, the IMS resource adapter can also be used by services that submit IMS commands to IMS.

IMS resource adapter and MFS formatting

WebSphere Studio, in conjunction with the IMS resource adapter, can be used to create service definitions for both MFS and non-MFS based IMS transactions. For MFS IMS transactions, the IMS resource adapter formats transaction input and output messages based on MFS source files. For non-MFS based IMS transactions, the IMS resource adapter formats transaction input and output messages based on IMS application program data structures (such as copybooks). In both cases, MFS online processing is bypassed because the transaction input and output messages are provided to IMS using OTMA.

For more information on MFS-based transactions, see What is MFS?

Prerequisites for using the IMS resource adapter

This topic describes the prerequisites for using the IMS resource adapter as well as the supported software configurations.

WebSphere Studio Application Developer Integration Edition, Version 5.1.1 includes the following IMS resource adapters:

- **IMS Connector for Java Version 2.2.x**
This version of the IMS resource adapter is based on Version 1.0 of the J2EE Connector Architecture (JCA 1.0). This version of IMS Connector for Java runs with WebSphere Application Server Version 5.0.2 and above for distributed and z/OS platforms.
- **IMS Connector for Java Version 9.1.0.1.x**
This version of the IMS resource adapter is based on Version 1.0 of the J2EE Connector Architecture (JCA 1.0). This version of IMS Connector for Java runs with WebSphere Application Server Version 5.0.2 and above for distributed and z/OS platforms.

The following new function is included in Versions 2.2.4 and 9.1.0.1.2 of the IMS resource adapter:

- The control of whether undelivered output for commit mode 0 interactions on shareable persistent socket connections is queued or discarded. This function is controlled by the **purgeAsyncOutput** property.
- The option to provide the name of a destination for undelivered output for commit mode 0 interactions on shareable persistent socket connections. This function is controlled by the **reRoute** flag and **reRouteName** properties.
- Enhanced control of the retrieval of undelivered output with the introduction of two new interaction verbs: **SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT** and **SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT**.
- Support for use of RACF keyrings as SSL keystores and truststores.

Version 2.2.1 of the IMS resource adapter was included with WebSphere Studio Application Developer Integration Edition, Version 5.1; and Version 2.1.0.4 of the IMS resource adapter is included with WebSphere Studio Application Developer Integration Edition, Version 5.0. The prerequisites depend on which version of the IMS resource adapter that your application uses. These prerequisites apply to running an application using a WebSphere test environment inside Integration Edition and to running an application in a stand-alone WebSphere Application Server.

Note: Version 2.2.0 of the IMS resource adapter is English-only and was replaced by Version 2.2.1 which is the National Language version.

Version 2.1.0 added the following functions:

- Support for Secure Sockets Layer (SSL) connections between IMS Connector for Java and IMS Connect
- Support for SYNC_SEND_RECEIVE and SYNC_RECEIVE_ASYNCOUTPUT Commit Mode 0 interactions on a transaction socket
- Global transaction support using TCP/IP communication between IMS Connector for Java and IMS Connect
- Support for IMS services that use the Message Format Services (MFS) support of the IMS resource adapter

The prerequisites for new Version 2.1.0 functions are:

- IMS Connect Version 2.1
- IMS Version 8.1 or later releases

If your application does not use any of the new functions, earlier versions of the IMS resource adapter can be used and the prerequisites are:

- IMS Connect Version 1.2 or later releases
- IMS Version 7.1 or later releases

Version 2.2.1 added the following functions:

- Support for commit mode 0 interactions on dedicated and shareable persistent socket connections
- Support for Socket Timeout
- A CCI record helper class
- Support for IMS conversational transactions, including a sample application

The prerequisites for Version 2.2.1, regardless of the functions your application is using, are:

- IMS Connect Version 2.1 and an APAR or PTF
- IMS, Version 8.1 and an APAR or PTF, or later releases
- An APAR or PTF to WebSphere Application Server and the WebSphere Test Environment of Integration Edition

Version 2.2.2 added the following functions:

- Send Only support

The prerequisites for Version 2.2.2, regardless of the functions your applications are using, are:

- IMS Connect Version 2.1 and APARs or PTFs or,

- IMS Connect Version 2.2 and APARs or PTFs
- IMS, Version 8.1, or later releases

See the IMS Connector for Java web site (www.ibm.com/ims) for specific information about the PTFs and APARs required for IMS Connect and IMS. APAR numbers can be found in the README.html files for each release, as appropriate.

See the *IMS Connect Guide and Reference* for information about which versions of OS/390® and z/OS® are required. **Note:** The SSL support included in the IMS Connect, Version 2.1.0 and higher requires z/OS, Version 1.4 or higher.

For information regarding deployment of existing applications for use with IMS Connector for Java Version 2.2.1, see Compatibility of Existing Applications with IMS Connector for Java Version 2.2.1. For information regarding deployment of existing MFS applications, see Migration considerations for MFS-based applications.

If you use SSL, you might want to use a tool for key management. A key tool enables you to add or delete certificates to a keystore. IMS Connector for Java does not require specific key management tools for SSL support. Any tool that can manage 'JKS' keystores and X.509 certificates can be used. Some of the commonly used tools are Keytool (shipped with JDK1.4) and IKEYMAN (IBM's key management tool).

Platform configurations and communication protocol considerations

The communication protocol you use depends on the platform configuration of WebSphere Application Server and IMS. The IMS resource adapter can be deployed to WebSphere Application Server for distributed platforms (AIX®, HP_UX, Linux, Linux for z/OS, Solaris, or Windows®) and to WebSphere Application Server for z/OS. The IMS resource adapter, deployed in WebSphere Application Server, can communicate with IMS Connect using either the TCP/IP or Local Option communication protocol. Where TCP/IP uses sockets, Local Option provides non-socket access (an MVS™ program call) to IMS Connect from WebSphere Application Server for z/OS.

- If WebSphere Application Server is running on a distributed platform, you must use TCP/IP to connect to IMS Connect.
 - If you use global transaction (two-phase-commit) support with TCP/IP, RRS is required. Also, IMS Connect, IMS, and RRS must reside in the same MVS image.
- If WebSphere Application Server is running on z/OS, you can use either TCP/IP or Local Option to connect to IMS Connect depending on your configuration. For example:
 - If WebSphere Application Server and IMS Connect are on the same MVS image, you can use Local Option or TCP/IP; however Local Option is recommended.
 - If WebSphere Application Server and IMS Connect are on different MVS images, you must use TCP/IP.
 - If you want to use global transaction support and your IMS and WebSphere Application Server are on the same MVS image, the Local Option communication protocol is recommended. If you are using global transaction support with Local Option protocol, RRS, IMS, IMS Connect, and WebSphere Application Server must be in the same MVS image.

- If you want to use global transaction support and your IMS and WebSphere Application Server are on different MVS images, you must use TCP/IP as your communication protocol. If you are using global transaction support with TCP/IP protocol, RRS, IMS, and IMS Connect must reside in the same MVS image.

The following table describes the relationship between the different platform configurations, communication protocols, and global transaction support:

Platform of WebSphere Application Server with IMS resource adapter	Supported communication protocol	Global transaction (two-phase-commit) support
AIX	TCP/IP	Yes*
HP_UX	TCP/IP	Yes*
Linux	TCP/IP	Yes*
Linux for zSeries® and S/390®	TCP/IP	Yes*
Solaris	TCP/IP	Yes*
Windows	TCP/IP	Yes*
z/OS, OS/390	TCP/IP	Yes*
	Local Option	Yes

* Global transaction support with TCP/IP requires IMS Connect 2.1 or later.

Preparing to use the IMS resource adapter

The IMS resource adapters, IMS Connector for Java Versions 2.2.4 packaged in `ims224.rar` and 9.1.0.1.2 packaged in `ims91012.rar` are provided as part of the J2C feature of WebSphere Application Developer Integration Edition 5.1.1. IMS Connector for Java Version 2.2.4 and 9.1.0.1.2 are based on Version 1.0 of the J2EE Connector Architecture and provide equivalent function.

The version of the IMS resource adapter that you use depends on which version of IMS Connect you use. IMS Connector for Java Version 2.2.4 is intended for use with IMS Connect 2.2 while IMS Connector for Java Version 9.1.0.1.2 is targeted to IMS Connect 9.1.

If you are planning to develop a Java application that runs an IMS transaction, you must import the IMS resource adapter into your workbench (for example, into the Business Integration perspective). For information about how to do so, see [Importing a resource adapter](#).

For information on how to deploy the IMS resource adapter on WebSphere Application Server, see the appropriate `README.html` file that is available on the [IMS Connector for Java Downloads](#) page.

Chapter 2. Developing your application

There are a couple of ways to develop an application. One way to develop your application is to use the tooling provided in WebSphere Studio Application Developer Integration Edition. Another way is to use the J2C Common Client Interface (CCI) provided by the IMS resource adapter.

We recommend that you use WebSphere Studio Application Developer Integration Edition to generate code because the tooling provides the following benefits:

- Helps optimize and simplify building, testing, integrating, and deploying J2EE applications
- Assists large-scale application development
- Provides real-time application flexibility with WebSphere Application Server

If you choose to use CCI to develop your application, the IMS resource adapter provides a CCI helper class to help you create the transaction input and output messages for your application.

Overview of the Common Client Interface (CCI) record helper class

Note: The class IMSCCIRecord is deprecated as of IMS Connector for Java Version 9.1.0.1.2 and IMS Connector for Java Version 2.2.4. The functions provided by this class are now available in the development environments WebSphere Studio Application Developer Integration Edition and Rational Application Developer. For more information, see:

- “Building a Java application that uses the J2EE Connector Architecture Common Client Interface” on the IMS Examples Exchange at <http://www.ibm.com/software/data/ims/examples/exHome.html>
- “Using IMS data bindings in a CCI application” in the online help for the IMS resource adapter in Rational Application Developer

The Common Client Interface (CCI) API, a part of the Jave 2 Enterprise Edition (J2EE) connector architecture, provides access from J2EE clients, such as enterprise beans, JavaServer Pages (JSP) technology, and servlets, to an underlying Enterprise Information System (EIS), such as IMS. You can use WebSphere Studio Application Developer Integration Edition to build applications that access IMS transactions or you can write applications using CCI.

The tooling in WebSphere Studio Application Developer parses the cobol copybook for you and generates the classes needed to communicate with the IMS resource adapter using WebSphere invocation framework (WSIF). These classes create the input byte array to be sent to IMS and extract the fields from the output byte array from IMS and make it available to the client application. If you write your own application using CCI, you must provide your own conversion routines and create the input byte array for IMS.

To simplify the process of writing applications using the CCI, the IMS resource adapter provides a CCI record helper class, IMSCCIRecord, that can be used with any J2EE-compliant application server. IMSCCIRecord is an external, stand-alone helper class that incorporates routines to take care of the data conversion and create the byte array for IMS. You can use the CCI record helper class to extend

your input and output records. The CCIRRecord helper class contains two APIs, field-specific and type-specific. These two APIs provide you with some flexibility in using the helper class.

Type-Specific API

In this API, the input and output records are very simple. You need only specify the message format of the input and output record classes. The setter and getter methods for the fields are implemented in the IMSCCIRRecord. The setter method format is `setTypeName()`; for example, `setString()`. To use this API, the client application needs the type of the field of the set and get methods in order to call the appropriate method. Also, the type-specific API does not require any setter or getter methods in the input and output record helper classes. Rather, you need to provide the `DLTypeInfo`. This model is similar to the one used in IMS Java.

Field-Specific API

In this API, you code the specific methods that you want to call and the associated parameters. When you want to customize setter and getter methods for each field, you should use the field-specific API. Specifying each function for each method takes more code to write, but it is customized exactly to your application and need only be written once. In addition, it is easier to write the classes because you can call a single method to set and get fields of any type. The setter method format is `set_FieldName()`; for example, `set_InLast1()` where `InLast1` is a field of the transaction. WebSphere Studio Application Developer Integration Edition uses this format for naming methods in some of its generated classes.

Chapter 3. Configuring your application

The topics in this section describe how to configure your application for your service. The topics included are:

Execution timeout

The *execution timeout value* for the IMS resource adapter is defined as the maximum amount of time allowed for IMS Connect to send a message to IMS and receive a response from IMS. For details about the execution timeout value, see Setting execution timeout values and Valid execution timeout values.

Before the introduction of the `executionTimeout` property, you were limited to setting a timeout value on a global level, which was specified in the IMS Connect configuration file. Every interaction between IMS Connect had the same timeout value.

With the `executionTimeout` property, you can set individual timeout values on a per interaction basis rather than on a global basis. If an interaction isn't complete before timeout occurs, IMS Connect returns an error message to the IMS resource adapter. The IMS resource adapter returns an exception indicating that the duration of time for IMS to respond to IMS Connect has exceeded the execution timeout value.

Note: Because the connection between the IMS resource adapter and IMS Connect is persistent, when execution timeout occurs, the socket is not closed. Instead, the socket is available for reuse for subsequent interactions..

Execution timeout in conversational transactions

In a conversational transaction, the execution timeout value applies to each iteration of that conversation. An *iteration* consists of one input message sent to IMS and one output message received from IMS. If one iteration of the conversation times out, the entire conversation ends.

Execution timeout exceptions

If a valid execution timeout value is specified for a particular interaction and execution timeout occurs, the Java application submitting the interaction receives the exception `javax.resource.spi.EISSystemException`. If you specify an invalid execution timeout value, the exception `javax.resource.NotSupportedException` is thrown when execution timeout occurs.

Valid execution timeout values

The execution timeout value is represented in milliseconds and must be a decimal integer in the range of 1 to 3600000, inclusively. That is, the `executionTimeout` value must be greater than zero and less than or equal to one hour. The execution timeout value can also be -1 if you want an interaction to run without a time limit. The execution timeout value cannot contain non-numeric characters.

If you do not specify an execution timeout value or if the value that you specify is invalid:

- For SYNC_SEND_RECEIVE interactions, the timeout value in the IMS Connect configuration member is used and the interaction continues to run.
- For SYNC_RECEIVE_ASYNCOUTPUT, SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT, and SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT interactions, IMS Connect will set the timeout value to two seconds and the interaction continues to run.

Additionally, if you specify an invalid value, the exception `javax.resource.NotSupportedException` is thrown when timeout occurs for that interaction.

Tip: The host system administrator determines the global timeout value in the IMS Connect configuration member. To display this value, issue the `VIEWHWS` command on the MVS console. See the *IMS Connect User's Guide and Reference* (SC27-0946-03) for more information on the `VIEWHWS` command.

If a valid execution timeout value is set, this value is converted into a value that IMS Connect can use. The following table describes how the values you specify are converted to the values that IMS Connect uses:

Range of user-specified values	Conversion rule
1 - 250	If the user-specified value is not divisible by 10, it is converted to the next greater increment of 10.
251 - 1000	If the user-specified value is not divisible by 50, it is converted to the next greater increment of 50.
1001 - 60000	The user-specified value is converted to the nearest increment of 1000. Values that are exactly between increments of 1000 are converted to the next greater increment of 1000.
60001 - 3600000	The user-specified value is converted to the nearest increment of 60000. Values that are exactly between increments of 60000 are converted to the next greater increment of 60000.

For example, if you specify a value of 1, this value is converted to 10 (because 1 is not divisible by 10 and 10 is the next increment that is greater than 1). The following examples illustrate how the conversion works for each range of values:

User-specified value (milliseconds)	Converted value (milliseconds)
1	10
11	20
251	300
401	450
1499	1000
1500	2000
60000	60000

User-specified value (milliseconds)	Converted value (milliseconds)
89999	60000
3600000	3600000
3750000	3600000

Setting execution timeout values

`executionTimeout` is a property of the `IMSInteractionSpec` class. The execution timeout value that you set is converted to a value that IMS Connect uses. This conversion occurs to meet the requirements of IMS Connect. **Important:** Other timeout values can affect your interactions. If other timeout values are less than the execution timeout value you set for your IMS interaction, these other timeout values can cause the interaction to expire. Other timeout values include:

- Connection timeout property of J2C connection factories
- EJB transaction timeout value
- Browser timeout value
- Servlet HTTP session or EJB session timeout values

For example, when WebSphere Application Server is running on the z/OS platform, the server consists of two parts, a controller and a set of one or more servants. Application work is dispatched into servant regions. Application work is, by default, timed. In general, when an application in dispatch reaches its timeout, the servant region where it is dispatched is abended and restarted. The server stays up and continues taking work. For this reason, you should use care when choosing execution timeout values that are greater than WebSphere Application Server timeout values, or when choosing the execution timeout value of -1 (wait forever). In addition, if you are planning on disabling WebSphere Application Server timeouts, you should check the server documentation in order to better understand the implications of doing this.

Another example is if you configure the execution timeout value to be greater than the timeout value specified for a browser response, then the execution timeout value is never used because the browser timeout value is exceeded first.

You can provide a value for the `executionTimeout` property of an `IMSInteractionSpec` class in one of two ways:

- Using WebSphere Studio Application Developer Integration Edition
- Using the `setExecutionTimeout` method

With the first method, using WebSphere Studio Application Developer Integration Edition, you can set the execution timeout value when you initially define the operation binding properties for an IMS service. To see an example that includes defining the operation binding properties for an IMS service, follow the steps in Sample: Creating an enterprise service for an IMS transaction.

To edit the operation binding properties that are already defined for an IMS service, complete the following steps:

1. Open the appropriate IMS binding WSDL file using the WSDL Editor.
2. In the Bindings container of the Graph view, expand the IMS binding WSDL file and expand the appropriate binding operation file.

3. Select the **operation extensibility element** (for example, **ims:operation**) and edit the values of the properties in the property table.
4. Select the operation extensibility element again to indicate that changes have been made.
5. Close the editor and click **Yes** to save your changes.

Note: You can also code individual timeout values for different interactions using the method described below in Exposing the executionTimeout property of the IMSInteractionSpec and Using the setExecutionTimeout method. If you code an execution timeout value in your Java client application code, that value overrides any execution timeout value that you set in WebSphere Studio.

You can use the procedure outlined in InteractionSpec and ConnectionSpec properties as data to gain access to the executionTimeout property of the IMSInteractionSpec. Once you have done this, you can set the value of the executionTimeout property from within your application. This enables you to set separate executionTimeout values for each application. For specific information about exposing the IMSInteractionSpec and IMSConnectionSpec, see Creating an application to run a commit mode 0 transaction.

With the second method, you can use the setExecutionTimeout method to set an execution timeout value. If you are creating a CCI application, you will have access to the setExecutionTimeout method of the IMSInteractionSpec. To use the setExecutionTimeout method, you need to instantiate a new IMSInteractionSpec or obtain the IMSInteractionSpec from your specific interaction. Then, set the executionTimeout value for the IMSInteractionSpec by using the setExecutionTimeout method provided by the IMSInteractionSpec class. For example:

```
interactionSpec.setExecutionTimeout(timeoutValue);
```

After you set the executionTimeout value for the IMSInteractionSpec, assign this interactionSpec to the specific interaction.

Socket timeout

Socket timeout is the maximum amount of time IMS Connector for Java will wait for a response from IMS Connect before disconnecting the socket and returning an exception to the client application.

If there are network problems or routing failures, the socketTimeout property prevents a hang in the system where the client using the IMS resource adapter is waiting indefinitely for a response from IMS Connect. Because the socketTimeout property is based on the TCP/IP sockets with which IMS Connect and the IMS resource adapter use to communicate, the socketTimeout property is not applicable with Local Option.

With the socketTimeout property, you can set individual timeout values for a particular interaction using a socket. The value, in milliseconds, can be set on the socketTimeout property in IMSInteractionSpec. If the socketTimeout property is not specified for an interaction or it is set to zero milliseconds, this means there is no socket timeout and the connection will wait indefinitely. The default socket timeout value is zero.

When determining the Socket Timeout value, other existing timeout values should be taken into account. For example, browser session timeout value, Execution

Timeout, EJB transaction timeout value, WebSphere Application Server connection timeout value, and HTTP session timeout value used by servlets and stateful session beans.

If a valid socket timeout value is specified for a particular interaction and socket timeout occurs, a *java.io.IOException* is thrown and the J2EE JCA exception, *javax.resource.spi.CommException* is raised. The J2EE JCA exception message indicates that the client has spent more time than was allocated by the socketTimeout value to communicate with IMS Connect.

Setting the Socket Timeout Value

When setting the socketTimeout value, you need to consider the executionTimeout value. The executionTimeout property is the maximum amount of time allowed for IMS Connect to send a message to IMS and receive a response from IMS. The socketTimeout value encapsulates the executionTimeout value. Therefore, the socketTimeout value should be greater than the executionTimeout property because the socket may time out unnecessarily if its value is set to less than the executionTimeout value. The following table lists suggested values for socketTimeout based on executionTimeout values.

Execution Timeout Value (milliseconds)	Execution Timeout Behavior	Suggested Socket Timeout Value
0 (or no value)	The default value from the IMS Connect configuration file is used.	The socket timeout value should be greater than the execution timeout default value specified in the IMS Connect configuration file.
1 - 3,600,000	The wait response times out after the specified millisecond value.	The socket timeout value should be greater than the execution timeout value.
-1	The wait response is indefinite.	Set the socket timeout value to 0 so that the connection waits indefinitely.

There are two ways to set the socket timeout value. You can either write an application using the JCA Common Client Interface (CCI) to access the getter and setter methods provided with the *IMSInteractionSpec* or use the tooling provided by WebSphere Studio Application Developer Integration Edition.

Using the CCI application to set a socket timeout value

If you are creating a CCI application, you will have access to the *setSocketTimeout* method of the *IMSInteractionSpec*. To use the *setSocketTimeout* method, you need to instantiate a new *IMSInteractionSpec* or obtain the *IMSInteractionSpec* from your specific interaction. Then set the socketTimeout value for the *IMSInteractionSpec* by using the *setSocketTimeout* method provided by the *IMSInteractionSpec* class. For example:

```
interactionSpec.setSocketTimeout(timeoutValue1);
interaction.execute(interactionSpec,input,output);

interactionSpec.setSocketTimeout(timeoutValue2);
interaction.execute(interactionSpec,input,output);
```

Using WebSphere Studio Application Developer Integrated Edition to set a socket timeout value

You can use WebSphere Studio Application Developer Integrated Edition to set the socket timeout value when you initially define the operation binding properties for an IMS service. To edit the operation binding properties that are already defined for an IMS service, complete the following steps:

1. Open the appropriate IMS binding WSDL file using the WSDL Editor.
2. In the Bindings container of the Graph view, expand the IMS binding WSDL file and expand the appropriate binding operation file.
3. Select the **operation extensibility element** (for example, **ims:operation**) and edit the values of the properties in the property table.
4. Select the operation extensibility element again to indicate that changes have been made.
5. Close the editor and click **Yes** to save your changes.

Connection properties

When you create an IMS service definition or define an IMS connection factory to WebSphere Application Server, you must provide values for certain properties of the connection between IMS Connector for Java and IMS Connect. The following list describes these connection properties:

Host name

Mandatory for TCP/ IP connections: The IP address or host name of the machine running the target IMS Connect. You must replace the value "myHostNm " with a value that is valid for your IMS environment.

Port number

Mandatory for TCP/IP connections: The number of a port used by the target IMS Connect for TCP/IP connections. Multiple sockets can be open on a single TCP/ IP port. See "Configuring IMS Connect" in the *IMS Connect Guide and Reference* (SC27-0946-03) for additional information about the PortNumber property. You must replace the value of "0" with a value that is valid for your IMS environment.

CM0Dedicated

The default is false. A value of FALSE indicates the connection factory will generate shareable persistent socket connections and IMS Connector for Java will generate a clientID to identify the socket connection. These connections can be used by commit mode 0 and commit mode 1 interactions. A value of TRUE indicates the connection factory will generate dedicated persistent socket connections, which require user-specified clientIDs to identify the socket connections. A dedicated persistent socket connection is reserved for a particular clientID and only commit mode 0 interactions are allowed. This property applies to TCP/IP connections only.

SSL Enabled

The default is false. This property is only valid for TCP/IP connections. A value of true indicates that IMS Connector for Java will create an SSL socket connection to IMS Connect using the HostName and PortNumber specified in these connection properties. This port must be configured as an SSL port by IMS Connect. A value of false indicates that SSL sockets will not be used for connecting to the port specified in the Port Number property.

KeyStore Name

For non-z/OS platforms, specify the fully-qualified path name of your JKS keystore file. For z/OS, specify the name of your JKS keystore file as above, or a special string that provides the information needed to access your RACF keyring.

Private keys and their associated public key certificates are stored in password-protected databases called keystores. For convenience, trusted certificates can also be stored in the keystore and then the Truststore Name property can either be empty or could point to the keystore file. If the TrustStore Name/TrustStore Password property is left empty, an informational message is generated in the server log.

The keystore name can be used to specify either a JKS keystore or a RACF keyring when running on z/OS. An example of a fully-qualified path name of your JKS keystore file is *c:\keystore\MyKeystore.ks*. A RACF keyring is specified as: *keystore_type:keyring_name:racfid*. The *keystore_type* must be either **JCERACFKS** when software encryption is used for SSL or **JCE4758RACFKS** if hardware encryption is used. Replace *keyring_name* with the name of the RACF keyring that you are using as your keystore and *racfid* with a RACF ID that is authorized to access the specified keyring. Examples of RACF keyring specifications are "JCERACFKS:myKeyring:kruser01" or JCE4758RACFKS:myKeyring:kruser01". When running in z/OS, if the keystore name matches the above RACF keyring format, IMS Connector for Java will use the specified RACF keyring as its keystore. If the keystore type specified is anything other than **JCERACFKS** or **JCE4758RACFKS**, IMS Connector for Java attempts to interpret the keystore name specified as the name of a JKS keystore file.

Note: The JKS file can have other file extensions; it does not have to have to be .ks.

KeyStore Password

Specify the password for the keystore. Private keys and their associated public key certificates are stored in password-protected databases called keystores.

TrustStore Name

For non-z/OS platforms, specify the fully-qualified path name of your JKS truststore file. For z/OS, specify the JKS name or the RACF keyring of the truststore. The same format is used for the values of the Keystore Name and Truststore Name properties. See the description of the Keystore Name property for a discussion of this format.

A truststore file is a key database file (keystore) intended to contain public keys or certificates. For convenience, private keys can also be stored in the Truststore and then the Keystore Name property can either be empty or could point to the truststore file. If the KeyStore Name/KeyStore Password property is left empty, an informational message will be generated in the server log.

Note: The JKS file can have other file extensions; it does not have to have to be .ks.

TrustStore Password

Specify the password for the truststore. A truststore file is a key database file that contains public keys.

Encryption Type

Select the encryption type. Strong and weak are related to the strength of

the ciphers, that is, the key length. All those ciphers that can be used for export come under the weak category and the rest go into the strong category. By default, the encryption type is set to weak.

IMS Connect name

Mandatory for Local Option connections: The job name of the target IMS Connect. If the IMS Connect name is specified, it overrides the Host name, Port number, and SSL-related properties.

Default user name

Optional: The default security authorization facility (SAF) user name that will be used for connections created by this connection factory if no UserName property is provided by the application component.

Default password

Optional: The password that will be used for connections created by this connection factory if the default user name is used.

Default group name

Optional: The IMS group name that will be used for all connections created by this connection factory if the default user name is used.

Note: The GroupName property can only be provided in a component-managed environment.

Data store name

Mandatory: The name of the target IMS datastore. It must match the ID parameter of the Datastore statement that is specified in the IMS Connect configuration member. It also serves as the XCF member name for IMS during internal XCF communications between IMS Connect and IMS OTMA. You must replace the default value "myDStrNm" with a value that is valid for your IMS environment.

Trace level

Optional: The level of information to be traced. For additional information on trace level, see Logging and tracing with the IMS resource adapter.

TransactionResourceRegistration

Optional: The type of transaction resource registration (enlistment). Valid values are either "static" (immediate) or "dynamic" (deferred). If this property is set to "dynamic", the enlistment of the resource to the transaction scope will be deferred until the resource is used for an interaction for the first time.

MFS XMI Repository ID

A resource property of a defined J2C Connection Factory, which is accessible on the J2C options page of the server configuration. This field contains a unique name for identifying the repository location. This ID must match the repository field defined in the generated format handler of your application. The default for this field is "default".

MFS XMI Repository URI

A resource property of a defined J2C Connection Factory, which is accessible on the J2C options page of the server configuration. This field specifies the physical location of the XMI repository. Valid formats for this field include:

- file://path_to_xmi, where path_to_xmi is a directory on the local file system containing the xmi files, for example file://c:/xmi.

- `http://url_to_xmi`, where `url_to_xmi` is a valid url that resolves to a directory containing the xmi files, for example `http://sampleserver.com/xmi`.
- `hfs://path_to_xmi` where `path_to_xmi` is the HFS directory on the host z/OS. This format is only supported for WebSphere Application Server for z/OS.

Operation binding properties

When you define an operation for an IMS service, you must provide values for certain operation binding properties of the interaction with IMS. These properties are all part of the `IMSInteractionSpec`. The following list describes all of the operation binding properties of the `IMSInteractionSpec`, including those that are not set by the application component:

asyncOutputAvailable

This is an output only property. It can be used by a Java application to determine if there is queued output for the TPIPE associated with the connection used for a `commitMode 0` interaction. For dedicated persistent socket connections, this is the value in the `clientID` property of `IMSConnectionSpec`. For shareable persistent socket connections, this is value generated by IMS Connector for Java. The value of `asyncOutputAvailable` is true if there are messages in the queue. The `asyncOutputAvailable` property is not set on input by the application component. **Note:** If your Java application uses this property, it must be exposed as an output property of `IMSInteractionSpec`. See *Creating an application to run a Commit mode 0 transaction* for information on exposing the properties of `IMSInteractionSpec`.

convEnded

This is an output only property. It can be used by a Java application to determine if a conversation has ended (true). The `convEnded` property is not set on input by the application component. **Note:** If your Java application uses this property, it must be exposed as an output property of `IMSInteractionSpec`. See *Creating an application to run a commit mode 0 transaction* for information on exposing the properties of `IMSInteractionSpec`.

commitMode

Used by the IMS resource adapter to indicate the type of commit mode processing to be performed for an IMS transaction. See *Overview of commit mode processing* for more information. The `commitMode` property can be set to 0 or 1 when `interactionVerb` is set to `SYNC_SEND_RECEIVE`. When `interactionVerb` is set to `SYNC_RECEIVE_ASYNCOUTPUT`, `SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT`, `SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT`, or `SYNC_SEND`, IMS Connector for Java uses `commitMode 0`. `commitMode 1` is required when `interactionVerb` is set to `SYNC_END_CONVERSATION`.

If `commitMode` is 0 and a shareable persistent socket is used for the interaction, the `clientID` must not be specified. If `commitMode 0` is specified for an interaction on a shareable persistent socket, the output message from a transaction can be purged or rerouted. The undelivered secondary output from a program to program switch can also be purged or rerouted.

If a dedicated persistent socket connection is used for an interaction, the `commitMode` must be 0 and the `clientID` property of the

IMSConnectionSpec used for the connection must be provided. If a dedicated persistent socket is used for a commitMode 0 interaction, undelivered output messages are always recoverable and cannot be purged or rerouted.

socketTimeout

The maximum amount of time IMS Connector for Java will wait for a response from IMS Connect before disconnecting the socket and returning an exception to the client application. The socketTimeout value is represented in milliseconds. To use socket timeout, the value must be greater than zero. If a socket timeout is not specified for an interaction or it is supplied with a socket timeout value of zero milliseconds, this will result in no socket timeout or an infinite wait. For more information see Socket timeout and Setting socket timeout values.

executionTimeout

The maximum amount of time allowed for IMS Connect to send a message to IMS and receive a response. The executionTimeout value is represented in milliseconds and must be a decimal integer that is either -1 or between 1 and 3,600,000, inclusively. That is, the executionTimeout value must be greater than zero and less than or equal to one hour. If a -1 value is set for this property, the interaction will run without a time limit. For more information, see Execution timeout, Setting execution timeout values, and Valid execution timeout values.

imsRequestType

Indicates the type of IMS request and determines how output from the request is handled by the IMS resource adapter. Integer values are:

Value	Named constant in IMSInteractionSpecProperties	Description
1	IMS_REQUEST_TYPE_IMS_TRANSACTION	<p>The request is an IMS transaction. Normal transaction output returned by IMS is used to populate the application's output message. If IMS returns a "DFS" message, the IMS resource adapter throws an IMSDFSMessageException containing the "DFS" message.</p> <p>This value for imsRequestType is used for applications that are not generated using WebSphere Studio MFS support.</p>
2	IMS_REQUEST_TYPE_IMS_COMMAND	<p>The request is an IMS command. Command output returned by IMS, including "DFS" messages, is used to populate the application's output message. The IMSDFSMessageException is not thrown.</p> <p>This value for imsRequestType is used for applications that submit IMS commands.</p>

Value	Named constant in IMSInteractionSpecProperties	Description
3	IMS_REQUEST_TYPE_MFS_TRANSACTION	<p>This value for imsRequestType is reserved for applications that are generated using WebSphere Studio MFS support.</p> <p>Normal transaction output returned by IMS, as well as "DFS" messages, are used to populate the application's output message. The IMSDFSMessageException is not thrown.</p>

interactionVerb

The mode of interaction between the Java application and IMS. The values currently supported by the IMS resource adapter are:

Value	Named constant in IMSInteractionSpecProperties	Description
0	SYNC_SEND	<p>The IMS resource adapter sends the client request to IMS through IMS Connect and does not expect a response from IMS. With a SYNC_SEND interaction, the client does not need to synchronously receive a response from IMS. SYNC_SEND is supported on both shareable and dedicated persistent socket connections and is only allowed with commitMode 0 interactions. If the interactionVerb is set to SYNC_SEND, execution timeout and socket timeout values are ignored. Note: imsRequest type 2 is not allowed with SYNC_SEND and will generate an exception.</p>

Value	Named constant in IMSInteractionSpecProperties	Description
1	SYNC_SEND_RECEIVE	<p>The execution of an IMS Interaction sends a request to IMS and receives a response synchronously. A typical SYNC_SEND_RECEIVE interaction is the running of a non-conversational IMS transaction in which an input record (the IMS transaction input message) is sent to IMS and an output record (the IMS transaction output message) is returned by IMS. SYNC_SEND_RECEIVE interactions are also used for the iterations of a conversational IMS transaction. A conversational transaction requires commitMode 1. A non-conversational transaction can run using either commitMode 1 or commitMode 0. If commitMode 0 is used on a dedicated persistent socket, a value for the clientID property of IMSConnectionSpec must be provided. If commitMode 0 is used on a shareable persistent socket, a value for the clientID property of IMSConnectionSpec must not be provided.</p>
3	SYNC_END_CONVERSATION	<p>If the application executes an interaction with interactionVerb set to SYNC_END_CONVERSATION, the IMS resource adapter sends a message to force the end of an IMS conversational transaction.</p> <p>The IMSInteractionSpec property, commitMode, and the IMSConnectionSpec property, clientID, do not apply when SYNC_END_CONVERSATION is provided for interactionVerb.</p>

Value	Named constant in IMSInteractionSpecProperties	Description
4	SYNC_RECEIVE_ASYNCOUTPUT	<p>interactionVerb SYNC_RECEIVE_ASYNCOUTPUT is valid on both shareable persistent and dedicated persistent socket connections. SYNC_RECEIVE_ASYNCOUTPUT is used to retrieve asynchronous output that was not delivered. When SYNC_RECEIVE_ASYNCOUTPUT is used on a dedicated persistent socket, a value must be provided for the clientID property of IMSConnectionSpec.</p> <p>A SYNC_RECEIVE_ASYNCOUTPUT interaction on a shareable persistent socket connection must be in the same application as the original SYNC_SEND or SYNC_SEND_RECEIVE interaction and must use the same shareable persistent connection. This primarily occurs following execution timeout.</p> <p>With this type of interaction, the Java client can only receive a single message. If there are no messages in the IMS OTMA Asynchronous Queue for the clientID when the request is made, no further attempts are made to retrieve the message. No message is returned and a timeout will occur after the length of time specified in the executionTimeout property of the SYNC_RECEIVE_ASYNCOUTPUT interaction.</p>

Value	Named constant in IMSInteractionSpecProperties	Description
5	SYNC_RECEIVE_ASYNCOUPTPUT_SINGLE_NOWAIT	<p>interactionVerb</p> <p>SYNC_RECEIVE_ASYNCOUPTPUT_SINGLE_NOWAIT is valid on both shareable and dedicated persistent socket connections. It is used to retrieve asynchronous output.</p> <p>A</p> <p>SYNC_RECEIVE_ASYNCOUPTPUT_SINGLE_NOWAIT</p> <p>interaction on a shareable persistent socket connection must be in the same application as the original SYNC_SEND or SYNC_SEND_RECEIVE interaction and must use the same shareable persistent connection. This primarily occurs following execution timeout.</p> <p>With this type of interaction, the Java client can only receive one single message. If there are no messages in the IMS OTMA Asynchronous Queue for the clientID when the request is made, no further attempts will be made to retrieve the message. No message will be returned and a timeout will occur after the length of time specified in the executionTimeout property of the</p> <p>SYNC_RECEIVE_ASYNCOUPTPUT_SINGLE_NOWAIT</p> <p>interaction.</p> <p>Note: The interactionVerbs, SYNC_RECEIVE_ASYNCOUPTPUT and</p> <p>SYNC_RECEIVE_ASYNCOUPTPUT_SINGLE_NOWAIT</p> <p>, perform the same function. However, it is recommended to use</p> <p>SYNC_RECEIVE_ASYNCOUPTPUT_SINGLE_NOWAIT</p> <p>with WebSphere Application Developer Integrated Edition 5.1.1 with the IMS resource adapter 9.1.0.1.2 or 2.2.4.</p>

Value	Named constant in IMSInteractionSpecProperties	Description
6	SYNC_RECEIVE_ASYNCOUPTUT_SINGLE_WAIT	<p>interactionVerb</p> <p>SYNC_RECEIVE_ASYNCOUPTUT_SINGLE_WAIT is used to retrieve asynchronous output. It is valid on both shareable and dedicated persistent socket connections.</p> <p>A</p> <p>SYNC_RECEIVE_ASYNCOUPTUT_SINGLE_WAIT</p> <p>interaction on a shareable persistent socket connection must be in the same application as the original SYNC_SEND or SYNC_SEND_RECEIVE interaction and must use the same shareable persistent connection. This primarily occurs following execution timeout.</p> <p>With this type of interaction, the Java client can only receive one single message. If there are no messages in the IMS OTMA Asynchronous Queue for the clientID when the request is made, IMS Connect waits for OTMA to return a message. IMS Connect waits the length of time specified in the executionTimeout property of the</p> <p>SYNC_RECEIVE_ASYNCOUPTUT_SINGLE_WAIT</p> <p>interaction before returning an exception.</p>

The J2EE Connection Architecture (JCA) values SYNC_RECEIVE (2) is not currently supported.

ltermName

The LTERM name used to override the value in the LTERM field of the IMS application program's I/O PCB. See the *IMS Connect User's Guide and Reference* (SC27-0946-23) for a description of how to use the LTERM override.

The value of this property can be set if the client application wants to provide an LTERM override name. This name will be in the IMS application program's I/O PCB, with the intent that the IMS application will make logic decisions based on this override value.

mapName

The mapName field typically contains the name of a Message Format Service (MFS) control block. MFS is the component of IMS that performs online formatting of transaction input and output messages. Since IMS Connect uses IMS OTMA to access IMS, MFS online formatting is bypassed. However, the mapName field can still be used by a Java application to input the name of an MFS control block to an IMS

application program or to retrieve the name of an MFS control block provided by an IMS application program.

On input, typically the value of the mapName property is the name of an MFS Message Output Descriptor, or "MOD". The MOD name will be provided to the IMS application program in the I/O PCB.

On output, the value of the mapName property is the name of an MFS Message Output Descriptor, or "MOD". This is the MOD name that the IMS application program specified when inserting the transaction output message to the I/O PCB.

Note: The mapName field should not be used by Java applications that use an enterprise service whose input and output messages are generated by WebSphere Studio MFS support.

purgeAsyncOutput

This is an input property. This property determines whether or not IMS Connect purges undelivered output.

This property is only valid for interactions on shareable persistent socket connections that use IMS interaction verb SYNC_SEND_RECEIVE. It is not valid for any interactions on dedicated persistent socket connections. It applies to commit mode 0 interactions. It does not apply to commit mode 1 interactions. However, if a commit mode 1 interaction executes a program-to-program switch, the spawned program will run commit mode 0 and therefore the property will apply.

If the purgeAsyncOutput property is not specified on a SYNC_SEND_RECEIVE interaction on a shareable persistent socket connection, the default is TRUE and the following output messages are purged:

- Undelivered output message inserted to the I/O PCB by the primary IMS application program.
- Output messages inserted to the I/O PCB by secondary IMS application programs invoked by a program to program switch.

reRoute

This is an input property.

This property is only valid for interactions on shareable persistent socket connections that use IMS interaction verb SYNC_SEND_RECEIVE. It is not valid for any interactions on dedicated persistent socket connections. It applies to commit mode 0 interactions. It does not apply to commit mode 1 interactions. However, if a commit mode 1 interaction executes a program-to-program switch, the spawned program will run commit mode 0 and therefore the property will apply. This property determines if undelivered output is to be rerouted to a named destination specified in the reRouteName field. If reRoute is TRUE, the asynchronous output is not queued to the TPIPE of the generated clientID. Instead, the asynchronous output is queued to the destination specified in the reRouteName field. The default value for reRoute is FALSE.

If both reRoute and purgeAsyncOutput are set to TRUE, an exception is thrown.

reRouteName

This property provides the name of the destination to which asynchronous output is queued. If reRoute is TRUE, this property provides the named destination. If reRoute is FALSE, the reRouteName property is ignored.

If the `reRoute` property is set to `TRUE`, and no `reRouteName` is provided, the value for the `reRouteName` property is:

1. The value specified in the IMS Connect configuration file.
2. If no value is specified in the IMS Connect configuration file, the value `"HWS$DEF"` is used.

Valid values for the `reRouteName` property:

- Must be a string of 1 to 8 alphanumeric (A-Z, 0-9) or special (@, #, \$) characters.
- Must not start with the character string, `"HWS"`.
- Must not be an IMS Connect port number.
- If lowercase letters are provided, the letters will be changed to uppercase.

The property, `reRouteName`, is only valid for `SYNC_SEND_RECEIVE` interactions on shareable persistent socket connections. It is not valid for any interactions on dedicated persistent socket connections.

required

Leave this field empty.

Chapter 4. Security

The topics in this section describe security issues for the IMS resource adapter. The topics included are:

IMS resource adapter security

Information in an Enterprise Information System (EIS) such as IMS must be protected from unauthorized access. The J2EE Connector Architecture (J2C) specifies that the application server and the EIS must collaborate to ensure that only authenticated users are able to access an EIS. The J2C security architecture extends the end-to-end security model for J2EE-based applications to include integration with EISs.

EIS sign-on

The J2C security architecture supports a user ID and password authentication mechanism specific to an EIS. For more information, see Java 2 Connector security in the WebSphere Application Server documentation.

The user ID and password for the target EIS is supplied either by the application component (component-managed sign-on) or by the application server (container-managed sign-on).

For IMS Connector for Java, IMS is the target EIS. The security information is passed to the IMS resource adapter, which then passes it to IMS Connect. IMS Connect uses this information to perform user authentication and passes it on to IMS OTMA which also uses this information to verify authorization to access IMS.

In a typical environment, the IMS resource adapter passes on the security information (user ID, password, and optional group name) that it receives to IMS Connect in an IMS OTMA message. Depending on its security configuration, IMS Connect may then call the host's Security Authorization Facility (SAF).

- For WebSphere Application Server on distributed platforms or z/OS with TCP/IP, using either component-managed or container-managed sign-on:
 - If RACF=Y is set in the IMS Connect configuration member or if the IMS Connect command SETRACF ON has been issued, IMS Connect calls the SAF to perform authentication using the user ID and password passed by IMS Connector for Java in the OTMA message. If authentication succeeds, the user ID, groupname, and UTOKEN returned from the IMS Connect call to the SAF are passed to IMS OTMA for use in verifying authorization to access IMS.
 - If RACF=N is set in the IMS Connect configuration member or if the IMS Connect command SETRACF OFF has been issued, IMS Connect does not call the SAF. However, the user ID and groupname are still passed to IMS OTMA for use in verifying authorization to access IMS.
- For WebSphere Application Server on z/OS with Local Option, using either component-managed or container-managed sign-on:
 - Regardless of the RACF® setting in the IMS Connect configuration member or in the SETRACF command, IMS Connect does not call the SAF, because authentication has already been performed by WebSphere Application Server

for z/OS. The UTOKEN generated when WebSphere Application Server for z/OS calls RACF is passed to IMS for use in verifying authorization to access IMS.

- WebSphere Application Server for z/OS can be configured to use the user identity associated with the thread of execution to authenticate a user. The application server creates and passes the UTOKEN representing the user identity to the IMS resource adapter. The IMS resource adapter then passes the token to IMS Connect for sign-on to IMS. For information about the RunAs Identity support in WAS, consult the security documentation for WebSphere Application Server z/OS.

The level of authorization checking performed by IMS is controlled by the IMS command, /SECURE OTMA. See the *IMS OTMA Guide and Reference* for more information about this command.

Java2 Security Manager

The IMS resource adapter works with the WebSphere Application Server Java2 Security Manager. Components such as resource adapters must be authorized to perform protected tasks, such as making socket calls. The IMS resource adapter is already authorized to perform these tasks. No action is required by the application component.

See the Managing secured applications in the WebSphere Application Server documentation for more information about the Java2 Security Manager.

Component-managed EIS sign-on

When you specify `<res-auth>Application</res-auth>` in the deployment descriptor of your application, component-managed EIS sign-on is used. Your application (the component) should provide the security information (user ID, password, and optional group name) used for EIS sign-on:

- If your application uses the J2EE Connector Architecture Common Client Interface (CCI), it performs component-managed sign-on by first populating an `IMSConnectionSpec` object with the security information. Then, when the application establishes a connection to IMS, it passes the `IMSConnectionSpec` object as a parameter of the `IMSConnectionFactory.getConnection` method. The IMS resource adapter uses this security information for the sign-on to IMS.
- If your application is a service-based application built by WebSphere Studio, the security information is passed as application input data. For information about exposing the `InteractionSpec` and `ConnectionSpec`, see `InteractionSpec` and `ConnectionSpec` properties as data. For specific information about how to pass the `IMSConnectionSpec` properties user ID, password, and group name as application input data for the IMS resource adapter, see *Creating an application to run a commit mode 0 transaction*.

If your application does not use one of the above methods to provide security information, WebSphere Application Server will obtain the security information from the J2C connection factory's custom properties. **Note:** If you specified a component-managed JAAS Authentication alias while setting up your connection factory, the user ID and password in the alias will override the `userName` and `password` values in the connection factory custom properties during the start-up of the WebSphere Application Server.

Configuring component-managed EIS sign-on

The following steps explain how to configure component-managed EIS sign-on for an EJB.

1. Set the `<res-auth>` directive to **Application**.
 - a. In WebSphere Studio, open your **EJB module** with the Deployment Descriptor Editor.
 - b. Click the **References** tab and select **Application** in the **Authentication** field, which maps to the `<res-auth>` directive.
 - c. When you close the EJB Deployment Descriptor Editor and click **Yes** to save your changes, the following code is added to the deployment descriptor of your EJB:

```
<res-auth>Application</res-auth>
```

2. Typically, component-managed sign-on does not require further configuration because the security information is provided by the application in the `IMSCConnectionSpec` object. However, if your application does not provide an `IMSCConnectionSpec` object, or the user ID is not specified in the `IMSCConnectionSpec` object that is provided, the IMS resource adapter will obtain default security values from the connection factory used by your application.

The default security values for a connection factory can be provided in two ways:

- a. When you use a component-managed authentication alias.
 - To use a component-managed authentication alias, you must define a JAAS authentication alias.
 - 1) In the **Server Configuration** view, open the editor for your server and select the **Security** tab.
 - 2) Select the **Add** button beside the JAAS Authentication Entries list.
 - 3) Enter an alias name, your user ID, password, and optional description. Select **OK**.
 - 4) Save your changes and close the editor.
 - Select the JAAS authentication alias for the Component-managed authentication alias property of the J2C Connection Factory used by your application. You can do this when you first create the connection factory or later by editing the connection factory. To edit the connection factory:
 - 1) Open the editor for your server and select the **J2C** tab.
 - 2) On the **J2C Options** page, select the connection factory and the associated **Edit** button.
 - 3) Select the alias from the drop-down list of the **Component-managed authentication alias** field.
 - 4) Select **OK**.

The user ID and password associated with the component-managed authentication alias will be used to set (over override if applicable) the default values in the custom properties of the associated connection factory during application server startup.

- b. When you create a connection factory.
 - If you do not assign a valid JAAS authentication alias to the component-managed authentication alias field of your J2C connection factory, you can assign values for the `userName`, `password`, and `groupName` fields on the J2C options page of your J2C connection factory.

- For instructions on creating a connection factory, see Adding a J2C connection factory and Connection Properties. Using a component-managed authentication alias is preferred over specifying values in the custom properties of your J2C connection factory because the component-managed authentication alias provides greater security for the user ID and password.

Note: The process for configuring component-managed sign-on in WebSphere Application Server is similar to the process for WebSphere Studio. For information on this topic as it relates to WebSphere Application Server, see:

- Managing J2C authentication data entries
- Java 2 Connector authentication data entry settings
- Configuring J2C connection factories in the administrative console
- J2C connection factory settings

Container-managed EIS sign-on

When `<res-auth>Container</res-auth>` is specified in the deployment descriptor of the application, container-managed EIS sign-on will be used. When container-managed sign-on is used, your application does not programmatically provide the security information. Instead, the application server (the container) provides the security information (user ID and password). One way to accomplish this when using `DefaultPrincipalMapping`, is to provide values for the user ID and password to be used by the application server as follows:

- Define a JAAS Authentication alias, associating the user ID and password you wish to use for EIS sign-on with the alias
- Associate this alias with the J2C connection factory used by your application

For TCP/IP, the application server passes the security information in the alias to the IMS resource adapter. The IMS resource adapter passes the security information to IMS Connect for authentication. IMS Connect authenticates the user and passes the security information for sign-on to IMS. If IMS Connect cannot authenticate the user, a security failure is returned to the IMS resource adapter which, in turn, passes an exception back to the application.

For Local Option, a z/OS-only feature in which both the server and WebSphere Application Server are running in the same MVS image, the application server authenticates the user based on the security information defined in the container-managed alias. The application server creates and passes a UTKEN representing the authenticated user to the IMS resource adapter. The IMS resource adapter then passes the UTKEN to IMS Connect which in turn passes it on to IMS OTMA for use in signing on to IMS.

Alternatively, when using Local Option communications, you can specify in the application server configuration that the user identity associated with the current thread of execution is to be used by the application server when performing user authentication. In this case, you do not specify a JAAS container-managed authentication alias in the J2C connection factory used by your application. This option is only available if you are using Local Option communications.

Note: When using container-managed sign-on, if your application does pass security information to the IMS resource adapter using the `userName`, `password` or `groupName` properties of `IMSConnectionSpec`, it is ignored. However, if you pass

other information in the `IMSCConnectionSpec` object, such as `clientID` used with commit mode 0 interactions, this information will be used by the IMS resource adapter.

Configuring container-managed EIS sign-on

The following steps explain how to configure container-managed EIS sign-on for an EJB.

1. Set the `<res-auth>` directive to **Container**.
 - a. In WebSphere Studio, open your **EJB module** with the Deployment Descriptor Editor.
 - b. Click the **References** tab. Select **Container** in the **Authentication** field, which maps to the `<res-auth>` directive.
 - c. When you close the EJB Deployment Descriptor Editor and click **Yes** to save your changes, the following code is added to the deployment descriptor of your EJB:

```
<res-auth>Container</res-auth>
```
2. Specify a method of providing the user ID and password that you want the application server. To use a JAAS authentication alias to provide the user ID and password that you can use for EIS sign-on, complete the following steps:
 - a. In the **Server Configuration** view, open the editor for your server. Select the **Security** tab.
 - b. Select the **Add** button beside the JAAS Authentication Entries list.
 - c. Enter an alias name, your user ID, password, and optional description. Select **OK**.
 - d. Save your changes and close the editor.
3. Select the JAAS authentication alias for the Container-managed authentication alias property of the J2C connection factory used by your application. You can do this when you first create the connection factory or later by editing the connection factory. To edit the connection factory:
 - a. Open the editor for your server and select the **J2C** tab.
 - b. On the **J2C Options** page, select the connection factory and the associated **Edit** button.
 - c. Select the alias from the drop-down list of the **Container-managed authentication alias** field.

If you do not specify a method to the application server for providing sign-on information such as using the user ID associated with the thread of execution or assigning a JAAS authentication alias to the container-managed authentication alias property of your J2C connection factory on the J2C options page of your server, you will receive an exception if you attempt to execute a container-managed interaction using that connection factory.
 - d. Select **OK**.

For instructions on creating a connection factory, see [Adding a J2C connection factory and Connection Properties](#).

Note: The process for configuring container-managed sign-on in WebSphere Application Server is similar to the process for WebSphere Studio. For information on this topic as it relates to WebSphere Application Server, see:

- [Managing J2C authentication data entries](#)
- [Java 2 Connector authentication data entry settings](#)

- Configuring J2C connection factories in the administrative console
- J2C connection factory settings

Overview of secure socket layer (SSL)

With the evolution of e-business, data security has become very important for Internet users. The Secure Socket Layer (SSL) protocol ensures that the transfer of sensitive information over the Internet is secure. SSL protects information from:

- Internet eavesdropping
- Data theft
- Traffic analysis
- Data modification
- Trojan horse browser /server

One way IMS Connector for Java communicates with IMS Connect is through TCP/ IP sockets. If IMS Connector for Java uses TCP/ IP, SSL can be used to secure the TCP/ IP communication between the two entities. The SSL support provided by IMS Connector for Java, along with the support provided by IMS Connect, uses a combination of public and private keys along with symmetric key encryption schemes to achieve client and server authentication, data confidentiality, and integrity. SSL rests on top of TCP/ IP communication protocol and allows an SSL-enabled server to authenticate itself to an SSL-enabled client and vice versa. Once authentication is complete, the server and client can establish an encrypted connection that also preserves the integrity of the data.

For SSL support when running in a WebSphere environment, IMS Connector for Java uses the IBM® implementation of Java Secure Socket Extension (IBM JSSE). The SSL library is included in WebSphere Studio Application Developer Integration Edition and in WebSphere Application Server.

SSL concepts

Certificate

A digital certificate is a digital document that validates the identity of the certificate's owner. A digital certificate contains information about the individual, such as their name, company, and public key. The certificate is signed with a digital signature by the Certificate Authority (CA), which is a trustworthy authority.

Certificate authority

A Certificate Authority (CA) is a trusted party that creates and issues digital certificates to users and systems. The CA, as a valid credential, establishes the foundation of trust in the certificates.

Certificate management

Certificates and private keys are stored in files called keystores. A keystore is a database of key material. Keystore information can be grouped into two categories: key entries and trusted certificate entries. The two entries can be stored separately for security purposes.

Keystore

A keystore holds key entries, such as the private key of the user, IMS Connector for Java

Truststore

A truststore is a keystore that holds only certificates that the user trusts. An entry should be added to a truststore only if the user makes a decision to trust that entity. An example of a truststore entry would be the certificate for the target IMS Connect.

For convenience, IMS Connector for Java allows the user to store key entries and trusted certificate entries in either the keystore or the truststore. The user may still choose to store them separately. IMS Connector for Java supports only the keystore type "JKS" and X.509 certificates.

SSL process

The SSL protocol consists of server authentication, client authentication (optional) followed by an encrypted conversation. The following scenario steps through the SSL process.

Server authentication

A customer, Alice, wants to send sensitive information to her bank. She needs confirmation that the server to which she is sending her personal information is really her bank. SSL-server authentication allows a client to confirm a server's identity. SSL-enabled client software uses standard techniques of public-key cryptography to ensure that a server's certificate and public ID is valid and that the certificate and ID was issued from one of the client's list of trusted certificate authorities (CA).

Client authentication

Likewise, the bank needs confirmation that it is really Alice who is sending information. SSL-client authentication allows a server to confirm a client's identity. Using the same techniques used for server authentication, SSL-enabled server software verifies that a client's certificate and public ID is valid and that the certificate and ID was issued by one of the server's list of trusted certificate authorities (CA).

SSL handshake

Both Alice and the bank store their certificates and private keys in keystores. The actual SSL session between Alice and the bank is established by following a handshake sequence between client and server. The sequence may vary depending on whether the server is configured to provide a server certificate or to request a client certificate, and which cipher suites are being used. A cipher is an encryption algorithm. The SSL protocol determines how the client and server negotiate the cipher suites to authenticate one another, to transmit certificates, and to establish session keys. Some of the algorithms used in cipher suites include:

- DES - Data Encryption Standard
- DSA - Digital Signature Algorithm
- KEA - Key Exchange Algorithm
- MD5 - Message Digest algorithm
- RC2 and RC4 - Rivest encryption ciphers

- RSA - A public key algorithm for both encryption and authentication
- RSA key exchange - A key-exchange for SSL based on the RSA algorithm
- SHA-1 - Secure Hash Algorithm
- SKIPJACK - A classified symmetric-key algorithm implemented in FORTEZZA-compliant hardware
- Triple-DES - DES applied three times.

SSL 2.0 and SSL 3.0 protocols support overlapping sets of cipher suites. Administrators can enable or disable any of the supported cipher suites for both clients and servers. When a particular client and server exchange information during the SSL handshake, the client and server identify the strongest enabled cipher suites that they have in common and use them for the SSL session.

Transport Layer Security, Version 1 (TLS V1) is the successor to SSL 3.0 protocol. IMS Connector for Java **only** supports TLS V1. There are no backward compatibility issues.

Using secure socket layer (SSL) support

The following table provides a high level description of how IMS Connector for Java and IMS Connect SSL support is set up and configured. Follow the steps in the order outlined below:

SSL Client (IMS Connector for Java)	SSL Server (IMS Connect)
	1. Decide if client authentication is required. If client authentication is not required, skip to Step 5.
2. If client authentication is required, obtain signed certificates and private key.	
3. If client authentication is required, create a keystore and insert the client's private key and certificate. For more detail, see the description below.	
	4. If client authentication is required, insert the client's public key certificate into the keyring. See <i>IMS Connect User's Guide</i> (SC27-0946-03) for more information.
5. Create a truststore (another keystore) and insert the Server's public key certificate.	
	6. Decide which IMS Connect SSL port to use. Set up the IMS Connect and SSL Configuration members with the appropriate values. For more information about setting up the configuration file, see <i>IMS Connect User's Guide</i> (SC27-0946).
7. Obtain the IMS Connect SSL port number.	
8. Set up the connection factory with the appropriate SSL parameters. For more detail, see the description below.	
9. Bind the application to the SSL connection factory.	

Creating the keystore or truststore for the client

For the client and server to authenticate one another, you must provide a JKS keystore with valid X.509 certificates at both the client and server end. If client authentication by the server is not required, it is not necessary to create the client certificate and add it to the server's keyring. There are several tools available for managing the keystore. To provide a JKS keystore at both the client and server end, you must perform the following steps:

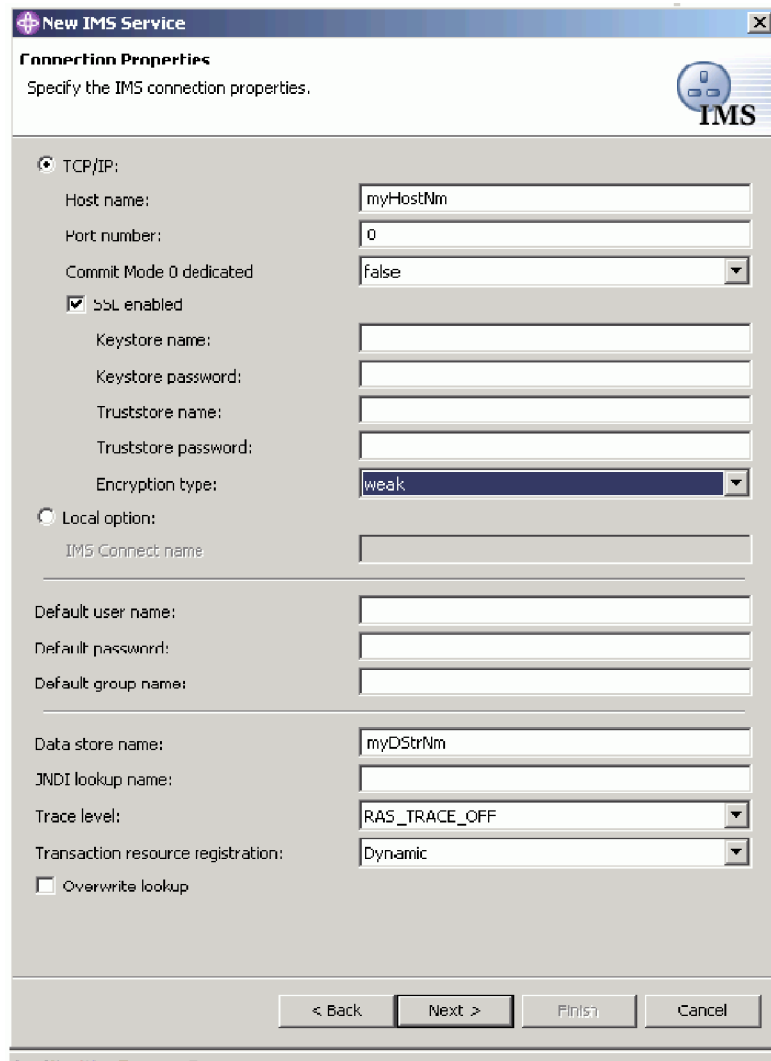
- To set up the Client, create a certificate and have it signed by a Certificate Authority (for example, VeriSign), or create your own CA using software such as OpenSSL to sign your own certificate.
- To create a keystore, use a key management tool such as Ikeyman or Keytool. After the keystore is created, import the client certificate (if one is available) into the keystore.
- To create a truststore, create another keystore and import the server certificate.
Note: If you want to create only one keystore, import the server certificate into the same keystore as the client certificate.

SSL configuration

The SSL properties are used to create a secure SSL connection between a Java client application and IMS Connect. See Connection properties for a description of the values that have to be provided to IMS Connector for Java.

There are two ways to set up SSL properties:

1. You can use the tooling in WebSphere Studio Application Developer Integration Edition (development environment). WebSphere Studio Application Developer Integration Edition maps or binds the connection factory resource reference in the Java client application, which is installed on WebSphere Application Server, to the SSL-configured connection factory by providing the JNDI name of the connection factory. The following figure displays the IMS connection properties interface:



New IMS Service

Connection Properties
Specify the IMS connection properties.

TCP/IP:

Host name: myHostNm

Port number: 0

Commit Mode 0 dedicated: false

☒ SSL enabled

Keystore name:

Keystore password:

Truststore name:

Truststore password:

Encryption type: weak

Local option:

IMS Connect name:

Default user name:

Default password:

Default group name:

Data store name: myDStrNm

JNDI lookup name:

Trace level: RAS_TRACE_OFF

Transaction resource registration: Dynamic

☐ Overwrite lookup

< Back Next > Finish Cancel

2. You can use the property sheet in WebSphere Application Server (runtime environment).

The connection factory created by the client is used during the runtime environment to set up a secure socket connection. The following figure displays the SSL connection factory property sheet:

HostName	myHostNm	TCP/IP host name of the target IMS Connect	false
PortNumber	0	Target TCP/IP port number of IMS Connect	false
DataStoreName	myDStrNm	Name of the target IMS datastore	false
SSLEnabled	FALSE	Indicates if SSL is enabled for this connection factory	false
SSLEncryptionType	Weak	The type of cipher suite to be used for encryption	false
SSLKeyStoreName	-	Name (full path) of SSL keystore for client certificates/private keys	false
SSLKeyStorePassword	-	Password of SSL keystore for client certificates/private keys	false
SSLTrustStoreName	-	Name (full path) of SSL keystore for trusted certificates	false
SSLTrustStorePassword	-	Password of SSL keystore for trusted certificates	false
CMODedicated	FALSE	Indicates if sockets are dedicated to specific CMQ clients	false
UserName	-	Default name of the user to be authorized	false
Password	-	Default password of the user	false

Note: Informational messages and warnings can be found in the **trace.log** file generated by the server.

At runtime, when the Java client application executes an interaction with IMS, the interaction flows on a secure (SSL) connection to IMS Connect. The following steps are transparent to the Java client application. The IMS resource adapter interacts with IMS Connect using the SSL protocol as follows:

- IMS Connector for Java initiates a connection by sending a client hello. The server replies with a server hello and its certificate.
- If the server does not require client authentication, the client authenticates the server's certificate using the server's public key from its truststore. If authentication is successful, the SSL handshake is completed. A session key has been established at both ends.
- If the server does require client authentication, the client authenticates the server's certificate using the server's public key from its truststore. If this authentication is successful, a client certificate is sent from the client's keystore. If this certificate is authenticated successfully by the server, the SSL handshake is completed. A session key has been established at both ends.
- The client and server are then ready to send and receive encrypted data.

Chapter 5. Commit mode processing

The topics in this section describe some of the different processing models that a Java client can use with the IMS resource adapter. The topics included are:

Overview of commit mode processing

Commit mode refers to the type of commit processing performed by IMS. The Java client specifies the commit mode protocol to be used when it submits a transaction request to IMS. There are two types of commit mode processing supported by IMS Connect and IMS: commit mode 0 (commit-then-send) where IMS commits the IMS database changes and then sends the output to the client and commit mode 1 (send-then-commit) where IMS sends the output to the client and then commits the database changes.

Associated with the commit mode protocols, IMS Connect and IMS also support three synchronization levels (synch levels): NONE, CONFIRM, and SYNCPT. All three synch levels can be used with commit mode 1. Only CONFIRM can be used with commit mode 0. However, IMS Connector for Java does not currently support commit mode 1, synch level CONFIRM.

Currently, the synchronization level is not set by the Java client. IMS Connector for Java automatically provides the synchronization level when communicating with IMS Connect.

IMS Connector for Java supports the following combinations:

- Commit mode 1 with synch level NONE
This combination is used for non-transactional interactions. For non-conversational applications, use the SYNC_SEND_RECEIVE interaction. For conversational applications, use SYNC_SEND_RECEIVE or optionally, SYNC_END_CONVERSATION interaction.
- Commit mode 1 with synch level SYNCPT
This combination is used by IMS Connector for Java when participating in two-phase commit processing with IMS. For more information, see Global transaction support with two-phase commit.
- Commit mode 0 with synch level CONFIRM
This combination is used by IMS Connector for Java for non-transactional SYNC_SEND_RECEIVE, SYNC_SEND, SYNC_RECEIVE_ASYNCOUTPUT, SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT and SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT interactions.
Note: Commit mode 0 is only supported for non-conversational applications running on TCP/IP connections.

The synchronization level is not set by the Java client. IMS Connector for Java automatically provides the synchronization level when communicating with IMS Connect.

If the Java client submits a transaction request with commit mode 1 synch level NONE, IMS Connector for Java passes the request through IMS Connect to IMS. IMS processes this transaction and attempts to send the output message to the Java client. The Java client may receive the output message from the transaction or may

receive an exception. In either case, IMS will have already committed the changes to the database and discarded the output message of the IMS transaction.

Similarly, if the Java client sends a transaction with commit mode 0 synch level CONFIRM, the Java client may receive the output message from the transaction or may receive an exception. However, if the Java client receives an exception when commit mode 0 is used, the output may or may not be queued for later retrieval. Whether or not the output message that was not delivered to a Java client will be queued depends on the type of socket connection the Java client uses for the commit mode 0 interaction.

The type of exception also determines whether or not an output message is available for retrieval. For example, if the Java client receives an `IMSDFSMessageException` indicating that the transaction is stopped, the application was not run; therefore, there is no output message available for retrieval. However, if the transaction runs but the `executionTimeout` value expires before the output message is returned to IMS Connect, the Java client will receive an `EISSystemException` that an execution timeout has occurred. In this case, the output message will be queued to the appropriate IMS OTMA Asynchronous Output Queue or TPIPE for later retrieval.

Note: In IMS/OTMA terminology, a transaction pipe (TPIPE) is a logical connection between a client (IMS Connect) and the server (IMS/OTMA). For commit mode 0 interactions, the TPIPE is identified by the `clientID` used for the interaction. Each `clientID` used for a commit mode 0 transaction will have its own TPIPE. For commit mode 1 interactions, the TPIPE is identified by the IMS Connect port number used for the interaction. Therefore, each port will have a TPIPE which will be used for all clients running commit mode 1 interactions on that port.

Regardless of whether your Java client is running an IMS transaction with commit mode 1 or commit mode 0, the Java client specifies a value for the `interactionVerb` property of `IMSInteractionSpec`. If a commit mode 0 interaction is specified, the Java client may also have to provide a value for the `clientID` property of `IMSConnectionSpec`. `clientID` is a property of `IMSConnectionSpec` and identifies the IMS OTMA Asynchronous Output Queue or TPIPE where the recoverable output messages are placed. Whether or not a Java client provides a `clientID` for a commit mode 0 interaction depends on the type of socket connection being used by the Java client.

To retrieve output messages from a TPIPE, the Java client submits a request in which it specifies one of the values `SYNC_RECEIVE_ASYNCOUTPUT`, `SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT`, or `SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT` for the `interactionVerb` property of `IMSInteractionSpec` and a value for the `clientID` property of `IMSConnectionSpec`. For more information about asynchronous output support, see *Chapter 9: Protocols in IMS Connect Guide and Reference*.

In general, the `SYNC_RECEIVE_ASYNCOUTPUT`, `SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT`, or `SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT` interactions can be used to retrieve output messages queued for any `clientID`, regardless of how those messages were queued to the associated `clientID` - either as a result of a failed commit mode 0 transaction or from an IMS application that issued an insert to an ALTPCB (Alternate Program Communication Block). In the case of retrieving an output message from a failed commit mode 0 transaction, the `clientID` provided in

the IMSConnectionSpec for retrieval request must match the clientID that was specified on the failed commit mode 0 transaction.

If there is nothing in the OTMA Asynchronous Output Queue for that particular clientID, you will receive an execution timeout exception. The timeout exception can mean either that there are no messages in the queue or that the timeout value did not provide enough time for IMS Connect to retrieve the message from the queue. For both SYNC_RECEIVE_ASYNCOUTPUT, SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT, or SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT, as well as SYNC_SEND_RECEIVE interactions, executionTimeout is the length of time IMS Connect will wait for a response from IMS. If you do not specify an execution timeout value for a retrieval request, the default execution timeout value will be used. The default timeout value is the IMS Connect configuration member TIMEOUT value. The user may need to experiment with the execution timeout value, to ensure that output messages are returned for all types of interactions.

Commit mode processing and socket connections

All socket connections created by the IMS resource adapter are persistent. In other words, the same socket connection between IMS Connector for Java and IMS Connect can be serially reused for multiple interactions with IMS Connect. The socket connection will not be closed and reopened between interactions. There are two types of persistent sockets; shareable and dedicated.

Shareable Persistent Socket

The shareable persistent socket can be shared (serially reused) by multiple applications executing either commit mode 1 or commit mode 0 interactions. For an application executing a commit mode 0 interaction on a shareable persistent socket, the IMS resource adapter automatically generates a clientID with the prefix "HWS". This clientID represents and identifies the socket connection as well as the associated OTMA TPIPE. For this type of socket, only clientIDs generated by the IMS resource adapter are allowed. A user-specified clientID is not allowed with shareable persistent socket support.

Note: IMS application programs that insert messages to an alternate PCB must not use names beginning with "HWS" for the alternate PCBs.

Any output message that cannot be delivered to a Java client executing a commit mode 0 interaction on a shareable persistent socket can be queued for later retrieval. Also, any commit mode 1 or commit mode 0 interaction on a shareable persistent socket that spawns a program-to-program switch which invokes another commit mode 0 interaction resulting in secondary output, can be requeued for later retrieval. SYNC_RECEIVE_ASYNCOUTPUT, SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT, and SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT interactions are supported on shareable persistent sockets. To retrieve undelivered output messages that are queued in the IMS OTMA Asynchronous Hold Queue or TPIPE, the interaction verbs must be invoked within the same client application, because the same generated client ID that identifies the shareable socket connection and the associated OTMA TPIPE must be used.

On shareable persistent sockets, the undelivered output messages can be handled in more than one way. One way is to purge the undelivered output. To purge undelivered output messages, you must ensure the IMSInteractionSpec property

purgeAsyncOutput is TRUE. This input property determines if IMS Connect purges the undelivered I/O PCB output. The purgeAsyncOutput property is only valid with the SYNC_SEND_RECEIVE interaction verb. If the property is not specified on SYNC_SEND_RECEIVE, the default is TRUE.

Another option of handling undelivered output messages on shareable persistent sockets is rerouting the messages to another destination. You can reroute the undelivered output message to a different destination by setting the IMSInteractionSpec property, reRoute, to TRUE. This property is only valid for the SYNC_SEND_RECEIVE interaction verb. If reRoute is set to TRUE, the undelivered output message is queued to a named destination provided by the client application, which is specified on the reRouteName IMSInteractionSpec property. If the reRoute property is set to TRUE and no reRouteName is provided, the value of the reRouteName property is the value specified in the IMS Connect configuration file. If no value is specified in the IMS Connect configuration file, the default value HWS\$DEF is used.

Shareable persistent socket connections are created by an IMS Connection Factory with values for at least the following custom properties:

- Host name = TCP/IP host name of machine running IMS Connect
- Port number = associated port number
- Datastore name = name of target IMS
- CM0Dedicated = FALSE

FALSE is the default value for the endCM0Dedicated property and ensures that the connection factory will create shareable persistent socket connections.

Dedicated persistent socket

A dedicated persistent socket is used for Java applications executing commit mode 0 interactions only. It can be shared (serially reused) by multiple applications with the same user-specified clientID. For this type of socket, only interactions with user-specified clientIDs are allowed. A valid user-specified clientID:

- Must be a string of 1 to 8 alphanumeric (A-Z, 0-9) or special (@, #, \$) characters.
- Must not start with the character string, "HWS".
- Must not be an IMS Connect port number.
- If lowercase letters are provided, the letters will be changed to uppercase

A dedicated persistent socket means the socket connection is assigned to a specific clientID and will remain dedicated to that particular clientID until it is disconnected. SYNC_SEND_RECEIVE, SYNC_SEND, SYNC_RECEIVE_ASYNCOUTPUT, SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT, and SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT interactions are supported on dedicated persistent sockets.

SYNC_RECEIVE_ASYNCOUTPUT, SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT, and SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT interactions on dedicated persistent sockets enable client applications to retrieve messages that were placed on an IMS OTMA Asynchronous Output Queue as a result of a failed commit mode 0 interaction, from an IMS application that issued an insert to an ALTPCB (Alternate Program Communication Block), or from the reroute of the output from a transaction that was executed on a shareable connection factory. To retrieve the

messages, the client application must provide the clientID, which represents the TPIPE that has asynchronous output messages queued. Interactions on dedicated persistent sockets that have undelivered output messages cannot be rerouted or purged.

Dedicated persistent socket connections are created by an IMS Connection Factory with values for at least the following custom properties:

- Host name = TCP/IP host name of machine running IMS Connect
- Port number = associated port number
- Datastore name = name of target IMS
- CM0Dedicated = TRUE

A value of TRUE for the endCM0Dedicated property ensures that the connection factory will create dedicated persistent socket connections.

Note: If you have more than one connection factory configured to create dedicated persistent sockets to the same IMS Connect instance, only one connection factory can dedicate a socket to a particular clientID at one time. For example, if the first connection factory successfully creates a socket connection dedicated to clientID, CLIENT01; the second connection factory will receive the following exception if it tries to create a socket connection dedicated to CLIENT01 while the socket connection created by the first connection factory is still connected to IMS Connect:

```
javax.resource.spi.EISSystemException: IC00001E:  
com.ibm.connector2.ims.ico.IMSTCIPManagedConnection@23766050.processOutputOTMAMsg  
(byte [], InteractionSpec,Record) error. IMS Connect returned error: RETCODE=[8],  
REASONCODE=[DUPECLNT].  
Duplicate client ID was used; the client ID is currently in use.
```

Releasing Persistent Sockets

A TCP/IP connection between IMS Connector for Java and IMS Connect is persistent; in other words it remains open as long as IMS Connector for Java or IMS Connect does not disconnect it due to an error. This is the case for both a shareable persistent socket connection and a dedicated persistent socket connection. However, in the case of a dedicated persistent socket connection, the socket connection can only be used by interactions that have the same clientID that was used to establish the connection. The number of socket connections will increase as new clientIDs are used for interactions on dedicated persistent socket connections.

If you have the Max connections property set to a non-zero value and you also have a non-zero value for the Connection timeout property, when the MaxConnections is reached and all the connections are in use, the application will get a ConnectionWaitTimeoutException after the seconds specified in Connection timeout have elapsed. This is standard behavior for WebSphere Application Server. The ConnectionWaitTimeoutException applies to both dedicated persistent sockets and shareable persistent sockets.

However, if MaxConnections has been reached and one of the persistent socket connections is currently not in use, then WebSphere Application Server will disconnect that socket in order to respond to the request to create a new persistent socket connection. This also is standard behavior for the WebSphere Application Server and applies to both dedicated and shareable persistent sockets.

SYNC_SEND_RECEIVE programming model

To run a transaction in IMS, your Java application executes a SYNC_SEND_RECEIVE interaction. Your application provides a value of SYNC_SEND_RECEIVE for the interactionVerb property and a value of 0 or 1 for the commitMode property of the IMSInteractionSpec object used by the execute method. However, the SYNC_SEND_RECEIVE interaction processing is different for shareable and dedicated persistent socket connections.

Shareable persistent socket processing model

The following scenarios describe the SYNC_SEND_RECEIVE interaction on a shareable persistent socket during normal processing, error processing, and execution timeout. These steps apply for both commit mode 1 and commit mode 0.

- Normal processing scenario

The IMS resource adapter, with the application server, obtains either an available connection from the connection pool or creates a new connection. The IMS resource adapter, as part of initializing a new connection generates a clientID for the connection. The generated clientID identifies the socket connection, and in the case of commit mode 0 interactions, the TPIPE and associated OTMA Asynchronous Hold Queue.

The IMS resource adapter ensures that a socket is associated with the connection and sends the request with input data to IMS Connect using that socket. IMS Connect then sends the message to IMS, where IMS runs the transaction and returns the output message.

For commit mode 0 interactions, on receiving the output message, the IMS resource adapter internally sends an ACK message to IMS which signals IMS to discard the output from the IMS queue. When the client application closes the connection or terminates, the connection is returned to the connection pool for reuse by other commit mode 0 or commit mode 1 interactions.

- Error processing scenario

All errors result in a resource exception being thrown to the client application. In addition, some errors result in the socket being disconnected by IMS Connect. In the case of commit mode 0 interactions, an exception means the output message cannot be delivered to the client application. However, following exceptions undelivered output messages for commit mode 0 interactions on shareable persistent socket connections can be retrieved if the SYNC_SEND_RECEIVE interaction specified that undelivered output should be rerouted to a specific destination. To have an undelivered output message rerouted to a specific destination, the following additional properties must be specified in the IMSInteractionSpec object passed on the SYNC_SEND_RECEIVE interaction:

- The purgeAsyncOutput property must be set to FALSE so that undelivered output is not purged
- The reRoute property must be set to TRUE and a reroute destination specified in the RouteName property

To retrieve undelivered output from a reroute destination, a separate client application issues a SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT or SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT interaction on a dedicated persistent socket connection, providing the reroute destination as the clientID of the interaction.

Note: The default value of the purgeAsyncOutput property is TRUE.

When purgeAsyncOutput is TRUE, the following output messages are purged:

- Undelivered output message inserted to the I/O PCB by the primary IMS application program.
- Output messages inserted to the I/O PBC by secondary IMS application programs invoked by program to program switch.

A value of FALSE for the PurgeAsyncOutput property should only be used if the reroute destination is specified.

- ExecutionTimeout scenario

If an execution timeout occurs, the socket connection remains open but the output message is not delivered to the client application. However, following an execution timeout exception, undelivered output messages for commit mode 0 interactions on shareable persistent socket connections can be retrieved in either of the following two ways:

- The same client application that issued the SYNC_SEND_RECEIVE interaction can issue a SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT or SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT interaction.
- The undelivered output message can be rerouted to a specific destination as described in the error processing scenario above.

When the client application closes the connection or terminates, the connection is returned to the connection pool so it can be reused by other commit mode 0 or commit mode 1 interactions.

Dedicated persistent socket processing model

Dedicated persistent socket connections can only be used for commit mode 0 interactions. The following scenarios describe the commit mode 0

SYNC_SEND_RECEIVE interaction on a dedicated persistent socket during normal processing, error processing, and execution timeout.

- Normal processing scenario

Under normal circumstances, when a commit mode 0 SYNC_SEND_RECEIVE interaction is executed by a client application, the application server returns an existing connection with the user-specified clientID, or creates a new connection with the user-specified clientID. The user-specified clientID identifies the socket connection and the TPIPE and associated OTMA Asynchronous Hold Queue.

The IMS resource adapter ensures that a socket is associated with the connection and sends the request with input data to IMS Connect using that socket. IMS Connect then sends the message to IMS, where IMS runs the transaction and returns the output message. On receiving the output message, the IMS resource adapter internally sends an ACK to IMS which signals to discard the output from the IMS queue. When the connection is closed or the application terminated, the connection is returned to the connection pool for reuse by another application that is running a commit mode 0 interaction with the same user-specified clientID.

- Error processing scenario

All errors result in a resource exception being thrown to the client application. In addition, some errors result in the socket being disconnected by IMS Connect. In the case of commit mode 0 interactions, this means the output message cannot be delivered to the client application. The undelivered output is queued to the TPIPE associated with the user-specified clientID.

The properties, purgeAsyncOutput and reRoute are not applicable to dedicated persistent sockets. You can not purge or reroute undelivered output messages on a dedicated persistent socket.

- ExecutionTimeout scenario

If an execution timeout occurs, the socket remains open and the output of the commit mode 0 interaction is queued to the TPIPE associated with the user-specified clientID for later retrieval. When the connection is closed or the application terminated, the IMSManagedConnection object is returned to the connection pool for reuse by another application that is running a commit mode 0 interaction with the same user-specified clientID.

Retrieving asynchronous output

There are two types of socket connections, shareable persistent socket and dedicated persistent socket, that can be used to retrieve asynchronous output. The way to retrieve asynchronous output messages is different depending on the type of socket connection used. The interactionVerb property values that can be used to retrieve asynchronous output are:

SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT, and
SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT (along with the older
SYNC_RECEIVE_ASYNCOUTPUT).

Note: There is no difference in function between
SYNC_RECEIVE_ASYNCOUTPUT and
SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT. However, it is
recommended that you use the new name
SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT with V9.1.0.1 and
later deliverables of the IMS resource adapter. Only the new name,
SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT, will be used in the
rest of this document.

The difference between SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT and SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT determines how IMS Connect checks for output on the IMS OTMA Asynchronous Hold Queue. For SYNC_RECEIVE_ASYNCOUTPUT or SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT interactions, if there is no asynchronous output in the IMS OTMA Asynchronous Hold Queue when the retrieve request is made, IMS Connect will return an execution timeout notification as soon as the execution timeout value specified by the client application has passed. For this reason, the shortest possible execution timeout value, 10, is recommended for SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT interactions.

For a SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT interaction, if there is no asynchronous output in the IMS OTMA Asynchronous Hold Queue when the retrieve request is made, IMS Connect will wait up to the length of time specified in the executionTimeout property of the interaction for OTMA to return a message. If there is still no asynchronous output in the hold queue when the execution timeout has passed, IMS Connect will return an execution timeout error. For a SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT interaction, you should select an appropriate execution timeout value, rather than the shortest possible value.

All three interactionVerb property values require commit mode 0 and can be used on both shareable persistent socket and dedicated persistent socket connections. In addition, IMSInteractionSpec properties purgeAsyncOutput, reRoute and reRouteName do not apply to interactions which use these three interactionVerbs and are ignored by IMS Connector for Java. The way that interactionVerb properties are invoked on dedicated and shareable persistent socket connections is different.

Retrieving asynchronous output on dedicated persistent socket connections

To retrieve the queued output message on a dedicated persistent socket, the client application must execute a commit mode 0 interaction with the `interactionVerb` property of `IMSInteractionSpec` set to `SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT`, or `SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT`.

In addition to executing a commit mode 0 interaction on a dedicated persistent socket connection with the appropriate `interactionVerb` property of `IMSInteractionSpec`, the client application must also provide a value for the `clientId` property of `IMSConnectionSpec`. The `clientId` is required because it determines the TPIPE from which the asynchronous output will be retrieved. To retrieve output messages from a commit mode 0 interaction on a dedicated persistent socket, the `clientId` specified on the `SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT/NOWAIT` interaction must match the value specified for the original commit mode 0 interaction. To retrieve output messages sent to an alternate PCB, the `clientId` specified on the `SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT/NOWAIT` interaction must match the name of the alternate PCB. To retrieve output messages which were rerouted to a `reRouteName` destination, the `clientId` on the `SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT/NOWAIT` interaction must be set to that `reRouteName` property destination.

Retrieving asynchronous output on shareable persistent socket connection

To retrieve an undelivered output message resulting from an interaction on a shareable persistent for which the `reRoute` flag has not been set to `TRUE`, the client application must execute a `SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT` or `SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT` interaction on the same shareable persistent socket connection in the same application that invoked the interaction that led to the asynchronous output being queued. The reason that the two interactions must be invoked within the same client application is that the IMS resource adapter automatically generates a client-ID for shareable persistent socket connections. This generated `clientId` identifies the socket connection as well as the associated OTMA TPIPE to which the asynchronous output is queued. A new client-ID is generated when a new connection is established. On shareable persistent socket connections, the `clientId` is generated by IMS Connector for Java and is unique for each connection. Therefore, to retrieve asynchronous output for a specific generated `clientId`, a connection with the same `clientId` must be used. This means that, for shareable persistent socket connections (which always have unique generated `clientId`s) the same connection must be used. The only way to guarantee that the same connection will be used is to execute both interactions (the original interaction as well as the `SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT` or `SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT` interactions) within the same client application.

The following situations can result in an output message being queued on an IMS OTMA Asynchronous Hold Queue:

1. An IMS application program inserts an output message to an alternate PCB.
2. The output from a commit mode 0 interaction on a shareable or dedicated persistent socket cannot be delivered to the client application.
3. An interaction spawns a program-to-program switch for which the secondary output is not delivered to the client application. Secondary output is always commit mode 0 output.

Do not specify a value for the `clientID` property of `IMSConnectionSpec` for interactions on shareable persistent socket connections. On shareable persistent socket connections a user specified `clientID` is not allowed since IMS Connector for Java automatically generates the `clientID`.

Displaying output message counts

Using IMS Connect commands, you can choose to display output message counts. This topic describes how to display those message counts.

In IMS and OTMA terminology, a transaction pipe (TPIPE) is a logical connection between a client, such as IMS Connect, and the server, such as IMS OTMA. For commit mode 0 interactions, the TPIPE name is the `clientID` used for the interaction. For commit mode 0 interactions the IMS OTMA Asynchronous Hold Queue associated with the TPIPE has the same name as the `clientID`.

For commit mode 1 interactions, the TPIPE name is the IMS Connect port number used for the interaction, or in the case of Local Option the TPIPE name is the word, `LOCAL`. Therefore, each port will have a TPIPE which will be used for all clients running commit mode 1 interactions on that port.

You can use the IMS Connect command `/DISPLAY T MEMBER IMSConnect_Name TPIPE ALL` to view counts of the output messages sent to IMS Connector for Java, as well as messages inserted to ALTPCBS (Alternate Program Communication Blocks). The following sample output is from a `/DISPLAY T MEMBER HWS1 TPIPE ALL` command. A brief description of the types of TPIPEs and counts for the command output is also provided.

DFS000I	MEMBER/TPIPE	ENQCT	DEQCT	QCT	STATUS	IMS1
DFS000I	HWS1	IMS1				
DFS000I	-9999	0 0	IMS1			
DFS000I	-HWSMIJRC	2 2 0	IMS1			
DFS000I	-CLIENT01	3 2 1	IMS1			
DFS000I	-ALTPCB1	2 1 1	IMS1			
DFS000I	-HWS\$DEF	1	0	1	IMS1	
DFS000I	-RRNAME	1 0	1	IMS1		

Commit Mode 1 interactions on a shareable persistent socket

- The TPIPE name is the port number used for the interaction. For example, 9999.
- The enqueue count (ENQCT) and dequeue count (DEQCT) will be equal and the queue count (QCT) will be 0, because undelivered output messages are not recoverable for commit mode 1 transactions.

Commit Mode 0 interactions on a shareable persistent socket

- The TPIPE name is generated by IMS Connector for Java and will have a prefix of "HWS". For example, HWSMIJRC.
- The enqueue count (ENQCT) and dequeue count (DEQCT) will be equal and the queue count (QCT) will be 0 if all messages are delivered to IMS Connector for Java.
- If output messages are not delivered to IMS Connector for Java on `SYNC_SEND_RECEIVE` interactions and the default values of `reRoute` `FALSE` and `purgeAsyncOutput` `TRUE` are used, the enqueue count (ENQCT) and dequeue count (DEQCT) will be equal and the queue count (QCT) will be 0. All undelivered output messages are discarded.
- If output messages are not delivered to IMS Connector for Java on `SYNC_SEND_RECEIVE` interactions and `reRoute` is set to `TRUE` and

purgeAsyncOutput is set to FALSE, then the enqueue count (ENQCT) will be greater than the dequeue count (DEQCT) and the queue count (QCT) will be the number of messages that were not delivered to IMS Connector for Java. The TPIPE name is the value specified for the reRouteName property; for example, RRNAME, or a default value; for example, HWS\$DEF.

- For SYNC_SEND interactions, output is not expected, so undelivered output does not apply. If SYNC_RECEIVE_ASYNCOUTPUT, SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT and SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT interactions are unsuccessful, the queue count does not change.

Commit Mode 0 interactions on a dedicated persistent socket

- Typically, the TPIPE name is provided by the Java application and will not have a prefix of "HWS". For example, CLIENT01. However, you may occasionally see a TPIPE name of "HWS\$DEF". This is the default value for the reRouteName property.
- The enqueue count (ENQCT) and dequeue count (DEQCT) will be equal and the queue count (QCT) will be 0 if all messages are delivered to IMS Connector for Java, and no undelivered messages were rerouted from interactions on shareable persistent socket connections.
- If output messages are not delivered to IMS Connector for Java or rerouted from interactions on shareable persistent socket connections, the enqueue count (ENQCT) will be greater than the dequeue count (DEQCT) and the queue count (QCT) will be the number of messages that were not delivered. The TPIPE name is the user specified clientID name, for example, CLIENT01.

Output messages inserted to ALTPCBs (Alternate Program Communication Blocks)

- The TPIPE name is the name of the Alternate PCB. For example, ALTPCB1.

SYNC_SEND programming model

If your Java client application issues a SYNC_SEND interaction, the IMS resource adapter sends the request to IMS through IMS Connect and does not expect a response from IMS. Because the IMS resource adapter performs a "send only" interaction with IMS, a SYNC_SEND interaction is typically used with a non-response mode transaction.

To use a SYNC_SEND interaction to run a transaction, your application must provide a value of SYNC_SEND for the interactionVerb property and a value of 0 for the commitMode property of the IMSInteractionSpec object used by the execute method. SYNC_SEND interaction processing varies depending on the type of persistent socket used (shareable or dedicated) and the type of IMS transaction that is run.

Note: IMSInteractionSpec properties purgeAsycOutput, reRoute and reRouteName do not apply to SYNC_SEND interactions and are ignored by IMS Connector for Java.

Shareable persistent socket processing model

The following scenarios describe a SYNC_SEND interaction on a shareable persistent socket connection for different type of transactions.

- Non-response mode transaction

An IMS application program associated with a transaction defined to IMS as non-response mode typically does not require an output message to the I/O PCB, therefore no output message is created and nothing is queued on the TPIPE.

- Response mode transaction

The IMS application program associated with a transaction defined to IMS as non-response mode typically will insert an output message to the I/O PCB. Because the IMS resource adapter does not expect a response from a SYNC_SEND interaction, the output message, if inserted, is queued on the TPIPE with the name of the generated clientID. However, interactions SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT or SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT can be used to retrieve the response, if performed following the SYNC_SEND interaction and in the same application and on the same connection.

- Non-response mode or response mode transactions that invoke an IMS application program that inserts to an alternate PCB

A message inserted to an alternate PCB can be retrieved by executing an interaction on a dedicated persistent socket connection. This can be done by the following ways:

1. Ensuring that the connectionFactory used by the interaction is configured with a value of TRUE for the CM0Dedicated property.
2. Providing the following values for the interaction:
 - IMSInteractionSpec property
interactionVerb=SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT or
SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT
 - IMSInteractionSpec property commitMode=0
 - IMSConnectionSpec property clientID= the name of the alternate PCB

Dedicated persistent socket processing model

The following scenarios describe a SYNC_SEND interaction on a dedicated persistent socket connection for different types of transactions. SYNC_SEND interactions use commitMode0 and dedicated persistent socket connections can only be used for commitMode 0 interactions.

- Non-response mode transaction

The IMS application program associated with a transaction defined to IMS as non-response mode typically does not insert an output message to the I/O PCB, therefore no output message is created and nothing is queued on a TPIPE.

- Response mode transaction

The IMS application program associated with a transaction defined to IMS as non-response mode typically will insert an output message to the I/O PCB. Because the IMS resource adapter does not expect a response from a SYNC_SEND interaction, the output message, if inserted, is queued on the TPIPE with the name provided for the clientID of the interaction. Messages queued to this type of TPIPE can be retrieved by issuing a SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT or SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT interactions. The TPIPE name is the clientID specified for the SYNC_SEND interaction. clientID is required for interactions that use a dedicated persistent socket connection.

- Non-response mode or response mode transactions that invoke an IMS application that inserts to an alternate PCB

A message inserted to an alternate PCB can be retrieved by executing an interaction on a dedicated persistent socket connection. This can be done by the following ways:

1. Ensuring that the connectionFactory used by the interaction is configured with a value of TRUE for the CM0Dedicated property.
2. Providing the following values for the interaction:
 - IMSInteractionSpec property
interactionVerb=SYNC_RECEIVE_ASYNCOUPTUT_SINGLE_NOWAIT or
SYNC_RECEIVE_ASYNCOUPTUT_SINGLE_WAIT
 - IMSInteractionSpec property commitMode=0
 - IMSConnectionSpec property clientID= the name of the alternate PCB

Creating an application to run a Commit mode 0 transaction

To run a commit mode 0 transaction, in addition to providing the transaction input values for the transaction input message, you need to provide a value for the clientID property of IMSConnectionSpec, as well as values for properties of IMSInteractionSpec, such as commitMode. To provide a value for the clientID property, you must expose the clientID property of IMSConnectionSpec to your application so your application can set the value. clientID is the name of the IMS OTMA Asynchronous Hold Queue that will be used to hold the transaction output message, if it cannot be delivered to the client application. By exposing the properties of both InteractionSpec and ConnectionSpec as data, your application can dynamically provide values for all of the properties rather than relying on the WSDL for some of them. This allows for a more flexible application that uses a single enterprise service, rather than creating multiple enterprise services with different values for the properties that are not exposed.

Note: If you wish to use component-managed EIS sign-on to IMS, you must expose the userName and password properties of IMSConnectionSpec so your application can provide the values to those properties. If you wish to use container-managed EIS sign-on to IMS, you do not expose the userName and password properties of IMSConnectionSpec. However, if you use container-managed EIS sign-on for a commit mode 0 transaction, you must expose the clientID property of IMSConnectionSpec.

To expose the properties, you must modify the interface and binding WSDL files of the IMS service. As a result, the application can provide values for those properties as input to the IMS service. IMSInteractionSpec input properties can include commitMode, executionTimeout, imsRequestType, interactionVerb, ltermName, and mapName. IMSConnectionSpec input properties include userName, password, and clientID.

The following steps outline how to create a simple Java application to run a commit mode 0 transaction, as well as how to expose the input properties of IMSInteractionSpec and IMSConnectionSpec. Exposing properties other than clientID are for illustration purposes only. These steps are for a non-managed application that is, an application that is not deployed to WebSphere Application Server. However, the structure of a deployed application is very similar. (See Sample: Creating an enterprise service for an IMS transaction, for instructions to complete the following steps, as necessary.)

1. If necessary, import the IMS resource adapter into your workspace.
2. Create a service project for your IMS service.

3. Import the C, COBOL, or MFS file representing the input and output messages of your IMS transaction.
4. Generate the enterprise service for the IMS transaction. WebSphere Studio generates three WSDL files to describe a service:
 - The abstract service interface definition or interface WSDL file which contains the port types and messages elements.
 - The binding WSDL file, which contains the binding elements that describe how the service interface is implemented.
 - The service WSDL file, which contains the service and port elements that provide the service location as described by a service provider-specific port binding.
5. Expose the input properties of IMSInteractionSpec and IMSConnectionSpec. To do this, you need to modify the interface and binding WSDL files of the IMS service. To expose the properties, complete the following steps:
 - a. **Adding a part in the input message**
 - 1) Double-click on the interface WSDL file for your IMS service. The WSDL editor will open.
 - 2) Click on the **Graph** tab and locate the Messages section.
 - 3) Under the heading **Messages**, ensure that the Request message is selected. You will be adding parts, in addition to the transaction input message, to the Request message.
 - 4) Expand the Request message and click on the part you just added and select **Add Child -> Part**. A **New Part** window opens. Type in the name of the new part. The new part will be used to expose a property of IMSInteractionSpec or IMSConnectionSpec. The name of the new part can be anything you choose. For example, if you are exposing the input property clientID of IMSConnectionSpec, you may want to name the part, myInClientID.

Note: The name of the part you add to the input message in the interface WSDL file and the name you use in the input message in the binding WSDL file must be the same. See InteractionSpec and ConnectionSpec properties as data for the list of IMSInteractionSpec and IMSConnectionSpec properties that can be exposed as data.
 - 5) After you add the new part, a **Part** panel opens. To select the type for the new part, click on the button next to the **Type** field. The **Specify Type** window opens. Select **Select an existing type** and then locate the type of the property from the drop down list. For example, if you are exposing commitMode of IMSInteractionSpec, choose **xsd:int**. Click Finish.

Note: The type you select for the part must match the type of the IMSInteractionSpec or IMSConnectionSpec property that you want to expose. See InteractionSpec and ConnectionSpec properties as data for the list of IMSInteractionSpec and IMSConnectionSpec properties and their type information.
 - 6) Repeat steps d and e for each property you will expose. Below is a section of an interface WSDL file before and after modification to expose the input properties of IMSInteractionSpec and IMSConnectionSpec:

Before:

```
<message name='runShareablePBRequest'>
  <part name="INPUTMSG" type="tns:INPUTMSG"/>
</message>
```

```
<<message name='runShareablePBResponse">
<part name="OUTPUTMSGPart" type="tns:OUTPUTMSG"/>
</message>
<message name="runPB2Request">
```

After:

```
<message name="runShareablePBRequest">
<part name="INPUTMSG" type="tns:INPUTMSG"/>
<part name="myInCommitMode" type="xsd:int"></part>
<part name="myInExecutionTimeout" type="xsd:int"></part>
<part name="myInImRequestType" type="xsd:int"></part>
<part name="myInInteractionVerb" type="xsd:int"></part>
<part name="myInLtermName" type="xsd:string"></part>
<part name="myInMapName" type="xsd:string"></part>
<part name="myInUserName" type="xsd:string"></part>
<part name="myInSocketTimeout" type="xsd:int"></part>
<part name="myInClientID" type="xsd:string"></part>
<part name="myInUserName" type="xsd:string"></part>
<part name="myInPassword" type="xsd:string"></part>
<part name="myInGroupName" type="xsd:string"></part>
</message>

<message name="runShareablePBResponse">
<part name="OUTPUTMSGPart" type="tns:OUTPUTMSG"/>
<part name="myOutAsyncOutputAvailable" type="xsd:boolean"></part>
<part name="myOutConvEnded" type="xsd:boolean"></part>
</message>
```

7) Press **Ctrl-S** to save the file.

b. Binding the new part(s) for an input message

- 1) Double-click on the binding WSDL file for your IMS service. The WSDL editor will open.
- 2) Click on the **Graph** tab.
- 3) Locate the Bindings section and expand the binding operation.
- 4) Right-click on input under `ims:operation`, select Add Extensibility Element, and choose either `ims:interactionSpecProperty` or `ims:connectionSpecProperty` and the panel for either one will open.
- 5) In the `ims:interactionSpecProperty` or `ims:connectionSpecProperty` panel, expand the input binding message and highlight the property.
- 6) Click the value field of the part property and select from the drop down menu, the part you added in step d.
- 7) Click the value field of the `propertyName` property and select the name from the drop down menu. Ensure the property name matches the name of the `IMSInteractionSpec` or `IMSConnectionSpec` property.

For example, below is a section of a binding WSDL file before and after binding the message parts.

Before

```
<operation name="runShareablePB">
<ims:operation />
<input name="runShareablePBRequest"/>
<output name="runShareablePBResponse"/>
</operation>
```

After:

```
<operation name="runShareablePB">
</ims:operation>
<input name="runShareablePBRequest">
<ims:interactionSpecProperty part="myInCommitMode"
propertyName="commitMode"/>
<ims:interactionSpecProperty part="myInExecutionTimeout"
propertyName="executionTimeout" />
```

```

<ims:interactionSpecProperty part="myInImsRequestType"
propertyName="imsRequestType" />
<ims:interactionSpecProperty part="myInInteractionVerb"
propertyName="interactionVerb" />
<ims:interactionSpecProperty part="myInLtermName"
propertyName="ltermName"/>
<ims:interactionSpecProperty part="myInMapName"
propertyName="mapName"/>

<ims:interactionSpecProperty part="myInSocketTimeout"
propertyName="socketTimeout"/>
<ims:interactionSpecProperty part="myInCommitMode"
propertyName="commitMode" />
<ims:connectionSpecProperty part="myInClientID"
propertyName="clientID" />
<ims:connectionSpecProperty part="myInUserName"
propertyName="userName" />
<ims:connectionSpecProperty part="myInPassword"
propertyName="password" />
<ims:connectionSpecProperty part="myInGroupName"
propertyName="groupName" />
</input>
<output name="runShareablePBResponse">
<ims:interactionSpecProperty part="myOutAsyncOutputAvailable"
propertyName="asyncOutputAvailable" />
<ims:interactionSpecProperty part="myOutConvEnded"
propertyName="convEnded"/>
</output>
</operation>

```

8) Press **Ctrl-S** to save the file.

Note: If you do not expose the IMSInteractionSpec properties, the values in the WSDL element **<ims:operation>** will be used.

- c. Generate the Java proxy. You generate a Java proxy for the IMS service from the modified WSDL files.
- d. Test the IMS service in a non-managed environment by creating a Java class to invoke the proxy.

The following steps include code fragments from a Java class that invokes a Java proxy named ShareablePBProxy. The IMS service that ShareablePBProxy invokes was defined with a single REQUEST_RESPONSE operation named runShareablePB. Because the properties of IMSInteractionSpec and IMSConnectionSpec have been exposed, the signature of the runShareablePB method contains parameters, in addition to the transaction input message, that let you provide values for the exposed properties. At a high level, the logic of the Java class is:

1) Populate the transaction input message with data:

```

INPUTMSG input = new INPUTMSG();

input.setIn__ll((short)59);
input.setIn__zz((short) 0);
input.setIn__trcd("IVTNO");
input.setIn__cmd("DISPLAY");
input.setIn__name1("LAST1");
input.setIn__name2("");
input.setIn__extn("");
input.setIn__zip("");

```

2) Invoke the method of the proxy that invokes the IMS service, passing the populated input message and values for the exposed properties:

```

ShareablePBProxy proxy = new ShareablePBProxy();

RunShareablePBResponseMessage outM = proxy.runShareablePB(input,
0, //commitMode

```

```

10000, //executionTimeout
1, //imsRequestType
1, //interactionVerb
"LLLLNAME", //ltermName
"MMMMNAME", //mapName
0, //socketTimeout
"", //clientID
"", //userName
"", //password
"" //groupName
);

```

e. Run the Java application to test the service definition.

Programming model for Commit mode 0 applications

The logic for a commit mode 0 application, whether it is a simple Java class running in a non-managed environment or a deployed EJB is similar.

```

try
{
// Populate the IMS transaction input message.
//
// Invoke the IMS service to run IMS transaction, passing
// the input message, a value for clientID, and
// interactionVerb
// set to SYNC_SEND_RECEIVE.
//
// Process (e.g., display) the IMS transaction output
// message.
}
catch (Exception e)
{
// The IMS transaction output message may or may not
// be available from the OTMA Asynchronous Hold Queue,
// depending on the cause of the exception.
}

```

If the transaction output message cannot be delivered to IMS Connector for Java, it is queued to the OTMA Asynchronous Hold Queue. To retrieve a message from the OTMA Asynchronous Hold Queue, you invoke the IMS service again, set the interactionVerb to SYNC_RECEIVE_ASYNCOUTPUT, and pass a null input message. Because the messages are retrieved from the IMS OTMA Asynchronous Hold Queue according to their clientID, you must also specify the correct value for clientID.

Displaying output message counts

Using IMS Connect commands, you can choose to display output message counts. This topic describes how to display those message counts.

In IMS and OTMA terminology, a transaction pipe (TPIPE) is a logical connection between a client, such as IMS Connect, and the server, such as IMS OTMA. For commit mode 0 interactions, the TPIPE name is the clientID used for the interaction. For commit mode 0 interactions the IMS OTMA Asynchronous Hold Queue associated with the TPIPE has the same name as the clientID.

For commit mode 1 interactions, the TPIPE name is the IMS Connect port number used for the interaction, or in the case of Local Option the TPIPE name is the word, LOCAL. Therefore, each port will have a TPIPE which will be used for all clients running commit mode 1 interactions on that port.

You can use the IMS Connect command `/DISPLAY TMEBER IMSConnect_Name TPIPE ALL` to view counts of the output messages sent to IMS Connector for Java, as well as messages inserted to ALTPCBS (Alternate Program Communication Blocks). The following sample output is from a `/DISPLAY TMEBER HWS1 TPIPE ALL` command. A brief description of the types of TPIPEs and counts for the command output is also provided.

DFS000I	MEMBER/TPIPE		ENQCT	DEQCT	QCT	STATUS	IMS1
DFS000I	HWS1		IMS1				
DFS000I	-9999	0 0 0	IMS1				
DFS000I	-HWSMIJRC	2 2 0	IMS1				
DFS000I	-CLIENT01	3 2 1	IMS1				
DFS000I	-ALTPCB1	2 1 1	IMS1				
DFS000I	-HWS\$DEF	1	0		1	IMS1	
DFS000I	-RRNAME	1 0	1	IMS1			

Commit Mode 1 interactions on a shareable persistent socket

- The TPIPE name is the port number used for the interaction. For example, 9999.
- The enqueue count (ENQCT) and dequeue count (DEQCT) will be equal and the queue count (QCT) will be 0, because undelivered output messages are not recoverable for commit mode 1 transactions.

Commit Mode 0 interactions on a shareable persistent socket

- The TPIPE name is generated by IMS Connector for Java and will have a prefix of "HWS". For example, HWSMIJRC.
- The enqueue count (ENQCT) and dequeue count (DEQCT) will be equal and the queue count (QCT) will be 0 if all messages are delivered to IMS Connector for Java.
- If output messages are not delivered to IMS Connector for Java on SYNC_SEND_RECEIVE interactions and the default values of reRoute FALSE and purgeAsyncOutput TRUE are used, the enqueue count (ENQCT) and dequeue count (DEQCT) will be equal and the queue count (QCT) will be 0. All undelivered output messages are discarded.
- If output messages are not delivered to IMS Connector for Java on SYNC_SEND_RECEIVE interactions and reRoute is set to TRUE and purgeAsyncOutput is set to FALSE, then the enqueue count (ENQCT) will be greater than the dequeue count (DEQCT) and the queue count (QCT) will be the number of messages that were not delivered to IMS Connector for Java. The TPIPE name is the value specified for the reRouteName property; for example, RRNAME, or a default value; for example, HWS\$DEF.
- For SYNC_SEND interactions, output is not expected, so undelivered output does not apply. If SYNC_RECEIVE_ASYNCOUPTUT, SYNC_RECEIVE_ASYNCOUPTUT_SINGLE_NOWAIT and SYNC_RECEIVE_ASYNCOUPTUT_SINGLE_WAIT interactions are unsuccessful, the queue count does not change.

Commit Mode 0 interactions on a dedicated persistent socket

- Typically, the TPIPE name is provided by the Java application and will not have a prefix of "HWS". For example, CLIENT01. However, you may occasionally see a TPIPE name of "HWS\$DEF". This is the default value for the reRouteName property.
- The enqueue count (ENQCT) and dequeue count (DEQCT) will be equal and the queue count (QCT) will be 0 if all messages are delivered to IMS Connector for Java, and no undelivered messages were rerouted from interactions on shareable persistent socket connections.

- If output messages are not delivered to IMS Connector for Java or rerouted from interactions on shareable persistent socket connections, the enqueue count (ENQCT) will be greater than the dequeue count (DEQCT) and the queue count (QCT) will be the number of messages that were not delivered. The TPIPE name is the user specified clientID name, for example, CLIENT01.

Output messages inserted to ALTPCBs (Alternate Program Communication Blocks)

- The TPIPE name is the name of the Alternate PCB. For example, ALTPCB1.

Chapter 6. Transaction processing

The topics in this section describe how the IMS resource adapter supports global transaction management and two-phase commit processing so that your application can run in a J2EE-compliant application server to access IMS transactions. The topics included are:

Global transaction support with two-phase commit

To protect and maintain the integrity of your critical business resources, IMS Connector for Java, as a J2EE Connector Architecture resource adapter, supports global transaction management and two-phase-commit processing. Using this support, you can build a J2EE application to group a set of changes into one transaction, or a single unit of work, so that all changes within a transaction are either fully completed or fully rolled back. This enables your application to run in a J2EE-compliant application server (for example, WebSphere Application Server) to access IMS transactions and data in a coordinated manner. Global transaction management ensures the integrity of the data in IMS.

Example of global transaction support

When you make changes to your protected resources, you want to guarantee that the changes are made correctly. For example, as a bank customer you want to transfer money from your savings account to your checking account. You want to be sure that when the money is deducted from your savings account it is added to your checking account simultaneously. You would not want this transaction to be completed only partially with the money deducted from your savings account but not added to your checking account.

In another example, you need to buy a ticket from San Francisco to Paris but a direct flight is not available. Unless you can successfully reserve a ticket from San Francisco to Chicago and another ticket from Chicago to Paris, you will not commit to your trip to Paris. That is, you will "roll back" your decision to go to Paris because having a confirmed seat for only one part of your trip is not useful to you.

In both of these examples, several smaller transactions are required in order to complete one overall transaction. If there is a problem with one of these smaller transactions, you would not want to commit the overall transaction (such as transferring money or going to Paris). Instead, you would want to roll back every step of the transaction so that none of the smaller transactions are committed. To transfer your money or to go on your trip to Paris successfully, you want the smaller transactions to be managed and coordinated together to complete the overall transaction.

To ensure a coordinated transaction process, the J2EE platform (which consists of a J2EE application server, J2EE application components, and a J2EE connector architecture resource adapter) provides a distributed transaction processing environment where transactions are managed transparently and resources are updated and recovered across multiple platforms in a coordinated manner.

Global transaction and two-phase commit support process

A J2EE-compliant application server (such as WebSphere Application Server) uses a Java transaction manager, also known as an *external coordinator*, to communicate with the application components (for example, Java servlets or Enterprise Java Beans) and the resource managers (for example, IMS or DB2®) through the resource adapters (for example, IMS Connector for Java) to coordinate a transaction.

If a transaction manager coordinates a transaction, that transaction is considered a global transaction. If a transaction manager coordinates a transaction with more than one resource manager, the external coordinator uses two-phase commit protocol.

In the previous bank example, you want to transfer money from your savings account to your checking account. If your savings account information resides on a separate resource manager from your checking account information (for example, your saving account resides on IMS and your checking account resides on DB2), the transaction manager in the application server (WebSphere Application Server) helps the application to coordinate the changes between IMS and DB2 transparently using two-phase commit processing. Specifically, the transaction manager works with the IMS resource adapter to coordinate the changes in IMS.

IMS Connector for Java is designed to work together with the Java transaction manager in the J2EE platform, the Resource Recovery Services (RRS) of z/OS, and IMS Connect to make consistent changes to IMS and other protected resources.

To participate in two-phase commit processing with IMS, IMS Connector for Java uses the IMS OTMA Synchronization level sync-point protocol. To participate in global transaction and two-phase commit processing when the changes are requested from a remote application, IMS uses RRS on z/OS.

RRS, from the point of view of IMS, acts as the "external coordinator" or sync-point manager to coordinate the update and recovery of resources. IMS Connector for Java and IMS Connect, interact with the Java transaction manager running in the application server and RRS on z/OS to allow a global transaction running on a J2EE platform to participate in a coordinated update with IMS running on the host.

When setting up a J2EE application to participate in a global transaction, you must select one of the two available communication protocols to be used between IMS Connector for Java and IMS Connect. The two communication protocols supported by IMS Connector for Java and IMS Connect are TCP/IP and Local Option.

Global transaction with TCP/IP

In a global transaction scope, your J2EE application component can access an IMS transaction by establishing a TCP/IP connection with IMS Connect. Underlying, IMS Connector for Java interacts with the Java transaction manager using the X/Open (XA) protocol to manage the global transaction and two phase commit processing. The XA protocol defines a set of interfaces and interactions describing how the Java transaction manager and the resource managers interact in a distributed transaction processing environment. IMS Connector for Java, together with IMS Connect, uses the XA protocol and works with IMS and Resource Recovery Services (RRS) on z/OS to make consistent changes.

Restrictions: You are required to have RRS running on the same MVS system with IMS Connect.

To set up RRS on IMS Connect, refer to *IMS Connect Guide and Reference* (SC27-0946). For more information about TCP/IP communication protocol for global transaction and two-phase commit processing, see Platform considerations and communication protocol considerations and Two-phase commit environment considerations.

Global transaction with Local Option

If your J2EE application component is running on WebSphere Application Server for z/OS, you can submit IMS transaction messages using Local Option and participate in global transaction processing. This transaction processing is coordinated by Resource Recovery Services (RRS) on z/OS and WebSphere Application Server for z/OS. IMS Connector for Java is RRS-compliant and is designed specifically to work with RRS so that the Java transaction manager in WebSphere and IMS, as the resource manager, can work together to make consistent changes to multiple protected resources. The XA protocol is not used by IMS Connector for Java when running global transaction with Local Option.

Restriction:

- To run a global transaction with Local Option, WebSphere Application Server for z/OS, IMS Connect, and IMS must run in the same MVS system.

Recommendation:

- Use Local Option for optimal performance.
- If WebSphere Application Server for z/OS is running on a different MVS than IMS and IMS Connect, you must use TCP/IP for global transaction.

For information about Local Option communication protocol for global transaction and two-phase commit processing, see Platform considerations and communication protocol considerations, Two-phase commit prerequisites, and Two-phase commit environment considerations.

Additional information on transaction support

Local Transaction

The J2EE Connection Architecture defines the `javax.resource.cci.LocalTransaction` interface to allow a resource manager, rather than a transaction manager, to coordinate a transaction locally. However, IMS Connector for Java only supports transaction coordination with a transaction manager. Thus, IMS Connector for Java does not support the `javax.resource.cci.LocalTransaction` interface. If you call the `IMSCConnection.getLocalTransaction()` method you will get a `NotSupportedException`. To use transaction support with IMS Connector for Java, you need to either use the JTA transaction interface, or set an appropriate transaction attribute in the deployment descriptor in your application. See Using global transaction support in your application for more information.

One-phase commit processing

IMS Connector for Java supports one-phase commit optimization with the transaction manager. As a result, if all changes inside a transaction scope belong to the same IMS resource, the transaction manager might perform one-phase-commit

optimization such that the transaction manager sends the phase two commit request directly to the resource manager for committing the changes without sending the phase one prepare request.

Non-global transaction processing

If no global transaction processing is used in the application (for example, when the transaction attribute is set to TX_NOTSUPPORTED), all non-global transaction processing uses "Sync-On-Return" (OTMA SyncLevel=None). By the time the IMS transaction is committed, the output has been returned to the client.

Conversational transaction processing in global transaction scope

IMS uses a conversational program to divide processing into a connected series of client-to-program-to-client interactions (also called iterations). Each iteration is a type of IMS conversational transaction. Conversational processing is used when one transaction contains several parts. Each part that comprises one large transaction is separately committed or rolled back.

You can run a conversational transaction in the global transaction scope if:

- Each iteration is run under the same transaction level. For example, if the first iteration is processed with a global transaction scope, then all the subsequent iterations in that IMS conversational transaction must be processed at a global transaction level. If you issue the second iteration with no transaction scope, IMS OTMA reports an error.
- Each iteration must be completed with a commit or rollback call before issuing the next iteration in the IMS conversation. You cannot group multiple iterations in a single global transaction scope.

For more information about using global transaction support, see the IMS Connector for Java web page at www.ibm.com/ims and go to Hints and Tips on the Support page.

Two-phase commit prerequisites

The prerequisites for two-phase commit processing with TCP/IP are:

- IMS Connector for Java 2.1.0 or later
- IMS Connect 2.1 or later
- RRS on z/OS 1.2 or later
- IMS Version 8 or later
- WebSphere Application Server Version for z/OS or distributed platforms 5.0 or later

The prerequisites for two-phase commit processing with Local Option are:

- IMS Connector for Java 1.2.2 or later
- IMS Connect 1.2 or later
- The RRS level associated with z/OS Version 2 Release 10 or later
- IMS Version 7 or later
- WebSphere Application Server 4.0.1, PTF 4 or later

Note: RRS must be installed and running for two-phase commit processing to occur. IMS and IMS Connect must also be enabled for RRS processing. (If you are

using two-phase commit processing with Local Option, IMS Connect does not need to be enabled for RRS processing.) You can enable RRS processing on IMS Connect by either issuing the IMS Connect command, SETRRS ON or set RRS=Y in the IMS Connect configuration file. To ensure IMS is enabled with RRS, check that the RRS value in the startup parameter within your IMS environment is set to Y. This will appear in the job logs generated when IMS is brought up.

Additionally, to run two-phase commit IMS, IMS Connect, and RRS must all be in the same MVS image. For more information about two-phase commit, see Two-phase commit environment considerations.

Using global transaction support in your application

The J2EE platform allows you to use either a programmatic or a declarative transaction demarcation approach to manage transactions in your application. The programmatic approach is the component-managed transaction and the declarative transaction demarcation approach is the container-managed transaction.

Component-managed (or Bean-managed) transaction

The J2EE application uses the JTA `javax.transaction.UserTransaction` interface to demarcate a transaction boundary to a set of changes to the protected resource programmatically. Component-managed transactions can be used in both the servlet and the EJB environment. In the case of an EJB, you set the transaction attribute in its deployment descriptor as `TX_BEAN_MANAGED`.

A transaction normally begins with a `UserTransaction.begin()` call. When the application component is ready to commit the changes, it invokes a `UserTransaction.commit()` call to coordinate and commit the changes. If the application component must roll back the transaction, it invokes `UserTransaction.rollback()` and all changes are backed out. For example:

```
// Get User Transaction
javax.transaction.UserTransaction transaction =
ejbcontext.getUserTransaction();

// Start transaction
transaction.begin();

// Make changes to the protected resources.
// For example, use the J2EE/CA's CCI Interaction interface
// to submit changes to an EIS system(s)
interaction.execute(interactionSpec, input, output);

if (/* decide to commit */) {
// commit the transaction
transaction.commit();

} else { /* decide to roll back */
// rollback the transaction
transaction.rollback();
}
```

Container-managed transaction

Container-managed transactions can be used only in the EJB environment. The EJB specifies a container-managed transaction declaratively through the transaction attribute in the deployment descriptor (such as `TX_REQUIRED`). A container-managed transaction is managed by the EJB container. The container calls

the appropriate methods (such as begin, commit, or rollback) on behalf of the EJB component. This declarative approach simplifies the programming calls in the EJB.

Related Reading: For more information about the J2EE architecture and JTA specifications, see <http://java.sun.com/j2ee/docs.html>.

Two-phase commit environment considerations

To run a two-phase commit application, consider the following suggestions:

- It is best to have as many MPP regions as possible running to ensure that two-phase commit applications do not contend for a region; because a transaction that is within a two-phase commit application uses an MPP region for the duration of the entire two-phase commit transaction.
- If a number of IMS transactions are performed within a two-phase commit transaction, at least that many MPP regions must be available to avoid hanging the two-phase commit application.
- To safeguard against a transaction that may be waiting for an extensive amount of time for resources, it is recommended to set an appropriate timeout value for each interaction taking place within the global transaction.
- Avoid having an excessive number of database interactions performed in one two-phase commit transaction. If multiple IMS transactions are used within a two-phase commit transaction, they could possibly contend or lock in an attempt to update or modify the same data. To avoid this, it's best to write an application that will prevent a user from accessing duplicate entries within the same two-phase commit operation.
- Consider configuring your IRLM or PI locking manager to use a block size that is as small as the smallest entry to that database. Larger block sizes might have two transactions contending for entries that may not even be the same and yet reside close to one another on the hard disk.
- If multiple interactions are performed using the same IMS transaction on the same IMS database within a global transaction (unit of work), each interaction with that IMS transaction must run on a separate MPP region. The IMS transaction must have a SCHDTYP=PARALLEL and a PARLIM=0 value, to indicate that the IMS transaction can run on multiple MPP regions and that it will always meet the scheduling requirements (the number of messages will be greater than zero) to process every interaction on a new MPP region.
- If a region is hung, waiting for RRS-OTMA and no execution timeout value has been set, you can end the attempt to run a transaction that is hanging the MPP region. This can be done by issuing a stop region IMS command with the abend transaction parameter. For example, `/STOP REGION reg#ABDUMP tranname`. This will rollback the transaction for that particular interaction and free the MPP region.

For more information about two-phase commit, including sample applications, see the *IMS Connector for Java Guide and Reference*.

Chapter 7. Diagnosing problems

The topics in this section provide information on how to log and trace component information and diagnose problems, as well as list the messages and exceptions of the IMS resource adapter and MFS plugin. The topics included are:

Diagnosing problems when using the IMS resource adapter

If you are unable to access IMS from your Java application, consider performing the following actions to diagnose the problem:

- Verify that you have the correct prerequisites for using the IMS resource adapter. See Prerequisites for using the IMS resource adapter.
- Verify that IMS Connect is active by ensuring that the outstanding IMS Connect reply "HWSC0000I *IMS CONNECT READY* *ims_connect_name*" appears on the system console of the target machine.
- Verify that the PORT and DATASTORE are ACTIVE by entering the IMS Connect command VIEWHWS at the IMS Connect outstanding reply.
- Verify that IMS is active by ensuring that the outstanding IMS reply "DFS996I *IMS READY*" appears on the system console of the target machine.
- Verify that the XCF status of both the IMS and IMS Connect members is ACTIVE by entering the IMS command /DISPLAY OTMA at the outstanding IMS reply. The display output should be similar to the following:

DFS000I SECURITY	GROUP/MEMBER IMS1	XCF-STATUS	USER-STATUS	
DFS000I IMS1	XCFGRPNM			
DFS000I IMS1	-IMSNAME	ACTIVE	SERVER	FULL
DFS000I IMS1	-ICONNAME	ACTIVE	ACCEPT TRAFFIC	
DFS000I	*02033/143629*	IMS1		

- If you're using TCP/IP to communicate between the Java application and IMS Connect, verify that you can successfully "ping" the target host machine. If you cannot ping the host machine and you are using a host name rather than an IP address, ensure that the host name is sufficiently qualified.

If your IMS service is not providing the expected output from the IMS transaction, ensure that the output message returned by the IMS application program matches the output COBOL definition used by the service. For a J2EE application, you can view the IMS OTMA message containing the message returned by the IMS application program by setting the traceLevel property to 3. See Logging and tracing with the IMS resource adapter for instructions on how to turn on the IMS resource adapter trace. For more information on the IMS OTMA message, go to the IMS web site, <http://www.ibm.com/ims> and select **IMS Connect**.

Logging and tracing with the IMS resource adapter

The IMS resource adapter, in addition to other J2EE components, provides controls for logging and tracing component information. When these controls are set for logging and tracing and you run your Java application using the WebSphere Unit Test Environment, a trace file is created.

Note: Ensure that only one client is running when the trace is on.

To set controls for logging and tracing, complete the following steps:

1. In the Server Configuration view, double-click your **server configuration** to open the **WebSphere Server Configuration** editor.
2. Select the **J2C** tab in the editor.
3. On the **J2C Options** page, select an IMS resource adapter in the **J2C Resource Adapters** table.
4. Scroll down to the **J2C Connection Factories** table and select the connection factory for which you want to turn the trace on.
5. Scroll down to the **Resource Properties** table and select the **TraceLevel** resource property. Specify a non-zero value to enable logging and tracing. **TraceLevel** values correspond to constants in the interface `com.ibm.connector2.ims.ico.IMSTraceLevelProperties`.

TraceLevel Value	IMSTraceLevelProperties	Description
0	RAS_TRACE_OFF	No tracing or logging occurs.
1	RAS_TRACE_ERROR_EXCEPTION	Only errors and exceptions are logged.
2	RAS_TRACE_ENTRY_EXIT	Errors and exceptions plus the entry and exit of important methods are logged.
3	RAS_TRACE_INTERNAL	Errors and exceptions, the entry and exit of important methods, and the contents of buffers sent to and received from IMS Connect are logged.

6. After entering the **TraceLevel** value on the page for the **J2C** tab, select the **Trace** tab.
7. Ensure that the **Enable trace** check box is selected. To enable logging and tracing in the IMS Resource adapter, enter the following in the **Trace string** field:

```
com.ibm.connector2.ims.*=all=enabled  
com.ibm.ims.ico.*=all=enabled
```

Other combinations of trace strings will enable tracing in other components. For example, with the following trace string:

```
com.ibm.ejs.j2c.*=all=enabled:com.ibm.connector2.*=all=enabled
```

the string `com.ibm.ejs.j2c.*` provides you with logging and tracing of WebSphere's implementation of the J2EE Connector Architecture and the string `com.ibm.connector2.*` provides you with logging and tracing of all of the resource adapters, including IMS.

8. You can accept the default name and location of the trace output file or you can modify it. For example, depending on how you set your substitution variables, the default name and location might be:

```
your_workspace\.metadata\plugins\com.ibm.etools.server.core  
\tmp0\logs\server1\trace.log
```

To modify the default name and location, enter a different name and location of the file in the **Trace output file** field on the **Trace Options** page of the server configuration.

9. When you are finished making changes, close the editor and select **Yes** to save your changes.
10. Check the **Status** column of the **Servers** view and restart the server instance, if necessary. You will most likely have to restart the server instance if you are using the WebSphere Test Environment.
11. Run your Java application and then examine the trace file.

J2CA0056I, WLTC0017E, HWSP1445E, and HWSSL00E Error Messages

J2CA0056I

When IMS resource adapter throws an exception, it can be caught by a component other than your Java application. For example, when you run a deployed application, IMS Connector for Java exceptions are often caught by the WebSphere Application Server. WebSphere Application Server may then issue its own message, including in it the message from the IMS resource adapter exception. For example, when execution timeout occurs, you see the following on the Console:

- J2CA0056I: The Connection Manager received a fatal connection error from the Resource Adaptor for resource myConnFactory. The exception which was received is
IC00080E:
com.ibm.connector2.ims.ico.IMSTCIPManagedConnection@e59583c.
processOutputOTMAMsg(byte[],IMSInteractionSpec, int) error.
Execution timeout has occurred for this interaction.
The executionTimeout was [0] milliseconds. The IMS Connect TIMEOUT was used.

J2CA0056I is an informational message from WebSphere Application Server. The fatal connection error refers to the fact that IMS Connect closes the socket in the case of an execution timeout, which results in WebSphere Application Server's Connection Manager removing the connection object for the socket from the connection pool.

Another example occurs when a transaction (non-persistent) socket is used for a commit mode 0 interaction. In this case, you see the following on the Console:

- J2CA0056I: The Connection Manager received a fatal connection error from the Resource Adaptor for resource myConnFactory. The exception which was received is
IC00089I:
com.ibm.connector2.ims.ico.IMSTCIPManagedConnection@6db5d83a.call(Connection, InteractionSpec, Record, Record). Non-persistent socket closed for Commit Mode 0 IMS transaction.

J2CA0056I is an informational message from WebSphere Application Server. The fatal connection error refers to the fact that IMS Connect closes the transaction socket and the IMS resource adapter causes WebSphere Application Server's

Connection Manager to remove the connection object for the socket from the connection pool.

WLTC0017E

A local transaction containment (LTC) is used to define the application server behavior in an unspecified transaction context. For example, if a single method within a container managed EJB that has a transaction attribute of NotSupported is called outside of any transaction scope, WebSphere will create a local transaction to handle resources used during the execution of that method. The message above is produced by the WebSphere Transaction Monitor to indicate that the resources enlisted with the LTC were rolled back instead of committed due to `setRollbackOnly()` being called on the LTC. This message does not require any action by the user and is for your information only.

- WLTC0017E: Resources rolled back due to `setRollbackOnly()` being called.

Note: The prefix of a WebSphere Application Server message indicates the component that issued the message. You can find documentation of these messages, by component, in Integration Edition's Help using **WebSphere Application Server Enterprise > Quick reference > Messages**. All messages are documented with user/system action and explanation. These messages are also documented in the WebSphere Application Server Version 5 Information Center.

HWSP1445E

When you provide Connection Properties to the New IMS Service wizard in Integration Edition or when you configure a Connection Factory for use by your Java application, you choose whether or not you are using SSL with the **SSLEnabled** property. If you are using SSL (**SSLEnabled=TRUE**), then the port number you provide must be configured as an SSL port in IMS Connect. If you accidentally provide a non-SSL port for your Java application, unexpected results will occur when you run your application.

- IMS Connector for Java will throw an exception indicating a communication error:

```
javax.resource.spi.CommException:  
IC00003E:  
com.ibm.connector2.ims.ico.IMSTCIPManagedConnection@56503fc6.connect()  
error.  
Failed to connect to host [CSDMEC13], port [9999].  
[java.net.SocketException:  
Connection reset by peer: socket closed]
```

- The following IMS Connect message will be displayed on the MVS console:

```
HWSP1445E UNKNOWN EXIT NAME SPECIFIED IN MESSAGE PREFIX; MSGID=  
/9 * !hR, M=SDRC
```

The first step in establishing an SSL connection involves the SSL handshake protocol, in which the client (IMS Connector for Java) sends the server (IMS Connect) an SSL "Hello" message. In the scenario described above, IMS Connect is waiting for an incoming message on a non-SSL port. When IMS Connect receives the handshake message it interprets it as an OTMA message with a valid Exit name in the prefix and issues message HWSP1445E.

HWSSSL00E

The opposite scenario to the one above occurs when you are **not** using SSL (**SSLEnabled=FALSE**), but the port number you provide for your Java application is configured as an SSL port in IMS Connect. In this case:

- IMS Connector for Java will throw an exception indicating a communication error:

```
javax.resource.spi.CommException: IC00005E:  
com.ibm.connector2.ims.ico.IMSTCPManagedConnection@5bcdcd4.receive()  
error. A communication error occurred while sending or receiving  
the IMS message.  
[java.net.SocketException: Connection reset by peer: socket closed]
```

- The following IMS Connect message will be displayed on the MVS console:

```
HWSSSL00E Unable to initialize the SSL socket:Error while reading  
or writing data
```

IMS Connect's attempt to initialize the SSL socket fail, since it does not receive the initial client "Hello" message that is part of the SSL handshake protocol.

IMS resource adapter messages and exceptions

While you develop Java programs that use IMS Connector for Java, you might encounter situations in which your program throws exceptions. Some of these exceptions are thrown by IMS Connector for Java, while others are thrown by class libraries used by IMS Connector for Java (such as the Java class libraries). This topic provides information on exceptions generated by IMS Connector for Java J2C applications.

The following terms, in *italics* in the message descriptions that follow, are replaced by specific values at runtime.

hostname

The TCP/IP host name of the machine that is running IMS Connect.

innermethodname

The name of the method that originally throws this exception. This exception has been caught by IMS Connector for Java and is being re-thrown to another exception, according to the Common Connector Framework specification.

length The length of the data.

libraryFileName

The Local Option native library file name.

llvalue

The value of LL.

maxlength

The maximum valid length of the data.

methodname

The name of the method that is throwing this exception.

mode The type of interaction between IMS Connector for Java and the IMS Connect component on the host (as defined in the interactionspec).

nativeMethodName

The Local Option native method name.

portnumber

The port number that is assigned to IMS Connect.

propertyname

The name of the property.

propertyvalue

The value of the property.

reasoncode

The reason code that is returned by IMS Connect.

rectype

The type of the record.

returncode

The return code, formatted in decimal, that is returned by IMS Connect.

sensecode

The sense code, formatted in decimal, that is returned from IMS OTMA

socketexception

The socket exception.

source_exception

The exception thrown when the error first occurred in an internal method.

source_methodname

The internal method in which the error first occurred.

state The internal state of IMS Connector for Java.

Related Reading

- For information on exceptions that are thrown from other class libraries, see the Javadoc information for the specific class library.
- For information on exceptions related to Local Option support, see *IBM IMS Connect User's Guide and Reference*. Some exceptions are thrown based on IMS, IMS OTMA or IMS Connect errors returned by IMS Connect. For information on IMS OTMA and IMS Connect errors, see *IBM IMS Open Transaction Manager Access Guide* and *IBM IMS Connect User's Guide and Reference*, respectively. For information on IMS errors, see *IBM IMS Messages and Codes*.

Exceptions generated by IMS Connector for Java J2C applications

The following exception messages are produced by applications built with the Java 2 Platform, Enterprise Edition (J2EE) Connector Architecture (J2C) class libraries when an error condition is detected.

ICO0001E

```
javax.resource.spi.EISSystemException:
ICO0001E: methodname error.
IMS Connect returned error:
RETCODE=[returncode], REASONCODE=[reasoncode].
reasoncode_string.
```

Explanation: IMS Connect returned an error. The connection in error will not be reused. *reasoncode_string* provides a brief description of the *reasoncode* , if available.

User Action: Check the MVS console for associated IMS Connect error messages. IMS Connect error messages begin with the characters " **HWS**". For diagnostic

information on the return code (*returncode*) and reason code (*reasoncode*) values, as well as IMS Connect error messages, see the *IMS Connect Guide and Reference*.

ICO0002E

```
javax.resource.spi.EISSystemException:  
ICO0002E:methodname error.  
IMS OTMA returned error:  
SENSECODE=[sensecode], REASONCODE=[otmareasoncode].  
[source_methodname:source_exception]
```

Explanation: IMS OTMA returned a NAK error.

User Action: For diagnostic information on the sense code (*sensecode*) and OTMA reason code (*otmareasoncode*) values of the NAK error, see the *IMS OTMA Guide and Reference*. Note that IMS Connector for Java displays *sensecode* and *otmareasoncode* in decimal. If the application is running with two-phase commit, you may receive the following sense code values with the NAK error:

- Sense code = 17 (decimal, 23 Hex)
Your IMS is not enabled with RRS processing. Ensure your IMS has Protected Conversation processing with RRS enabled. See Two-phase commit prerequisites for more information.
- Sense code = 46 (decimal, 2E Hex)
RRS and two-phase commit processing is not supported by IMS Connect and IMS Connector for Java. Make sure that both your IMS Connect and IMS Connector for Java is at least version 2.1.0 or above.

ICO0003E

```
javax.resource.spi.CommException:  
ICO0003E:methodname error.  
Failed to connect to host [hostname],  
port [portnumber].  
[java_exception]
```

Explanation: IMS Connector for Java was unable to connect to the host and port combination. *java_exception* indicates the reason for the failure to connect. For additional information see the User Action section below.

User Action: Examine *java_exception* to determine the reason for the failure to connect to the host. Some values for *java_exception* are:

- **java.net.UnknownHostException: hostname**
The host name you specified when configuring the Connection Factory used by your application is invalid or your application specified an invalid host name. Check the spelling of the host name. You may have to use the fully qualified path for host name or the IP address.
- **java.net.ConnectException: Connection refused**
Some possible reasons for the exception are:
 - The port number is invalid. Ensure that you are using a valid port number for the IMS Connect indicated by *hostname*.
 - The specified port is stopped. This can be determined using the IMS Connect command VIEWHWS. If the port is stopped its status will be NOT ACTIVE. Use the IMS Connect command, **OPENPORT dddd**, where *dddd* is the specified port number, to start the port.
 - IMS Connect on the specified host is not running. Start IMS Connect on the host machine.

- TCP/IP was restarted without canceling and restarting IMS Connect or issuing **STOPPORT** followed by **OPENPORT** on the host.
- **java.net.SocketException: connect (code=10051)**
Some possible reasons for the exception are:
 - The machine with the specified host name is unreachable on the TCP/IP network. Ensure that the host machine is accessible from the TCP/IP network. Verify by issuing the ping command to the specified host machine. Enter the ping command on the machine on which IMS Connector for Java is running. Start TCPIP on the host, if it is not started.
 - TCP/IP was restarted but the status of the port used by the application was NOT ACTIVE. To correct this situation you can do one of the following:
 -
 - Use the IMS Connect command **OPENPORT dddd**, where *dddd* is the port number, to activate the port
 - Restart IMS Connect

ICO0005E

javax.resource.spi.CommException:
ICO0005E:methodname error.
A communication error occurred while sending or receiving the IMS message.
[*java_exception*]

Explanation: IMS Connector for Java was unable to successfully complete a send and receive interaction with the target IMS Connect. *java_exception* indicates the reason for the failure to complete the interaction. For additional information see the User Action section below.

User Action: Examine *java_exception* to determine the reason for the failure. Some values for *java_exception* are:

- **java.io.EOFException**
Some possible reasons for the exception are:
 - The timeout value specified in the IMS Connect configuration file is exceeded before IMS Connect receives a response from IMS. Exceeding a timeout value typically occurs when there is no region available in IMS to run the IMS transaction that processes the client's request. If this is the case, ensure that an appropriate region is started and available to process the request. Exceeding a timeout value can also occur if the IMS application program associated with the transaction is stopped. If this is the case, use the IMS command **/START PROGRAM** to start the IMS application program.
 - **Note:** This is the expected behavior for the following configurations:
 - Releases of IMS Connector for Java prior to 1.2.6, running with IMS Connect 1.2
 - IMS Connector for Java 1.2.6 or 2.1.0, running with IMS Connect 1.2 plus APAR PQ71355
 - A Java client tries to use a previously active client (for example, a connection from the pool) for which an IMS Connect **STOPCLNT** command has been issued.
- **java.net.SocketException: Connection reset by peer: socket write error**
Some possible reasons for the exception are:
 - A Java client attempts to use a connection for which the underlying socket is no longer connected to IMS Connect. This can happen if IMS Connect is recycled, but the application server is not. After IMS Connect is restarted, the

connection pool will contain connections that formerly were successfully connected to IMS Connect. As clients attempt to reuse each of these connections, the exception **java.net.SocketException** is thrown and the connection object removed from the connection pool. Eventually all these connections will be removed from the pool and new connections will successfully be created.

- **Note:** This behavior can be changed in WebSphere Application Server by setting the **Purge Policy** of the connection factory used by the Java application to *Entire Pool*.
- TCP/IP on the host is coming down.

ICO0006E

```
javax.resource.ResourceException:  
ICO0006E:methodname error.  
The value provided for DataStoreName is null or an empty string.
```

Explanation: The method indicated in *methodname* was invoked using an empty DatastoreName parameter. This error message will appear in the trace log when a connection factory with an empty DatastoreName parameter is started. This message will be followed by a J2EE Connector warning.

```
J2CA0007W: An exception occurred while invoking method setDataStoreName on  
com.ibm.connector2.ims.ico.IMSManagedConnectionFactory used by resource  
Connection_Factory_JNDI_name.
```

Processing will then continue leading to other error messages after IMS Connect sends a response indicating that a datastore with a null name cannot be found. The underlying message which triggers the other messages is:

```
javax.resource.spi.EISSystemException: ICO0001E:  
com.ibm.connector2.ims.ico.IMSTCPManagedConnection@.processOutputOTMAMsg(byte[],  
InteractionSpec, Record) error. IMS Connect returned error: RETCODE=[4],  
REASONCODE=[NFDDST ]. Datastore not found.
```

When this error occurs, a corresponding HWSS0742W warning message is displayed on the MVS console of the host machine where IMS Connect is running. This HWSS0742W message will include a field showing the datastore name that it attempted to find, in this case all blanks:

```
DESTID=      ;
```

User Action: Provide a valid name for the DatastoreName parameter. In a managed environment, the DatastoreName is specified when you are configuring a Connection Factory to be used by WebSphere Application Server. In a non-managed environment, the DatastoreName is specified in your Java application.

ICO0007E

```
javax.resource.NotSupportedException:  
ICO0007E:methodname error.  
The [propertyName] property value [propertyValue] is not supported.
```

Explanation: The value *propertyValue* specified for the property *propertyName* is not supported.

User Action: Provide a supported value for the named property. For example, certain values of the InteractionVerb property of the InteractionSpec class that are defined in the J2C architecture are not supported by the IMSInteractionSpec class

in this release of IMS Connector for Java. Also the ReRoute value TRUE is not supported on dedicated persistent socket connections.

ICO0008E

javax.resource.ResourceException:
ICO0008E:*methodname* error. The value [*propertyValue*] of the [*propertyName*]
property exceeds the maximum allowable length
of [*maxPropertyLength*].

Explanation: The length of the value *propertyValue* supplied for property *propertyName* exceeds *maxPropertyLength*, the maximum length allowed for values of property *propertyName*.

User Action: Provide a value for the named property which does not exceed *maxPropertyLength*.

ICO0009E

javax.resource.ResourceException:
ICO0009E:*methodname* error.
The [*propertyName*] property value [*propertyValue*] is invalid.

Explanation: The value *propertyValue* specified for the property *propertyName* is not valid.

User Action: Provide a value which is valid for the named property. For example, valid values for the InteractionVerb property of the InteractionSpec class of IMS Connector for Java are listed in the Javadoc for the IMSInteractionSpec class.

ICO0010E

javax.resource.spi.IllegalStateException:
ICO0010E:*methodname* error.
Method invoked on invalid IMSConnection instance.

Explanation: The method indicated in *methodname* was invoked on an invalid IMSConnection instance. If the *methodname* is *lazyEnlist*, an attempt was made to enlist a connection in the current transaction that could not be enlisted.

User Action: The named method was most likely issued on an IMSConnection instance that was already closed. Ensure that the IMSConnection instance is not already closed before you attempt to use it or close it.

ICO0011E

javax.resource.spi.IllegalStateException:
ICO0011E:*methodname* error.
Method invoked on invalid IMSInteraction instance.

Explanation: The method indicated in *methodname* was invoked on an invalid IMSInteraction instance.

User Action: The named method was most likely issued on an IMSInteraction instance that was already closed. Ensure that the IMSInteraction instance is not already closed before you attempt to use it or close it.

ICO0012E

javax.resource.ResourceException:
ICO0012E:*methodname* error.
The value provided for HostName is null or an empty string.

Explanation: The method indicated in *methodname* was invoked using a null or empty *HostName* parameter.

User Action: Provide a valid *HostName* parameter. In a managed environment, the property value is specified when you are configuring a Connection Factory to be used by WebSphere Application Server. In a non-managed environment, the property value is specified in your Java application.

ICO0013E

```
javax.resource.ResourceException:  
ICO0013E:methodname error.  
ConnectionManager is null.
```

Explanation: The method indicated in *methodname* was invoked. The application server invoked the **createConnectionFactory** method of the **IMSManagedConnectionFactory** class with a null *ConnectionManager* object.

User Action: Provide a valid *HostName* parameter. This form of the **createConnectionFactory** method is used in a managed environment. It is not typically invoked by a client program. Contact the service personnel for your application server.

ICO0014E

```
javax.resource.ResourceException:  
ICO0014E:methodname error.  
Input record contains no data.
```

Explanation: The method indicated in *methodname* was invoked with an input record that contained no data.

User Action: Verify that the input record that you provide is not empty.

ICO0015E

```
ResourceAdapterInternalException  
ICO0015E:methodname error.  
Unexpected error encountered while processing the OTMA message.  
[java_exception]
```

Explanation: An unexpected internal error was encountered while processing the OTMA message.

User Action: Contact your IBM service representative.

ICO0016E

```
javax.resource.ResourceException:  
ICO0016E:methodname error.  
The value provided for DataStoreName is null or an empty string.
```

Explanation: The method indicated in *methodname* was invoked using an empty *DataStoreName* parameter. This error message will appear in the trace log when a connection factory with an empty *DataStoreName* parameter is started. This message will be followed by a J2EE Connector warning.

```
J2CA0007W: An exception occurred while invoking method setDataStoreName on  
com.ibm.connector2.ims.ico.IMSManagedConnectionFactory used by resource  
Connection_Factory_JNDI_name.
```

Processing will then continue leading to other error messages after IMS Connect sends a response indicating that a datastore with a null name cannot be found. The underlying message which triggers the other messages is:

```
javax.resource.spi.EISSystemException: ICO0001E:  
com.ibm.connector2.ims.ico.IMSTCIPManagedConnection@.processOutputOTMAMsg(byte [],  
InteractionSpec, Record) error. IMS Connect returned error: RETCODE=[4],  
REASONCODE=[NFNDST ]. Datastore not found.
```

When this error occurs, a corresponding HWSS0742W warning message is displayed on the MVS console of the host machine where IMS Connect is running. This HWSS0742W message will include a field showing the datastore name that it attempted to find, in this case all blanks:

```
DESTID=          ;
```

User Action: Provide a valid name for the DatastoreName parameter. In a managed environment, the DatastoreName is specified when you are configuring a Connection Factory to be used by WebSphere Application Server. In a non-managed environment, the DatastoreName is specified in your Java application.

ICO0017E

```
ResourceAdapterInternalException  
ICO0017E:methodname error.  
Invalid value provided for TraceLevel.
```

Explanation: An invalid trace level was specified.

User Action: Specify a valid trace level. Optionally, this exception can be ignored due to the fact that the default trace level will be used for this connection factory. In this case, the connection factory is still usable but the trace level will be the default trace level.

ICO0018E

```
javax.resource.ResourceException:  
ICO0018E:methodname error.  
The value provided for PortNumber is null.
```

Explanation: The method indicated in *methodname* was invoked using a null PortNumber.

User Action: Provide a valid PortNumber parameter. In a managed environment, the property value is specified when you are configuring a Connection Factory to be used by WebSphere Application Server. In a non-managed environment, the property value is specified in your Java application.

ICO0024E

```
javax.resource.ResourceException:  
ICO0024E:methodname error.  
Invalid segment length (LL) of [llvalue] in input object.[java_exception]
```

Explanation: The input message provided by the Java program for the IMS application program contains a value for its segment length which is either negative, 0, or greater than the number of bytes of data in the message segment.

User Action: Provide the correct value for the segment length of the input message.

ICO0025E

javax.resource.IllegalArgumentException:
ICO0025E:methodname error.
Invalid segment length (LL) of [llvalue] in OTMA message.

Explanation: The output message provided by the IMS application program contains a value for its segment length which is either negative, 0, or greater than the number of bytes of data in the message segment. The output message provided by the IMS application program is contained in the OTMA message.

User Action: Ensure that your IMS application program provides valid lengths for the segments of its output message.

ICO0026E

javax.resource.ResourceException:
ICO0026E:methodname error.
An error was encountered while processing the IMS message.
[source_methodname:source_exception]

Explanation: An error occurred while processing the IMS transaction input or output message. *source_exception* provides additional information regarding the cause of the error.

User Action: Examine *source_exception* for additional information regarding the cause of the error. Some suggested actions to take, based on the value of *source_exception* are:

- **java.io.IOException**

Error preparing input or output record. Ensure that the objects you are providing to IMS Connector for Java for use as the IMS transaction input and output are defined properly for the J2C architecture. For example, ensure that they implement the interfaces **javax.resource.cci.Record** and **javax.resource.cci.Streamable**.

- **com.ibm.ims.ico.IMSConnResourceException**

The OTMA message containing the IMS transaction output message contained an invalid length field (i.e., LLLL was ≤ 0). If this error continues to occur after verifying that your IMS application program is returning a valid output message, contact your IBM service representative.

- **java.lang.IllegalArgumentException**

The output message returned from IMS Connect is invalid. Ensure that the release levels of IMS Connector for Java and IMS Connect are compatible. For example, if you built a transactional required EJB application to perform a two phase commit transaction via TCP/IP by using IMS Connector for Java version 2.1, but at runtime, you are using IMS Connect version 1.2 instead of version 2.1, you will receive this error message. Hereby, either you update to IMS Connect version 2.1 or create a none global transactional EJB application.

ICO0030E

javax.resource.spi.ApplicationServerInternalException:
ICO0030E:methodname error.
[source_methodname:source_exception]

Explanation: A runtime error or exception was detected in *methodname* during the interaction. *source_methodname:source_exception* indicates where the error or exception that was detected in *methodname* originally occurred and may provide additional information regarding the cause of the error.

User Action: Examine *source_exception* for additional information regarding the cause of the error. The action(s) to be taken depend on the value of *source_methodname:source_exception*. Some suggested actions to take, based on the value of *source_methodname:source_exception* are:

- `java.lang.OutOfMemoryError`
This error is thrown when the Java Virtual Machine cannot allocate an object because it is out of memory, and no more memory could be made available by the garbage collector. Increase the amount of memory available to the virtual machine used by WAS.
- `java.io.InterruptedIOException`
An `InterruptedIOException` is thrown to indicate that an input or output transfer has been terminated because the thread performing it was terminated. Investigate reasons why the thread may have been terminated.

ICO0031E

```
javax.resource.spi.IllegalStateException:  
ICO0031E:methodname error.  
Protocol violation. The Interaction Verb [interactionverb] is not allowed for  
the current state [state].  
[java_exception]
```

Explanation: The interaction attempted by the application resulted in a protocol violation. [*interactionverb*] is the value of the `interactionVerb` property of the `IMSInteractionSpec` object that was used for the interaction. [*state*] is the current state of the protocol used for the interactions between IMS Connector for Java and IMS Connect.

For example, a protocol violation would occur if your Java program is not in conversation with IMS, but attempted an interaction with IMS using the `SYNC_END_CONVERSATION` value for the `interactionVerb` property.

User Action: Ensure that you are using an appropriate value for the `interactionVerb` property of `IMSInteractionSpec`. Check the IMS Connector for Java documentation for values of the `interactionVerb` property that are supported by IMS Connector for Java. A particular release of IMS Connector for Java may not support all the values defined by the J2EE Connector Architecture.

ICO0034E

```
javax.resource.NotSupportedException:  
ICO0034E:methodname error.  
Auto-commit not supported.
```

Explanation: Auto-commit is currently not supported by IMS Connector for Java.

User Action: Ensure that your Java application uses classes and methods that are appropriate for the level of support currently provided by IMS Connector for Java.

ICO0035E

```
javax.resource.NotSupportedException:  
ICO0035E:methodname error.  
Local Transaction not supported.
```

Explanation: Local Transactions are not currently supported by IMS Connector for Java.

User Action: Ensure that your Java application uses classes and methods that are appropriate for the level of support currently provided by IMS Connector for Java.

ICO0037E

javax.resource.NotSupportedException:
ICO0037E:*methodname* error.
ResultSet not supported.

Explanation: ResultSets are currently not supported by IMS Connector for Java.

User Action: Ensure that your Java application uses classes and methods that are appropriate for the level of support currently provided by IMS Connector for Java.

ICO0039E

javax.resource.spi.IllegalStateException:
ICO0039E:*methodname* error.
Not in CONNECT state.

Explanation: The sequence of interactions between IMS Connector for Java and IMS Connect is invalid. The current state of the protocol used for the interactions between IMS Connector for Java and IMS Connect is not CONNECT as it needs to be at this point in the interactions.

User Action: This is most likely an error in IMS Connector for Java or IMS Connect. Contact your IBM service representative.

ICO0040E

javax.resource.NotSupportedException:
ICO0040E:*methodname* error.
IMSConnector does not support this version of execute method.

Explanation: IMS Connector for Java does not support the form of the execute method that takes two input parameters and returns an object of type `javax.resource.cci.Record`.

User Action: Use the supported form of the execute method in class `IMSInteraction`. The supported form of the execute method has the following signature:

`boolean execute(InteractionSpec, Record input, Record output)`

ICO0041E

javax.resource.ResourceException:
ICO0041E:*methodname* error.
Invalid interactionSpec specified [*interactionSpec*].

Explanation: An invalid `InteractionSpec` object was passed to the execute method of class `com.ibm.connector2.ims.ico.IMSInteraction`.

User Action: Ensure that the `InteractionSpec` object that you pass to the execute method of class `com.ibm.connector2.ims.ico.IMSInteraction` is of type `com.ibm.connector2.ims.ico.IMSInteractionSpec`.

ICO0042E

javax.resource.ResourceException:
ICO0042E: *methodname* error.
Input is not of type Streamable.

Explanation: The input object provided to the execute method of **com.ibm.connector2.ims.ico.IMSInteraction** for the "input" parameter was either null or did not implement the interface **javax.resource.cci.Streamable**. This exception most likely occurs when an application is written to use the J2EE Connector Architecture Common Client Interface (CCI). This exception should not occur if WebSphere Studio Application Developer Integration Edition is used to build the input message.

The execute method allows null input objects for some types of interactions. For example, interactions with interactionVerb values of SYNC_END_CONVERSATION and SYNC_RECEIVE_ASYNCPUT allow null input objects.

User Action: Ensure that you are providing a valid **javax.resource.cci.Record** object for the "input" parameter to the execute method. For example, ensure that this object implements the interfaces **javax.resource.cci.Record** and **javax.resource.cci.Streamable**.

ICO0043E

javax.resource.ResourceException:

ICO0043E: *methodname* error.

Output is not of type Streamable.

Explanation: The output object provided to the execute method of **com.ibm.connector2.ims.ico.IMSInteraction** was either null or did not implement the interface **javax.resource.cci.Streamable**. This exception most likely occurs when an application is written to use the J2EE Connector Architecture Common Client Interface (CCI). This exception should not occur if WebSphere Studio Application Developer Integration Edition is used to build the output message.

User Action: Ensure that you are providing a valid output object to the execute method.

ICO0044E

javax.resource.NotSupportedException:

ICO0044E: *methodname* error.

RecordFactory is not supported by IMS Connector for Java.

Explanation: RecordFactory is currently not supported by IMS Connector for Java.

User Action: Ensure that your Java application uses classes and methods that are appropriate for the level of support currently provided by IMS Connector for Java.

ICO0045E

javax.resource.NotSupportedException:

ICO0045E: *methodname* error.

Invalid type of ConnectionRequestInfo.

Explanation: An invalid ConnectionRequestInfo object was passed to an IMS Connector for Java method.

User Action: This is most likely an error in IMS Connector for Java. Contact your IBM service representative.

ICO0049E

javax.resource.NotSupportedException:
IC00049E: *methodname* error.
The security credentials passed to getConnection do not match existing
security credentials.

Explanation: The security credentials in the request do not match the security credentials of the IMSManagedConnection instance that was being used to process the request.

User Action: Contact your IBM service representative.

ICO0053E

javax.resource.ResourceException:
IC00053E: *methodname* error.
Invalid clientID value. Prefix HWS is reserved by IMS Connector for Java.

Explanation: The value specified for the property clientID is invalid. The prefix 'HWS' is reserved by IMS Connector for Java.

User Action: Provide a valid value for clientID property. A valid value should follow the following rules:

- is not a null string;
- does not start with a blank field;
- does not start with IMS Connector for Java reserved prefix 'HWS';
- is 8 characters long;
- uses valid characters A - Z, 0 - 9, and @, #, \$.

ICO0054E

javax.resource.ResourceException:
IC00054E: *methodname* error.
Invalid ConnectionSpec.

Explanation: IMS Connector for Java was unable to cast the connectionSpec provided for this connection to type IMSConnectionSpec. While the Common Client Interface will accept a connectionSpec object for any supported connector, IMS Connector for Java will only work with an IMSConnectionSpec or a derivative of IMSConnectionSpec as its connectionSpec.

User Action: Ensure that the connectionSpec used by your application is an IMSConnectionSpec or inherits from IMSConnectionSpec.

ICO0055E

javax.resource.ResourceException:
IC00055E: *methodname* error.
Failed to cast the connection object to IMSConnection.

Explanation: IMS Connector for Java was unable to cast the connection object allocated by the ConnectionManager for this connection to type IMSConnection. IMS Connector for Java will only work with an IMSConnection or a derivative of IMSConnection as its connection object. This error might be the result of a problem with the ConnectionManager.

User Action: Please contact your IBM service representative.

ICO0057E

```
javax.resource.spi.IllegalStateException:  
ICO0057E:methodname error.  
Invoked with invalid connection handle.
```

Explanation: The application is in an illegal state: the connection handle (IMSConnection instance) used for this interaction is not valid. This could occur if the application attempted to use a connection handle for a previously used connection or the handle for the wrong connection if the application has more than one connection open.

User Action: Ensure that the application is using the currently valid IMSConnection instance for that connection.

ICO0058E

```
javax.resource.ResourceException:  
ICO0058E:methodname error.  
Interactions SYNC_SEND_RECEIVE, SYNC_SEND, SYNC_RECEIVE_ASYNCOUTPUT,  
SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT and  
SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT interactions with Commit Mode 0  
are not supported with Local Option.
```

Explanation: You can use Local Option to communicate with IMS Connect only if your application using IMS Connector for Java with the selection of Commit Mode 1.

User Action: Ensure that your application using IMS Connector for Java is selected with Commit Mode 1. If you plan to run your application with Commit Mode 0, correct your application to use TCP/IP communication.

ICO0059E

```
javax.resource.ResourceException:  
ICO0059E: methodname error.  
SYNC_END_CONVERSATION interaction with Commit Mode 0 is not supported.
```

Explanation: Interaction SYNC_END_CONVERSATION with Commit Mode 0 is not supported.

User Action: IMS Connector for Java supports the interaction combination SYNC_END_CONVERSATION with Commit Mode 1, SYNC_SEND_RECEIVE with Commit Mode 0, and SYNC_RECEIVE_ASYNCOUTPUT with Commit Mode 0.

ICO0060E

```
java.lang.UnsatisfiedLinkError:  
ICO0060E:methodname error.  
Error loading Local Option native library: libname=libraryFileName.  
[source_exception].
```

Explanation: The Local Option native library cannot be found in any of the directories listed in the libpath.

User Action: Ensure that the Local Option native library exists in one of the directories in the LIBPATH environment variable. If you are running IMS Connector for Java in WebSphere Application Server for z/OS and OS/390, ensure that the full name of the directory that contains the Local Option native library file is defined in the LIBPATH environment variable for your J2EE server. For more information, see “Preparing the base operating system” in the WebSphere Application Server Version 6.0 Information Center .

ICO0061E

javax.resource.ResourceException:
ICO0061E:methodname error.
Local Option runs only in z/OS and OS/390.

Explanation: You can use Local Option to communicate with IMS Connect only if your application using IMS Connector for Java is running on the z/OS or OS/390 platform.

User Action: Ensure that your application using IMS Connector for Java is running on the z/OS or OS/390. Note that it is also required that your application (or more precisely, the Web server where your application is running) must be running in the same MVS image as IMS Connect. If this is not the case, for example, if you plan to run your application on a workstation platform or if the Web server where you plan to run your application is on z/OS but in a different MVS image than IMS Connect, ensure that the connection factory used by your application is set up to use TCP/IP communication.

ICO0062E

javax.resource.ResourceException:
ICO0062E:methodname error.
Error loading Local Option native method: libfilename=libraryFileName,
methodname=nativeMethodName. [source_exception].

Explanation: The Local Option native method cannot be found.

User Action: Verify that you have the correct level of IMS Connector for Java resource adapter and Local Option native library installed on your system. Always use the version of the Local Option native library that shipped with the IMS resource adapter that you installed in your WebSphere Application Server for z/OS and OS/390 system. See "Prerequisites for using IMS Connector for Java" for more information.

ICO0063E

javax.resource.spi.ResourceAdapterInternalException:
ICO0063E:methodname error.
Exception thrown in native method. [source_exception].

Explanation: An internal error occurred in the Local Option native method.

User Action: Contact your IBM service representative.

ICO0064E

javax.resource.spi.SecurityException:
ICO0064E:methodname error.
Invalid security credential.

Explanation: The subject provided by WebSphere Application Server did not contain a security credential available that is supported by IMS Connector for Java.

User Action: Ensure that you have the correct level of WebSphere Application Server for z/OS and OS/390 installed. See the "Prerequisites for using IMS Connector for Java" section for details. Configure WebSphere Application Server for z/OS and OS/390 to provide a security credential that is supported by IMS Connector for Java. IMS Connector for Java supports the PasswordCredential for TCP/IP connections and the UToken GenericCredential for Local Option connections.

ICO0065E

javax.resource.spi.SecurityException:
ICO0065E:*methodname* error.
Error obtaining credential data from the security credential.[*source_exception*].

Explanation: There was a security related error in obtaining the credential data from the security credential provided by the application server.

User Action: Ensure that you have correctly set up security for your application server so that the user associated with the calling program is authorized to extract the data from a security credential.

ICO0066E

javax.resource.ResourceException:
ICO0066E:*methodname* error. Error loading WebSphere Application Server Transaction Manager. [*source_exception*].

Explanation: An error occurred when accessing the Transaction Manager of the WebSphere Application Server for processing the transaction request.

User Action: Ensure that you have the correct level of WebSphere Application Server for z/OS and OS/390 installed. See the "Prerequisites for using IMS Connector for Java" section for details.

ICO0068E

javax.resource.ResourceException:
ICO0068E:*methodname* error.
Error obtaining the transaction object. [*java_exception*]

Explanation: An error occurred while attempting to determine if a transaction has been started using the WebSphere Application Server Transaction Manager.

User Action: Ensure that you have the correct level of WebSphere Application Server for z/OS and OS/390 installed. See the "Prerequisites for using IMS Connector for Java" section for details.

ICO0069E

javax.resource.spi.ResourceAllocationException
ICO0069E:*methodname* error.
Error obtaining RRS transaction context token.
IMSConnResourceException: RRS retcode=[*rrs_routinecode*].

Explanation: An unexpected internal error occurred while obtaining an RRS transaction context token for processing the global transaction.

User Action: Check the RRS job log for associated RRS error messages. For diagnostic information on the RRS return code (*rrs_routinecode*) see the *MVS Programming: Resource Recovery* manual for your release of z/OS or OS/390.

ICO0070E

javax.resource.spi.EISSystemException
ICO0070E:*methodname* error.
IMS Connect reported an RRS error: IMS Connect Return Code=[*returncode*],
RRS Routine name=[*rrs_routine*], RRS Return code=[*rrs_routinecode*]."

Explanation: IMS Connect returned an error resulting from an RRS failure.

User Action: Check the MVS console for associated IMS Connect and RRS error messages. For diagnostic information on the return code (*returncode*) value, as well as IMS Connect error messages, see the *IMS Connect Guide and Reference*. For diagnostic information on the RRS return code (*rrs_routinecode*) locate the RRS routine name (*rrs_routine*) within the *MVS Programming: Resource Recovery* manual for your release of z/OS or OS/390.

ICO0071E

```
javax.transaction.xa.XAException  
ICO0071E:methodname error.  
A communication error occurred when processing the XA  
commandtype operation. [java_exception]
```

Explanation: There are numerous reasons why a communication failure could have occurred during the processing of a global transaction. A TCP/IP or socket failure could have taken place or IMS Connect could have been brought down. The connection in error will not be reused.

User Action: Examine the *java_exception* to determine the reason for the failure to connect to the host. Also check the MVS console for associated IMS Connect or TCP/IP error messages. Validate that machine can be reached through TCP/IP and that IMS Connect has not been brought down. The command type (*commandtype_string*) displayed in the error message refers to the stage at which this communication failure occurred during the global transaction: prepare, commit, rollback, recover, or forget.

ICO0072E

```
javax.transaction.xa.XAException:  
ICO0072E:methodname error.  
The associated UR for the Xid is not found.
```

Explanation: During transaction processing a UR that was tied to a specific Xid was eliminated by manual intervention or an error in IMS Connect or RRS.

User Action: Refer to the *WebSphere Application Server InfoCenter Reference Library* for steps on how to acquire transaction information and Xids within the WebSphere Application Server logs. Refer to the *IMS Connect Guide and Reference* for IMS Connect commands that will list out the Xid and their associated UR. Verify that a UR is listed for that Xid. Verify that the global transaction was not left in a heuristic state.

ICO0073E

```
javax.transaction.xa.XAException:  
ICO0073E:methodname error.  
RRS is not available.
```

Explanation: RRS has been brought down or communication between RRS and IMS Connect has ended.

User Action: Check the MVS console for associated IMS Connect and RRS error messages. Ensure that RRS has not been brought down on your z/OS or OS/390 system. Refer to the *IMS Connect Guide and Reference* for IMS Connect commands that can be used to verify that it is RRS enabled.

ICO0074E

```
javax.transaction.xa.XAException:  
ICO0074E: The RRS rrs_routine call returns with a return code [rrs_routinecode].
```

Explanation: During the processing of your global transaction the following RRS error message was passed in by IMS Connect.

User Action: Check the MVS console for associated IMS Connect and RRS error messages. For diagnostic information on the RRS return code (*rrs_routinecode*) locate the RRS routine name (*rrs_routine*) within the *MVS Programming: Resource Recovery* manual for your release of z/OS or OS/390.

ICO0075E

```
javax.transaction.xa.XAException:  
ICO0075E:methodname error.  
The transaction branch may have been heuristically completed. [rrs_exception]
```

Explanation: An RRS error has been passed in by IMS Connect that indicates that the processing of your transaction may have been affected in such a way as to leave it in a heuristic situation. It reveals a possibility that part of the transaction committed and part of it encountered an error during the commit phase which may have prevented it from committing. The *rrs_exception* is an ICO0074E error message indicating the RRS routine and return code associated with this issue.

User Action: Refer to the documentation of the ICO0074E error for more information regarding the RRS error message. Refer to the *WebSphere Application Server InfoCenter Reference Library* for steps on how to acquire transaction information and Xids within the WebSphere Application Server logs. Refer to the *IMS Connect Guide and Reference* for IMS Connect commands that will list out the Xid and their associated UR. Determine the Xid and URs involved and the result that should have been committed to IMS. Verify values within IMS to ensure that a heuristic state has occurred. A decision must then be made to take actions to rectify the data within IMS so that it matches the result that would have been committed or to rectify the other databases involved to return to a state prior to the execution of that transaction.

ICO0076E

```
javax.resource.ResourceException:  
ICO0076E:methodname error. An internal error occurred. [rrs_exception]
```

Explanation: An internal error occurred while trying to extract information about an RRS error message passed in by IMS Connect. The *rrs_exception* is an ICO0074E error message indicating the RRS routine and return code associated with the error.

User Action: Refer to the documentation of the ICO0074E error for more information regarding the RRS failure that has taken place. Please contact your IBM service representative.

ICO0077E

```
javax.resource.ResourceException:  
ICO0077E:methodname error. The transaction has already rolled back. [rrs_exception]
```

Explanation: An RRS error has been passed in by IMS Connect that indicates the attempt to rollback a transaction has been made a second time upon the same UR. RRS will prevent the second rollback from taking place and throw an error indicating that such an action is being attempted. The *rrs_exception* is an ICO0074E error message indicating the RRS routine and return code associated with the error.

User Action: No action is needed as the transaction should be rolled back. Refer to the documentation of the ICO0074E error for more information regarding the RRS

failure that has taken place. As a precaution, verify that data prior to the execution of the transaction has not been lost or modified.

ICO0078E

javax.resource.ResourceException:

ICO0078E: *methodname* error.

A valid user-specified clientID is required for interactions on a dedicated persistent connection.

Explanation: A valid, user-specified value is required for the clientID property when a value of 0 is specified for the commitMode property, and the interaction is using a dedicated persistent socket connection. This applies to SYNC_SEND_RECEIVE, SYNC_SEND, SYNC_RECEIVE_ASYNCOUTPUT, SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT and SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT interactions.

User Action: Provide a valid value for the clientID property. A valid value should follow the following rules:

- is not a null string
- does not start with a blank field
- does not start with IMS Connector for Java reserved prefix 'HWS'
- is 8 characters long
- has valid characters A - Z, 0 - 9, and @, #, \$

ICO0079E

com.ibm.connector2.ims.ico.IMSDFSMessageException:

ICO0079E: *methodname* error.

IMS returned DFS message: *DFS_message*

Explanation: IMS returned the message indicated by *DFS_message* instead of the output of the IMS transaction. This exception is thrown if the interaction uses the value IMS_REQUEST_TYPE_IMS_TRANSACTION for the imsRequestType property of IMSInteractionSpec.

For example, if the Java application attempts to run an IMS transaction that is stopped, this exception is thrown and the value of *DFS_message* is

DFS065 hh:mm:ss TRAN/LTERM STOPPED

User Action: Find the explanation and response that corresponds to *DFS_message* in the *IMS Messages and Codes* documentation, then address the problem in IMS.

ICO0080E

javax.resource.spi.EISSystemException:

ICO0080E: *methodname* error.

Execution timeout has occurred for this interaction. The executionTimeout was [*executionTimeout_value*] milliseconds. The IMS Connect TIMEOUT was used.

Explanation: The time it took for IMS Connect to send a message to IMS and receive the response was greater than the IMS Connect TIMEOUT value. The IMS Connect TIMEOUT value is:

- Specified in the IMS Connect configuration member for SYNC_SEND_RECEIVE interactions

- Two seconds for SYNC_RECEIVE_ASYNCOUTPUT, SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT, and SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT interactions

The reason of IMS Connect TIMEOUT value has been used is the executionTimeout property for this interaction was not specified or has been set to zero.

User Action: Ensure your application has set a valid executionTimeout value. To set the executionTimeout values, you can either use WebSphere Studio or use the setExecutionTimeout method. For detail instruction, please refer to the topic of [Setting execution timeout values in WebSphere Studio Application Developer Integration Edition 5.0.1 Help](#).

ICO0081E

```
javax.resource.spi.EISSystemException:
ICO0081E:methodname error.
Execution timeout has occurred for this interaction. The executionTimeout
value specified was [executionTimeout_value] milliseconds.
The value used by IMS Connect was
[rounded_executionTimeout_value] milliseconds.
```

Explanation: The time it took for IMS Connect to send a message to IMS and receive the response was greater than the executionTimeout value that was rounded to an appropriate execution timeout interval. Once a valid execution timeout value is set, this value is converted into a value that IMS Connect can use.

User Action: If the rounded execution timeout value is not what you expected, please verify with the follow table of conversion rules:

Range of user-specified values	Conversion rule
1 - 250	If the user-specified value is not divisible by 10, it is converted to the next greater increment of 10.
251 - 1000	If the user-specified value is not divisible by 50, it is converted to the next greater increment of 50.
1001 - 60000	The user-specified value is converted to the nearest increment of 1000. Values that are exactly between increments of 1000 are converted to the next greater increment of 1000.
60001 - 3600000	The user-specified value is converted to the nearest increment of 60000. Values that are exactly between increments of 60000 are converted to the next greater increment of 60000.

For more examples, please refer to the topic of [Valid execution timeout values in WebSphere Studio Application Developer Integration Edition 5.0.1 Help](#).

ICO0082E

```
javax.resource.NotSupportedException:
ICO0082E:methodname error.
Execution timeout has occurred for this interaction. The executionTimeout
```

value of [{*executionTimeout_value*}] milliseconds is not supported.
The valid range is [{*executionTimeout_waitforever_flag*}, 0 to
{*maximum_executionTimeout_value*}] milliseconds.
The IMS Connect TIMEOUT was used.

Explanation: The execution timeout value specified for the `executionTimeout` property was above or below the minimum or maximum timeout values respectively.

User Action: Ensure that your application has set a valid value for `executionTimeout` property. The execution timeout value is represented in milliseconds and must be a decimal integer in the range of 1 to 3600000, inclusively. Also it could be -1 if you want an interaction to run without a time limit.

ICO0083E

```
javax.resource.ResourceException::  
ICO0083E:methodname error.  
SYNC_SEND_RECEIVE, SYNC_SEND, SYNC_RECEIVE_ASYNCOUTPUT,  
SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT and  
SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT interactions with Commit Mode 0  
are not valid within the scope of a global transaction.
```

Explanation: `SYNC_SEND_RECEIVE`, `SYNC_SEND`, `SYNC_RECEIVE_ASYNCOUTPUT`, `SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT` and `SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT` interactions with Commit Mode 0 are not valid within the scope of a global transaction. Because currently the global transaction requires `SYNC_LEVEL_SYNCPOINT` and `SYNC_LEVEL_SYNCPOINT` only valid with Commit Mode 1.

User Action:

- If you want to use Commit Mode 0, ensure that your application is configured as a "non-transactional" application.
- If you want to run your interactions within the scope of a global transaction, then the `commitMode` property value must be 1.

ICO0084E

```
javax.resource.ResourceException:  
ICO0084E:methodname error.  
An unexpected internal IMS Connector for Java error occurred.  
[source_method] [source_exception]
```

Explanation: A `PrivilegedActionException` occurred while executing a `[source_method]` call in `methodname`. This exception will occur if Java 2 security is enabled and the user associated with the calling program, `methodname`, or any program in the current call stack is not authorized to execute `[source_method]`.

User Action: Ensure that you have correctly set up security for your application server so that the user associated with the calling program plus any programs in the current call stack at the time of the exception is/are authorized to execute `[source_method]`. Alternatively, you could turn off Java 2 security checking in the application server.

ICO0085E

```
javax.resource.ResourceException:  
ICO0085E: methodname error.  
Protocol violation. A user-specified clientID is not allowed for interactions  
on a shareable persistent socket.
```

Explanation: The value specified for clientID property is not allowed. Because the connection factory is configured for shareable persistent socket, a user-specified clientID is not allowed within this kind of connection factory.

User Action: For shareable persistent socket connection factory, IMS Connector for Java provides generated clientID. User-specified clientID is not allowed. To determine if you are using a shareable persistent socket, check for a value of FALSE for the CM0Dedicated property of the connection factory used by the interaction.

ICO0086E

```
javax.resource.ResourceException::  
ICO0086E:methodname error.  
Invalid value was specified for CommitMode property.
```

Explanation: The CommitMode value you have specified in the commitMode property field is invalid.

User Action: Ensure that your application has set a valid value for commitMode property. Values supported are:

- Value 1 (SEND_THEN_COMMIT), indicates that IMS processes the transaction and sends a response back before committing the data.
- Value 0 (COMMIT_THEN_SEND), indicates that IMS processes the transaction and commits the data before sending a response.

ICO0087E

```
javax.resource.ResourceException:  
ICO0087E: methodname error.  
Protocol violation. Commit Mode 1 is not allowed for interactions on a  
dedicated persistent socket.
```

Explanation: The value 1 specified for Commit Mode property is invalid. Because the connection factory is configured for dedicated persistent socket, Commit Mode 1 is not allowed within this kind of connection factory.

User Action: For dedicated persistent socket connection factory, Commit Mode 0 interactions are valid. To determine if you are using a dedicated persistent socket check for a value of TRUE for the CM0Dedicated property of the connection factory used by the interaction.

ICO0088E

```
javax.resource.ResourceException:  
ICO0088E: methodname error.  
Protocol violation. SYNC_RECEIVE_ASYNCOUTPUT interactions are not allowed  
on a shareable persistent sockets.
```

Explanation: The value SYNC_RECEIVE_ASYNCOUTPUT specified for interactionVerb property is invalid. Because the connection factory is configured for shareable persistent socket, SYNC_RECEIVE_ASYNCOUTPUT is not allowed within this kind of connection factory.

User Action: SYNC_SEND_RECEIVE, SYNC_SEND, and SYNC_END_CONVERSATION are the valid values for the interactionVerb property for interactions on a shareable persistent connection. To determine if you are using a shareable persistent connection, check for a value of FALSE for the CM0Dedicated property of the connection factory used by the interaction.

ICO0089I

```
javax.resource.ResourceException::  
ICO0089I: methodname.  
Non-persistent socket closed for Commit Mode 0 IMS transaction.
```

Explanation: Running CommitMode 0 with non-persistent socket (transaction socket), IMS Connector for Java will force removal of managed connection object from Connection Pool.

User Action: This is not an error message, no action required.

ICO0091E

```
javax.resource.ResourceException:  
ICO0091E: methodname  
error.SSL client context could not be created. [{1}]
```

Explanation: An SSL Context could not be created due to one of the following reasons:

- The algorithm used to check the integrity of the keystore cannot be found
- The certificates in the keystore could not be loaded
- The key cannot be recovered (e.g. the given password is wrong).

User Action: Ensure the following:

- The algorithm used to create certificates must be one that is supported by IBMJSSE.
- The passwords for the keystore and truststore are correct.

ICO0096I

```
javax.resource.ResourceException:  
ICO0096I: methodname  
Warning. Invalid value provided for SSL parameter.
```

Explanation: The method indicated in *methodname* was invoked using a null or empty SSLKeystoreName, SSLKeystorePassword, SSLTruststoreName or SSLTruststorePassword parameter. This is an informational message to let the user know that one of the above-mentioned parameters is a null or an empty string. This will not terminate the program execution.

User Action: Provide valid values for SSLKeystoreName, SSLKeystorePassword, SSLTruststoreName and SSLTruststorePassword parameters. For convenience, private keys and certificates can be stored either in a keystore or a truststore. Therefore only one set of valid values (either SSLKeystoreName and SSLKeystorePassword or SSLTruststoreName and SSLTruststorePassword) are required for proper execution.

ICO0097E

```
javax.resource.ResourceException:  
IC00097E:methodname error.  
{0} error. The given value is invalid for 'SSLEncryptionType'.  
The value must be 'STRONG' for strong encryption or 'WEAK'  
for weak encryption.
```

Explanation: A value other than strong or weak was provided for the SSLEncryptionType parameter.

User Action: Provide either strong or weak for the SSLEncryptionType parameter. The value is not case-sensitive.

ICO0111E

```
javax.resource.ResourceException:  
ICO0111E:methodname error.  
SSLEnabled must be set to FALSE when using Local Option.
```

Explanation: The property IMSConnectName is set to a non-null value and the property SSLEnabled is set to true. However, SSL is not supported on local option connections (which is indicated by providing a value for IMSConnectName parameter).

User Action: Set SSLEnabled to false.

ICO0113E

```
javax.resource.spi.CommException:  
ICO0113E: methodname error.  
Socket Timeout has occurred for this interaction. The Socket Timeout value  
specified was [socket timeout value] milliseconds.  
[source_exception:exception_reason]
```

Explanation: The time for IMS Connector for Java to receive a response from IMS Connect is greater than the time specified for Socket Timeout.

User Action: Ensure that the time value of Socket Timeout is sufficient for IMS Connector for Java to receive a response from IMS Connect. If it is not, increase the value. If the value of Socket Timeout given is sufficient, it is possible that network problems are causing delays. Contact your network administrator.

ICO0114E

```
javax.resource.ResourceException:  
ICO0114E: methodname error.  
The Socket Timeout Property value of [socket timeout value] is invalid.  
[source_exception:exception_reason]
```

Explanation: The value [socket timeout value] specified for the Socket Timeout property is not valid.

User Action: Review the exception_reason provided. Ensure a positive numerical value was given for Socket Timeout.

ICO0115E

```
javax.resource.spi.CommException:  
ICO0115E: methodname error.  
A TCP Error occurred.
```

Explanation: This is an error in the underlying protocol.

User Action: Contact your network administrator.

ICO0117E

javax.resource.ResourceException:
ICO0117E: *methodname* error.
Protocol violation: Commit Mode 1 is not allowed for SYNC_SEND,
SYNC_RECEIVE_ASYNCOUTPUT, SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT
and SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT interactions.

Explanation: The IMS resource adapter currently only supports Commit Mode 0 for SYNC_SEND interactions.

User Action: Commit Mode 0 is required for SYNC_SEND,
SYNC_RECEIVE_ASYNCOUTPUT,
SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT and
SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT ,
SYNC_RECEIVE_ASYNCOUTPUT,
SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT and
SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT interactions. Commit Mode 1 is
valid with SYNC_SEND_RECEIVE and SYNC_END_CONVERSATION
interactions.

ICO0118E

javax.resource.ResourceException:
ICO0118E: *methodname* error.
Protocol violation. IMS request type 2(IMS_REQUEST_TYPE_IMS_COMMAND)
is not allowed for SYNC_SEND, SYNC_END_CONVERSATION, SYNC_RECEIVE_ASYNCOUTPUT,
SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT and SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT
interactions.

Explanation: The value 2(IMS_REQUEST_TYPE_IMS_COMMAND) specified for
imsRequestType property is invalid.

User Action: ImsRequestType 2(IMS_REQUEST_TYPE_IMS_COMMAND) only
valid with SYNC_SEND_RECEIVE interaction. ImsRequestType
1(IMS_REQUEST_TYPE_IMS_TRANSACTION) is required for SYNC_SEND,
SYNC_END_CONVERSATION, SYNC_RECEIVE_ASYNCOUTPUT,
SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_NOWAIT and
SYNC_RECEIVE_ASYNCOUTPUT_SINGLE_WAIT interactions.

ICO0119E

javax.resource.ResourceException:
ICO0119E: *methodname* error.
A supported SSL provider was not found. [caught_exception]

Explanation: When attempting to initialize a Secure Sockets Layer TCP/IP
connection with IMS Connect, IMS Connector for Java needs to use one of the two
supported providers, com.ibm.jsse.JSSEProvider or sun.security.provider.Sun. This
error indicates that neither of these providers is available.

User Action: com.ibm.jsse.JSSEProvider should be added by default in an IBM
JVM and sun.security.provider.Sun should be added by default in a Sun JVM.
Ensure that you are running IMS Connector for Java in a supported IBM JVM if
running in WebSphere Application Server or a Sun JVM in other application
servers.

ICO0121E

```
javax.resource.ResourceException:  
IC00121E: methodname error.  
Invalid reRoute name value. Prefix HWS is reserved for use by  
IMS Connector for Java.
```

Explanation: The value specified for reRouteName property is invalid. The prefix 'HWS' is reserved for use by IMS Connector for Java.

User Action: Provide a valid value for reRouteName property. A valid value should adhere to the following rules:

- Is not a null string
- Does not start with a blank field
- Does not start with the IMS Connector for Java reserved prefix 'HWS'
- Is 8 characters long
- Uses the valid characters A - Z, 0 - 9, @, #, and \$

IC00122E

```
javax.resource.ResourceException:  
IC00122E: methodname error.  
Invalid reRoute value. When purgeAsyncOutput value is true, reRoute  
value cannot be true.
```

Explanation: The value specified for reRoute property is invalid. Because the value specified for purgeAsyncOutput property is TRUE, or the default value (TRUE) is used for purgeAsyncOutput property.

User Action: Ensure to set purgeAsyncOutput property to FALSE, if you want to set reRoute to TRUE.

Chapter 8. Migration and coexistence

The topics in this section describe migrating from WebSphere Studio Application Developer Integration Edition Version 5.0.1 to Version 5.1. This section also includes compatibility issues between existing applications and different versions of the IMS resource adapter.

Migration and coexistence considerations for the IMS resource adapter

In WebSphere Studio, there is no traditional migration path for VisualAge® for Java Enterprise Access Builder (EAB) applications, regardless of whether the applications are J2EE or CCF. However, you can still work with your EAB applications in WebSphere Studio using one of several options that employ either coexistence or reengineering techniques. The options include:

- Working with EAB applications using standard WebSphere Studio Application Developer tools
- Working with EAB applications using WebSphere Studio Application Developer Integration Edition tools
- Reengineering EAB applications in WebSphere Studio

If you choose to re-engineer your EAB applications, you can use the information in *Creating an enterprise service for an IMS transaction* to help you understand the service model.

For more information on migration, see *Migrating to WebSphere Studio Application Developer Integration Edition, V5.1*, which is also available in the *README* located in the root directory of the product distribution CD.

Compatibility of existing applications with IMS Connector for Java Version 2.2.1

WebSphere Studio Application Developer Integration Edition Version 5.1 corrects a problem with the code generated for a deployed enterprise service. Applications generated with WebSphere Studio Application Developer earlier than version 5.1 will fail when run in WebSphere Studio Unit Test Environment and in WebSphere Application Server that have Version 2.2.1 of the IMS resource adapter deployed.

The symptoms of failure will vary, depending on the type of socket being used for the interaction because Version 2.2.1 of IMS Connector for Java introduces two types of TCP/IP socket connections, **shareable persistent** and **dedicated persistent**.

- For applications running on a **shareable persistent socket**:

Starting with Version 2.2.1 of the IMS resource adapter, execution timeout on a shareable persistent socket **does not** result in the socket being disconnected by IMS Connect. The shareable persistent socket on which the execution timeout occurred will be reused for subsequent interactions.

If the application does not use the new generated code, the shareable persistent socket on which the execution timeout occurred will not be reused for subsequent interactions, resulting in an increase in the number of socket connections to IMS Connect.

- For applications running on a **dedicated persistent socket**:

Starting with Version 2.2.1 of the IMS resource adapter, an application that needs to recover undelivered output messages from a Commit Mode 0 interaction must run on an dedicated persistent socket. Execution timeout on a dedicated persistent socket **does not** result in the socket being disconnected by IMS Connect. The dedicated persistent socket on which the execution timeout occurred will be reused for subsequent interactions.

If the application does not use the new generated code, subsequent interactions on the dedicated persistent socket on which the execution timeout occurred will result in the following exception:

```
javax.resource.spi.EISSystemException: IC00001E:
com.ibm.connector2.ims.ico.IMSTCPManagedConnection@5c762147.
processOutputOTMAMsg(byte [],InteractionSpec, Record) error.IMS Connect returned
error: RETCODE=[8], REASONCODE=[DUPECLNT].Duplicate client ID was used;
the client ID is currently in use.
```

Regenerating code for existing applications

Note: This only applies to applications that are generated by releases of WebSphere Studio prior to Version 5.1.

Part of the deploy code generated by WebSphere Studio is a session bean that synchronously handles client requests. The generated code changed for this bean. To regenerate deploy code for an existing application perform the following steps. These steps show how to regenerate the deploy code for the IMS PhoneBook Sample provided in WebSphere Studio Application Developer Integration Edition Version 5.0.1. The EAR for the IMS PhoneBook Sample Version 5.0.1 can be found in:

```
<wsadie_install_dir>\eclipse\plugins\com.ibm.etools.ctc.samples.ims\
IMSPhoneBookServiceEAR501.ear
```

1. Import the EAR for the application into the Business Perspective of WebSphere Studio:
 - a. **File -> Import -> EAR file -> Next**
 - b. In the Enterprise Application Import wizard, navigate to the EAR of the application you wish to regenerate. For the IMS PhoneBook Sample Version 5.0.1, navigate to the IMSPhoneBookServiceEAR501.ear file. Accept the value in the **Project name** field and select **Next**
 - c. In the Import Defaults wizard, in the Utility JARs section, select the JAR file for the IMS enterprise service project. For the IMS PhoneBook Sample Version 5.0.1, select IMSPhoneBookService.jar. Depending on your Enterprise Application, you may also need to select additional JARs such as the JAR for a project containing a client application. For the IMS PhoneBook Sample Version 5.0.1, select IMSCClientSample.jar.
 - d. In the Project Import section, ensure that the **Expanded: extract project contents for development** radio button is selected and then click **Finish**
 - e. Your workspace will be updated with the IMS enterprise service project, the EJB project, and other projects in the EAR, such as a Web project. For the IMS PhoneBook Sample Version 5.0.1, your workspace will contain the IMS enterprise service project, IMSPhoneBookService; the EJB project, IMSPhoneBookServiceEJB; the Web project, IMSPhoneBookServiceWeb; and the client application project, IMSCClientSample.

- f. **Note:** At this point you may need to update the Java Build Path of the imported projects. For example, in the Package Explorer view right-click on the project whose Java Build Path you wish to update and select **Properties > Java Build Path**.

For an IMS enterprise service project

- 1) In the **Projects** tab, select the project into which you imported the IMS resource adapter.
- 2) In the **Libraries** tab, click **Add Library**.
- 3) Select the variable, **WAS_EE_V51**, and click **Extend** and add the following JARs from WAS_EE_V51/lib folder:
 - commons-logging-api.jar
 - ibmjsse.jar
 - j2ee.jar
 - marshall.jar
 - physicalrep.jar
 - qname.jar
 - soap.jar
 - wsatlib.jar
 - wsdl4.jar
 - wsif.jar
 - wsif-j2c.jar
- 4) In the **Libraries** tab, click **Add Library** and select the variable, **WAS_EE_V51**. Click **Extend** and add the following JAR from the WAS_EE_V51/java/jre/lib folder: **xml.jar**.

For an EJB project:

- a. In the Libraries tab, click **Add Library**. Select the variable, **WAS_EE_V51** and click **Extend**. Add the following JARs from the WAS_EE_V51/lib folder:
 - commons-logging-api.jar
 - marshall.jar
 - physicalrep.jar
 - qname.jar
 - soap.jar
 - wsatlib.jar
 - wsdl4.jar
 - wsif.jar
 - wsif-j2c.jar
- b. In the Libraries tab, click **Add Library**. Select the variable **WAS_EE_V51**, and click **Extend**. Add the following JARs from the WAS_EE_V51/java/jre/lib folder: **xml.jar**

For a Web project:

- a. In the Libraries tab, click **Add Library**. Select the variable **WAS_EE_V51** and click **Extend**. Add the following JARs from the WAS_EE_V51/lib folder:
 - commons-logging-appi.jar
 - qname.jar
 - soap.jar
 - wsadtlb.jar
 - wsdl4.jar

- wsif.jar
 - wsif-j2c.jar
- b. In the Libraries tab, click **Add Library**. Select the variable **WAS_EE_V51** and click **Extend**. Add the following JARs from the WAS_EE_V51/java/jre/lib/ext folder:
- activation.jar
 - mail.jar

For a project containing a client application:

- a. In the Libraries tab, click **Add Library**. Select the variable **WAS_EE_V51** and click **Extend**. Add the following JARs from the WAS_EE_V51/lib folder:
- qname.jar
 - soap.jar
 - wsatlib.jar
 - wsdl4.jar
 - wsif.jar
 - wsif-j2c.jar
- b. In the Libraries tab, click **Add Library**. Select the variable **WAS_EE_V51** and click **Extend**. Add the following JAR from the WAS_EE_V51/java/jre/lib folder: **xml.jar**
- c. In the Libraries tab, click **Add Library**. Select the variable **WAS_EE_V51** and click **Extend**. Add the following JARs from the WAS_EE_V51/java/jre/lib/ext folder:
- activation.jar
 - mail.jar

Note: The above JARs are provided as sample only. Each application has its own JAR dependencies.

2. Regenerate the deploy code:
 - a. **Note:** You may wish to save the existing files prior to this step.
 - b. In the Packages view, expand the IMS enterprise service project. For the IMS PhoneBook Sample Version 5.0.1, expand the project, **IMSPhoneBookService**.
 - c. In the IMS enterprise service project, right-click on the service wsdl file and select **Enterprise Services -> Generate Deploy Code**. For the IMS PhoneBook Sample Version 5.0.1, right-click on **PhoneBookIMSService.wsdl**.
 - d. In the Deploy a service wizard, provide the values appropriate to your application. Select **Create a new port and binding**.
 - e. Click **Finish**. When you select Finish, the Generate Deploy Code message box will be displayed indicating that the binding and port already exists. Select **OK** to indicate that you wish to overwrite their definitions.
3. Optionally test the application in the WebSphere Studio Unit Test Environment
4. Export a new EAR and deploy to WebSphere Application Server

Chapter 9. Samples

The topics in this section provide detailed steps on how to generate a sample. The samples are tutorials that show you how to work with an enterprise service. The topics included are:

Sample: Creating an enterprise service for an IMS transaction

Objectives

This sample has two main parts. Part 1: Creating the enterprise service demonstrates how to use WebSphere Studio tools to perform bottom-up development of an enterprise service. This is where you, the service provider, will:

- Generate an enterprise service from a COBOL representation of the input and output messages of an IMS transaction.
- Test the service by invoking it through a Java proxy.
- Deploy the service to a test version of WebSphere Application Server, and make the service available as either a SOAP or EJB service.

Part 2: Creating the client application demonstrates how you, the service consumer, can:

- Use other WebSphere Studio tools to generate the client-side proxy and build the client application to access the enterprise service.
- Deploy and test the generated enterprise service.

When you have completed the steps in this sample, you might be interested in reading the additional information in the What's next topic at the end of this document.

Tip: To get additional help on fields or buttons in a wizard, place the focus on a control and then press the **F1** key to view the context-sensitive help.

Time required

Allow 90 minutes. This will give you enough time to create the enterprise service as well as create the client application that will invoke the service.

Before you begin

In this sample, your client application uses the IMS resource adapter (also called IMS Connector for Java) to interact with IMS through the host product, IMS Connect. The sample service runs the PhoneBook IMS transaction on an IMS system that you specify. Since the PhoneBook IMS transaction is one of the IMS Installation Verification Programs, it is probably already installed on your IMS system.

Before running the sample:

- Contact your IMS system programmer to verify that the PhoneBook transaction is available.
- Ensure that your environment meets the prerequisites for using the IMS resource adapter.

- Import the IMS resource adapter, **ims.rar**. See Importing a resource adapter.

Important: If you want to run your application on a remote server, see Defining a WebSphere Server for publishing. The unit test environment in WebSphere Studio does not support running remote WebSphere Application Servers for z/OS.

Description

This sample leads you through detailed steps that describe how to generate an enterprise service based on a COBOL representation of the input and output messages of an IMS transaction. Within WebSphere Studio, you will use wizards to generate code for the service, and then deploy the code to the WebSphere test environment that is shipped with the WebSphere Studio product. You will also generate a client proxy to access the service, and you will create a sample client application. The client application provides input data for the IMS input message, which the service passes to the IMS system. The IMS transaction runs and returns an output message, the contents of which are returned to the client by the service. For this sample, you run all of the server and client applications on the same machine.

Part 1: Creating the enterprise service

Note: Before you can create the enterprise service, you must import the IMS resource adapter, **ims.rar**. See Importing a resource adapter.



In this part of the sample, you will complete the following tasks:

- Step 1: Creating the service project
- Step 2: Importing the COBOL file
- Step 3: Generating the enterprise service
- Step 4: Testing the generated enterprise service
- Step 5: Generating deploy code for the enterprise service
- Step 6: Binding the resource reference
- Step 7: Configuring the server and deploying the EAR project

Step 1: Creating the service project

The service project stores all of the files for your project, including imported source files and files generated by wizards. You will use the New Service Project wizard to create the project.

To create a service project, complete the following steps:

1. Open the Business Integration perspective by selecting **Window > Open Perspective > Business Integration** or click the Business Integration icon  on the left-vertical toolbar. You can click this icon at any time to return to the Business Integration perspective. See Business Integration perspective for more information on the perspective's different views.
2. From the toolbar, click the **Create a service project** icon . The New Service Project wizard opens.
3. Type myIMSPhoneBookService for the **Project Name**.
4. Select **Use default** to use the default location to store the new project.

5. Click **Finish** to create the project. You do not need to specify Java Build Path settings or dependent JAR files in subsequent pages of the wizard because these are automatically set for you.

Step 2: Importing the COBOL file


When you expand **Service Projects** in the Services view, you see the service project, myIMSPhoneBookService, which you have just created.

In this step, you import the COBOL copybook file that is needed to create your service definition. The Ex01.ccp file is located in:

```
WS_installdir\wstools\eclipse\plugins\com.ibm.etools.ctc.samples.ims_5.1.0\sampleparts
```

where *WS_installdir* is the directory where WebSphere Studio is installed. The Ex01.ccp file defines the structure of the input and output messages of the PhoneBook IMS transaction.

Before importing the Ex01.ccp file into the workbench, create a Java package to hold the file:

1. Select the **myIMSPhoneBookService** service project and click the **New Java Package** icon .
2. On the Java Package page, ensure that the **Source Folder** is **myIMSPhoneBookService**.
3. Type `sample.ims` for the name of the package and click **Finish**. The package is created in the myIMSPhoneBookService project.

Next, you need to import the COBOL copybook file into the **sample.ims** package:

1. In the Services view, expand the myIMSPhoneBookService service project and select the **sample.ims** package.
2. From the menu bar, select **File > Import** to open the Import wizard.
3. Select **File system** to import the resources from the local file system. Click **Next**.
4. Click **Browse** beside the directory field to locate the following directory:
`WS_installdir\wstools\eclipse\plugins\com.ibm.etools.ctc.samples.ims_5.1.1\sampleparts`

where *WS_installdir* is the directory where WebSphere Studio is installed. Click **OK**.

5. On the File system page, select the **IMS** folder and ensure that the check box is clear. In the right pane, select the **Ex01.ccp** check box.
6. Ensure that `myIMSPhoneBookService/sample/ims` is the name of the destination folder for the imported resource. **Create selected folders only** should also be selected. Click **Finish** to import the file and close the wizard.

If you are successful in importing the file, the Tasks view will not contain any errors and the **sample.ims** package will contain the **Ex01.ccp** file.

Tip: Many of the steps in the sample ask you to select artifacts before launching a wizard. As a result, many of the fields in the wizard contain default values based on the selected artifact. The default values make it much easier and faster for you to complete your tasks.

Now that you have imported the Ex01.ccp file, you can generate your enterprise service.

Step 3: Generating the enterprise service

The service definition is described in Web Services Description Language (WSDL), which is a standard for describing networked, XML-based services. WSDL provides a simple way to describe the basic format of system requests regardless of the underlying run-time implementation. A WSDL document describes where the service is deployed and what operations the service provides. WebSphere Studio tools for building enterprise services use WSDL as the model for describing any kind of service. Instead of generating a single WSDL file, WebSphere Studio separates the service into the following three WSDL files:

- The abstract service interface definition or interface WSDL file, which contains the port types and message elements.
- The binding WSDL file, which contains the binding elements that describe how the service interface is implemented. The tools for enterprise services support the following service-provider-specific bindings: SOAP, JMS, JCA, JavaBean, enterprise session beans, message-driven beans, flow, and transform.
- The service WSDL file, which contains the service and port elements that provide the service location as described by a service-provider-specific port binding.

For more information on these WSDL files, see “Service programming model” in the IBM WebSphere Business Integration Information Center .

To generate the enterprise service, complete the following steps:

1. Expand **Service Projects** > **myIMSPhoneBookService** > **sample.ims**.
2. Right-click **Ex01.ccp** and select **New** > **Service built from...**
3. In the Create Service page, select **IMS** and click **Next**.
Note: If IMS does not appear in the list of service providers, you need to import the IMS resource adapter, **ims.rar**, before proceeding with these steps.
4. In the Connection Properties page, type the property values appropriate for your environment. See Connection properties for a description of these properties. For example:
 - In the **Host name** field, type MYHOST.ABC.XYZ.COM
 - In the **Port number** field, type 9999
 - In the **Data store name** field, type MYDSTOR. **Note:** This field is case-sensitive.

Click **Next**. **Note:** Because the connection properties are not encrypted, you should remove at minimum the User name and password from the port definition after you have completed testing.

5. In the Service Binding page, ensure that the following values are correct:
 - The **Source folder** field contains /myIMSPhoneBookService
 - The **Package** field contains sample.ims
6. In the **Interface file name** field, type myPhoneBook. This file contains the interface that the service uses to send input and get output from the IMS transaction. In this sample, the service gets the results of the request, submitted by the client application, to run the PhoneBook transaction. When you type the interface file name, the wizard automatically enters values for the remaining fields in the Service Binding page.

7. Click **Finish** to accept all other default names. The wizard generates three WSDL files for the service:
 - The interface file, myPhoneBook.wsdl, which contains the port types and message elements.
 - The binding file, myPhoneBookIMSBinding.wsdl, which stores the operation and binding information.
 - The service file, myPhoneBookIMSService.wsdl, which stores the host information.
8. A Next Step Information window opens and asks when the Binding WSDL file opens in the editor, whether or not to proceed to the binding content. Select **Do not show this dialog again** and click **OK**. A WSDL editor opens with the bindings of **myPhoneBookIMSBinding.wsdl**.
9. In the **Bindings** container in the Graph view, right-click **myPhoneBookIMSBinding** and select **Generate Binding Content**.
10. The Specify Binding Details page opens. In the **Protocol** field, select **IMS**.
11. Click **Add** next to the **Add binding operations** field.
12. In the Operation Binding page, you specify a new operation. In the **Operation name** field type runPhoneBook. Leave the type of operation as REQUEST_RESPONSE because there will be two messages, one for the request to run the IMS transaction and one for the response from the IMS transaction. Click **Next**.
13. In the imsConnector Operation Binding Properties page, the properties of the interaction with the IMS application program are shown. Type the property values appropriate for your application. See Operation binding properties for a description of these properties. The PhoneBook application should not require you to change any of the default values. For example:
 - Ensure that the **imsRequestType** field is set to 1 to indicate that the interaction with IMS consists of running a transaction.
 - Ensure that the **interactionVerb** field is set to 1 to indicate that the interaction with IMS involves a send followed by a receive interaction.
 Click **Next**.
14. In the Operation Binding page, you create new input and output messages. Click **Import** next to the input message. The File Selection page opens. Import the Ex01.ccp file to specify the XML schema definition for the input part.
 - a. Expand **myIMSPhoneBookService > sample > ims** and select **Ex01.ccp**. Click **Next**.
 - b. In the COBOL Import Properties page, enter the following values:
 If you choose the z/OS platform, the values for all the fields except TRUNC will be automatically filled as shown in the following table. Because most IMS programs are compiled with the TRUNC(BIN) option, it is recommended that you change the value of TRUNC from STD to BIN.

Platform	z/OS
Codepage	037
Floating point format	IBM 390 Hexadecimal
Endian	Big
Remote integer endian	Big
External decimal sign	EBCDIC

QUOTE	DOUBLE
TRUNC	BIN
NSYMBOL	DBCS

Other values in the table above might differ for your environment. For example, you might need to specify a different value for the **Codepage** field if your IMS data is in a code page other than U.S. English (037). The value for the **QUOTE** field also might differ, depending on your COBOL source. Click **Next**.

- c. In the COBOL Importer page, the data structures from the file are displayed. Select **INPUTMSG**, which will populate the XSD type name with **INPUTMSG**. You can accept the default to overwrite the XSD types. Click **Finish**.

Invoking the `runPhoneBook` method will result in transaction input information being passed from the application to the input message and then to the EIS. Then the transaction output would be returned from the EIS to the output message and then to the application.

15. In the Operation Binding page, click **Import** next to the output message. The File Selection page opens. Import the `Ex01.ccp` file to specify the XML schema definition for the output part.
 - a. Expand **myIMSPhoneBookService > sample >ims** and select **Ex01.ccp**. In this sample, both the input and output message definitions are contained in the same COBOL source file, `Ex01.ccp`. Click **Next**.
 - b. In the COBOL Import Properties page, specify the same values that you entered in step 12b for input. Click **Next**.
 - c. In the COBOL Import window, select **OUTPUTMSG** in the data structures list, which will populate the XSD type name with **OUTPUTMSG**. You can accept the default to overwrite the XSD types. Click **Finish**.
 - d. In the Operation Binding page, click **Next**.
 - e. In the Operation Binding summary page, the new operation information is displayed. Click **Finish**. The wizard populates WSDL files with the operation information and saves the information to the `sample.ims` package of the `myIMSPhoneBookService` project.
16. Click **Finish** on the **Binding Wizard** page.
17. Press **Ctrl-S** to save the changes in the `myPhoneBookIMSBinding.wsdl` file and then close the editor.

Now you can create a proxy to test the service that you have just created.

Step 4: Testing the generated enterprise service

To test the service, you build a Java service proxy to access the service, and then you write code to test the proxy. To create and test the proxy, complete the following tasks:

1. Creating the Java service proxy.
2. Testing the Java service proxy.

Creating the Java service proxy

The Java service proxy provides a remote procedure call interface to the service. Using the proxy, the application calls a remote method on the service as if the

method were a local one. Once the application makes the remote call, the proxy handles all of the communication details between the application and the service.

To create the Java service proxy, complete the following steps:


1. Expand the **myIMSPhoneBookService** project and the **sample.ims** package. Select the service file **myPhoneBookIMSService.wsdl**.
2. Right-click the file and select **Enterprise Services > Generate Service Proxy**. The Generate Service Proxy wizard opens.
3. In the **Proxy selection** page, select **Web Services Invocation Framework** for the type of proxy to generate. Click **Next**.
4. Ensure that the service you want to create the proxy for is shown. Because you selected a file in the first step, most fields are populated with default values. These default values are generated based on the contents of the selected service file.
 - a. Change the class name for the proxy to **myPhoneBookIMSPProxy.java**, if necessary.
 - b. Ensure that the package name is **sample.ims**.
 - c. Ensure that **Generate helper classes** is selected. These Java helper classes are required by your service. Click **Next**.
5. In the Service Proxy page, specify the style of the proxy and the operations to expose in the proxy:
 - a. Select the **Client stub** proxy style.
 - b. Select the **runPhoneBook** check box to select the operations to include in the proxy.
6. Click **Finish**. The Java service proxy (myPhoneBookIMSPProxy) is generated in the myIMSPhoneBookService project.

Next, you need to write a Java class to test the proxy.

Testing the Java service proxy

To test the Java service proxy, you need to write client code that executes the proxy. This code sets parameters for the input message, invokes the proxy, passes the input message to the proxy, receives an output message back from the proxy, and then displays the message on the console. You use the service project you created named myIMSPhoneBookService to store your code.

To write the Java class, complete the following steps:

1. Expand the **myIMSPhoneBookService** project and then select the **sample.ims** package. From the toolbar click the **New Java class** icon .
2. Ensure that **myIMSPhoneBookService** is the source folder and that **sample.ims** is the package name.
3. Type myPhoneBookIMSPProxyTestApp for the name of the new class.
4. Accept all other defaults and click **Finish**.
5. Replace the code in the editor with the following Java code:

```
package sample.ims;

import com.ibm.connector2.ims.ico.IMSDFSMessageException;


public class myPhoneBookIMSPProxyTestApp {

    public static void main(String[] args) {
        try
```

```

{
    INPUTMSG input = new INPUTMSG();
    input.setIn__ll((short) 59);
    input.setIn__zz((short) 0);
    input.setIn__trcd("IVTNO");
    input.setIn__cmd("DISPLAY");
    input.setIn__name1("LAST1");
    input.setIn__name2("");
    input.setIn__extn("");
    input.setIn__zip("");
    myPhoneBookIMSPProxy proxy = new myPhoneBookIMSPProxy();
    OUTPUTMSG output = proxy.runPhoneBook(input);
    System.out.println(
        "\nMessage: "
        + output.getOut__msg()
        + "\nName: "
        + output.getOut__name1()
        + " "
        + output.getOut__name2()
        + "\nExtension: "
        + output.getOut__extn()
        + "\nZipcode: "
        + output.getOut__zip());
    }
    catch (Exception e)
    {
        if (e instanceof org.apache.wsif.WSIFException)
        {
            Throwable ic4jEx = ((org.apache.wsif.WSIFException) e).
                getTargetException();
            if (ic4jEx instanceof IMSDFSMessageException)
            {
                System.out.println(
                    "\nIMS returned message: "
                    + ((IMSDFSMessageException) ic4jEx).getDFSMessage());
            }
            else
            {
                System.out.println("\nIMS Connector exception is: " + ic4jEx);
            }
        }
        else
        {
            System.out.println("\nCaught exception is: " + e);
        }
    }
}
}

```

6. Press **Ctrl-S** to save the changes and then close the editor. If you see compilation errors in the code, ensure that you used the correct name when you generated the proxy.
7. Select **myPhoneBookIMSPProxyTestApp.java** and expand the **Run** icon  on the toolbar by selecting the arrow beside it. From the pop-up menu, select **Run As > Java Application**.
8. The Java application should run without exceptions, and you should see the following message on the console:

```

Message: ENTRY WAS DISPLAYED
Name: LAST1      FIRST1
Extension: 8-111-1111
Zipcode: D01/R01

```

Successfully running the application ensures that your core application logic works correctly.

Note that this example assumes that your IMS system has the non-conversational COBOL version of the IMS INSTALL/IVP program installed and that the pre-loaded entries in the IVPDB2 database have not been modified during previous testing.

Now you can generate deploy code for the service that you have tested.

Step 5: Generating deploy code for the enterprise service

You use the Generate Deploy Code wizard to generate the EJB session bean. The EJB session bean handles the client request to the PhoneBook service. A PhoneBook request is a request to run the IMS transaction to add, delete, update, or display information that is stored in the IMS PhoneBook database. In addition to generating the EJB session bean, this wizard also generates deployed classes that allow the session bean to operate on an EJB server such as the WebSphere Application Server.

To generate the EJB session bean, complete the following steps:

1. In the **myIMSPhoneBookService** project, expand the **sample.ims** package and select the **myPhoneBookIMSService.wsdl** service file.
2. Right-click the file and select **Enterprise Services > Generate Deploy Code**. The Generate Deploy Code wizard opens.
3. In the Deployment page, the **Service file name**, **Service name**, and **Port name** default to values based on the service file that you selected in step 1. The default values are:
 - **Service file name:** myIMSPhoneBookService/sample/ims/myPhoneBookIMSService.wsdl
 - **Service name:** myPhoneBookIMSService
 - **Port name:** myPhoneBookIMSPort
4. Ensure that **Create a new port and binding** is selected.
5. Because you might want to create a service that uses SOAP, ensure that **SOAP** is selected from the **Inbound binding type** list. Even if you plan to deploy the service only as a SOAP service, you still need to create two inbound bindings (SOAP and EJB) to access the service. The EJB binding is required because the SOAP binding is built on top of the EJB binding.
6. Default project names (based on your service project name) are provided for you in the following fields:
 - **EAR project:** myIMSPhoneBookServiceEAR
 - **EJB project:** myIMSPhoneBookServiceEJB
 - **Web project:** myIMSPhoneBookServiceWeb

These are new projects and they will be created for you. Accept all other defaults and click **Next**.
7. In the Inbound Service Files page, accept the default values. Note that the name of the service is myPhoneBookService. Click **Next**.
8. In the EJB Inbound Service Files page, accept the default values. Click **Next**.
9. In the EJB Port page, accept the default value for the **JNDI name**. Click **Next**.
10. In the SOAP page, you can accept the default transport, style, action and encoding for the SOAP binding properties. These properties are all specific to the SOAP specification. Click **Next**.
11. In the SOAP Port page, you can specify the **SOAP port address** or you can accept the default. Click **Finish**.

In the Services view under **Deployable Services**, the myIMSPhoneBookServiceWeb project contains the service file myPhoneBookSOAPService.wsdl and the binding file myPhoneBookSOAPBinding.wsdl. The myIMSPhoneBookServiceWeb project also includes a SOAP deployment descriptor that tells the SOAP server about the service. The deployment descriptor contains servlet initialization and mapping information, as well as additional settings for running the Web module within an application server.

Also, under **Deployable Services**, the myIMSPhoneBookServiceEJB project contains all of the resources for EJB applications, including the session bean (MyPhoneBookService), the deployment descriptor, the remote interface, and the EJB home. The service project (myIMSPhoneBookService) contains the service definition. This project is zipped as a JAR file and placed in the enterprise application project (myIMSPhoneBookServiceEAR), which can be viewed in the J2EE view.

Step 6: Binding the resource reference

After generating the deploy code, you must also bind the resource reference to a resource. The resource reference includes the Java Naming and Directory Interface (JNDI) related information. With this information, a factory object can generate a connection when needed by the application at run time.

To bind the resource reference, complete the following steps:

1. Click the **J2EE Hierarchy** tab in the Business Integration perspective and expand **EJB Modules**.
2. Double-click **myIMSPhoneBookServiceEJB** to open the Deployment Descriptor Editor. Click the **References** tab.
3. Expand **MyPhoneBookService** and select the **ResourceRef** element.
4. Select **Application** in the Authentication field.
5. Under **WebSphere Bindings**, type myIMSTarget for the JNDI name.
6. Select **TRANSACTION_NONE** for the Isolation level.
7. Click the **Assembly Descriptor** tab. Under Container Transactions, click **Add**.
8. The Add Container Transaction wizard opens. Select the **MyPhoneBookService** check box and click **Next**.
9. In the Container transaction type field, select **NotSupported**. Expand **MyPhoneBookService** and select the **MyPhoneBookService** check box to select all of the methods in that bean. Click **Finish**.
10. Press **Ctrl-S** to save the changes and then close the editor.

Step 7: Configuring the server and deploying the EAR project

To run a service in this sample, you must deploy the session bean to a server. In this case, the server runs in the WebSphere Test environment. This means that the server must be configured and started. For this sample service, you need to create one server instance and server configuration. A server instance identifies the run-time environment that you want to use for testing your project resources. A server configuration contains information that is required to set up and publish to a server. After you configure the server, you will deploy the EAR project containing the service. Complete the following steps to configure the server and deploy the EAR project:

1. Creating and configuring the server instance.
2. Adding a connection factory to the server configuration.

3. Adding the EAR project to the server configuration.

Creating and configuring the server instance

To create a server instance and configure it, complete the following steps:

1. In the Business Integration perspective, click the **Server Configuration** tab to open the Server Configuration view. Right-click anywhere in the Server Configuration view. Select **New > Server and Server Configuration**. The Create a New Server and Server Configuration wizard opens.
2. Type `myIMSServicesServer` for the server name. (The default folder name is `Servers`.)
3. Expand **WebSphere version 5.1** and select **Integration Test Environment**. Click **Next**.
4. Select **Use default port numbers**. The server port number defaults to **9080**. The port identifies the location of the service. Click **Finish**. The new server instance appears in the Server Configuration view and in the Servers view.

You have just created an instance of the WebSphere Application Server that is emulated by the WebSphere Test Environment running on your local host on port 9080.

Adding a Connection Factory to the server configuration

You need to add an instance of the J2C connection factory to the server configuration and configure its properties. The connection factory provides connections to the EIS on demand. You specify all of the information needed by the resource adapter to connect to a particular instance of the EIS. For the IMS resource adapter, you must specify at least the `HostName`, `DataStoreName`, and `PortNumber` properties that determine which IMS to connect to. These values determine the IMS that will be accessed through all of the connections created by this instance of the connection factory. You also specify the JNDI lookup name under which the new connection factory instance will be available to components. The components can use this lookup name to quickly make a connection to the EIS.

To add a connection factory, complete the following steps:

1. Click the **Server Configuration** tab to see the Server Configuration view. Expand **Servers**.
2. Double-click the server configuration `myIMSServicesServer`. An editor opens.
3. Click the **J2C** tab. Click **Add** beside the J2C Resource Adapters table.
4. From the Resource Adapter Name list, select the resource adaptor named `ims222Connector`. Click **OK**.
5. In the J2C Resource Adapters table, select the IMS resource adapter, then click **Add** beside the J2C Connection Factories table. The application client will look up this connection factory instance using the JNDI interface. The application client will then use this connection factory instance to get a connection to the underlying IMS.
6. In the Create Connection Factory window, type the name `ims_cf`. Type the JNDI name `myIMSTarget`. Click **OK**.
7. In the Resource Properties table, type the property values appropriate for your environment. (See Connection properties for a description of these properties.) You might need to scroll down to see this table. For example:
 - In the **HostName** field, type `MYHOST.ABC.XYZ.COM`

- In the **PortNumber** field, type 9999
 - In the **DataStoreName** field, type MYDSTOR
8. Press **Ctrl-S** to save the changes and then close the editor.

Adding the EAR project to the server configuration


Next you need to add the EAR project (myIMSPhoneBookServiceEAR) to the server configuration that you created. To add the project, complete the following steps:

1. In the Server Configuration view under Servers, right-click **myIMSServicesServer**.
2. Select **Add and Remove Projects**.
3. The Add and Remove Projects page opens. Select **myIMSPhoneBookServiceEAR** and click **Add**. myIMSPhoneBookServiceEAR is the name of the enterprise application project that you created earlier.
4. Click **Finish**.

You have now successfully generated an enterprise service from an IMS transaction and deployed that service to the WebSphere test environment.

Part 2: Creating the client application

Now that you have generated the enterprise service, you will create a client application that can be used to invoke the enterprise service. Before you build the client application, launch the New Project wizard to create a service project for the client side of your service. To do this, complete the following steps:

1. From the toolbar, click the **Create a service project** icon . The New Project wizard opens.
2. Type myIMSPhoneBookServiceClient for the project name.
3. Select **Use default** to use the default location to store the new project.
4. Click **Finish** to create the project. You do not have to specify Java Build Path settings or dependent JARs in subsequent pages of the wizard because these are automatically set for you.

Next, you build a client application that uses either a SOAP proxy or an EJB proxy to access the service. The proxy provides a remote procedure call interface to the service. The client uses the proxy to call a remote method on the service as if the method were a local one. When the client makes the remote call, the proxy handles all of the communication details between the application and the service.

- To create a client application that uses an EJB proxy, go to Option 1: Creating a client application that uses an EJB proxy.
- To create a client application that uses a SOAP proxy, go to Option 2: Creating a client application that uses a SOAP proxy.

You do not need to create both types of proxies.

Option 1: Creating a client application that uses an EJB proxy

Complete the following steps to build a client application that uses an EJB proxy:

1. Generating an EJB proxy.
2. Creating the client application to access the service.
3. Testing the client application.

4. Using the Universal Test Client tool.

Step 1: Generating an EJB proxy

Earlier in this sample, you created the EAR file that includes the EJB inbound binding type. In fact, you generated the `myPhoneBookEJBService.wsdl` and binding file `myPhoneBookIMSEJBBinding.wsdl` in the `myIMSPhoneBookServiceEJB` project. Now you are going to generate an EJB proxy for the client. An EJB proxy hides the complexity of invoking the session bean from the client side. It functions much the same as an access bean, except that it uses the Web Services Invocation Framework (WSIF) to invoke the session bean.

To generate the EJB proxy, complete the following steps:

1. Click the **Services** tab of the Business Integration perspective. Expand **Deployable Services** > **myIMSPhoneBookServiceEJB** > **ejbModule** > **sample.ims** and select **myPhoneBookEJBService.wsdl**.
2. Right-click the file and select **Enterprise Services** > **Generate Service Proxy**. The Generate Service Proxy wizard opens.
3. In the **Proxy selection** page, select **Web Services Invocation Framework** for the type of proxy to generate. Click **Next**.
4. In the Service Proxy page, the fields contain default values that are based on the service file you selected. You need to change the **Source folder** field to place the proxy in the client project. Click **Browse** and select `myIMSPhoneBookServiceClient` or type `/myIMSPhoneBookServiceClient` for the **Source folder**.
5. Ensure that the package name is `sample.ims.client`.
6. Change the **Class name** to `myPhoneBookEJBProxy.java`. Click **Next**.
7. In the Service Proxy Style page, select **client stub** for the **Proxy style** and then select the **runPhoneBook** check box. Click **Finish**.
8. Resolve any errors that appear in the generated proxy by adding the service project to the list of required projects on the build path. To add the service project to the build path:
 - a. Right-click **myIMSPhoneBookServiceClient** and click **Properties**.
 - b. Select **Java Build Path** and click the **Projects** tab. Select the **myIMSPhoneBookService** check box and click **OK**.

The EJB proxy, `myPhoneBookEJBProxy`, is generated in the `myIMSPhoneBookServiceClient` project. The enterprise service WSDL files are also copied into this service project.

Now you can test the EJB proxy.

Step 2: Creating the client application to access the service

To bind the client to the service and to invoke the service, you must write a client Java application. Within your Java package, you will create a new class that includes a main method and contains code to test the EJB proxy. To create a new class that will test the EJB proxy, complete the following steps:

1. In the Services view, expand **myIMSPhoneBookServiceClient** and select the **sample.ims.client** package.
2. Click the **New Java class** icon  on the toolbar. The New Java Class wizard opens.


3. In the New Java Class wizard, ensure that the Source folder is **myIMSPhoneBookServiceClient** and the Package is **sample.ims.client**.
4. In the **Name** field, type **myPhoneBookEJBProxyTestApp** for the name of the class that you are creating.
5. Accept the other defaults and click **Finish**. The Java class, **myPhoneBookEJBProxyTestApp.java**, is created and is opened in the editor view.
6. Replace the code in the editor with the following Java code:

```
package sample.ims.client;
import sample.ims.*;
public class myPhoneBookEJBProxyTestApp
{
    public static void main(String[] args)
    {
        try
        {
            INPUTMSG input = new INPUTMSG();
            input.setIn__ll((short) 59);
            input.setIn__zz((short) 0);
            input.setIn__trcd("IVTNO");
            input.setIn__cmd("DISPLAY");
            input.setIn__name1("LAST2");
            input.setIn__name2("");
            input.setIn__extn("");
            input.setIn__zip("");
            myPhoneBookEJBProxy proxy = new myPhoneBookEJBProxy();
            OUTPUTMSG output = proxy.runPhoneBook(input);
            System.out.println(
                "\nMessage: "
                + output.getOut__msg()
                + "\nName: "
                + output.getOut__name1()
                + " "
                + output.getOut__name2()
                + "\nExtension: "
                + output.getOut__extn()
                + "\nZipcode: "
                + output.getOut__zip());
        }
        catch (Exception e)
        {
            System.out.println("\nCaught exception is: " + e);
        }
    }
}
```

7. Press **Ctrl-S** to save the file and then close the **myPhoneBookEJBProxyTestApp.java** file.

Step 3: Testing the client application

Next, you can run **myPhoneBookEJBProxyTestApp.java** by completing the following steps:

1. Click the **Servers** tab and check the status of the server instance. If the status for the server is stopped, then right-click the server instance and select **Start**. Wait until the server is started. The server is started when you see Started next to the server on the **Servers** tab.
2. In the **myIMSPhoneBookServiceClient** project, select the **myPhoneBookEJBProxyTestApp.java** class.
3. On the toolbar, select the arrow beside the **Run** icon  to expand it.

4. From the pop-up menu, select **Run**.
5. Under Launch Configurations, select **Java Application** and click **New**.
6. Click the **JRE** tab and select **WebSphere v5.1 EE JRE** from the JRE list.
7. Click the **Classpath** tab and deselect the **Use default class path** check box.
8. Click **Add Projects** and select **myIMSPhoneBookServiceEJB**. Click **OK**.
9. Click **Add External Jars** and browse to select *WS_installdir*\runtimes\ee_v51\lib, where *WS_installdir* is the directory where WebSphere Studio is installed. Within the lib directory, hold down the control key and select the following files:
 - naming.jar
 - namingclient.jar
 - namingserver.jar
- Click **Open**.
10. Click **Advanced** and select **Add External Folder**. Click **OK**. Browse to select *WS_installdir*\runtimes\ee_v51\properties, where *WS_installdir* is the directory where WebSphere Studio is installed. Click **OK**, and then click **Apply**.
11. Click **Run**.

The application should run without exceptions. In the Console view you will see the following message, which is returned by the service:

```
Message: ENTRY WAS DISPLAYED
Name: LAST2      FIRST2
Extension: 8-111-2222
Zipcode: D01/R02
```

Note that this example assumes that your IMS system has the non-conversational COBOL version of the IMS INSTALL/IVP program installed and that the pre-loaded entries in the IVPDB2 database have not been modified during previous testing.

Step 4: Using the Universal Test Client tool

Another option to test your client application is using WebSphere Studio's IBM Universal Test Client tool. The Universal Test Client (UTC) tool runs in the application server itself. It allows you to view and invoke methods, objects, and classes to test the EJB session bean, and you do not need to write any code for the test. In this test, you will only test the generated stateless session bean, myIMSPhoneBookServiceEJB.

To test the client application, complete the following steps:

1. In the **Package Explorer** view, select **myIMSPhoneBookServiceEJB**. Right-click and select **Run on Server**. The IBM Universal Test Client Homepage opens.
2. If the **Server Selection** page opens, select **Use an existing server** and select **myIMSServicesServer** in the list of servers.
3. Click **Finish**. The **IBM Universal Test Client Homepage** opens.
4. Click the **JNDI Explorer** link.
5. In the JNDI Explorer page, expand the folders **sample > ims** and click **myPhoneBookServiceHome(sample.ims.MyPhoneBookServiceHome stub)**. The References pane is displayed.

6. In the References pane, expand **EJB References > MyPhoneBookService > MyPhoneBookServiceHome** and click **MyPhoneBookService create ()**.
7. In the Parameters pane, click **Invoke**. This creates the **MyPhoneBookServiceHome** object.
8. Click **Work with Object**. This creates the **MyPhoneBookService 1** bean, which is displayed in the References pane.
9. In the References pane under EJB References, expand **MyPhoneBookService 1**. This shows a list of methods that can be executed in the bean. Select the **OUTPUTMSG runPhoneBook(INPUTMSG)** method.
10. In the Parameters pane, you can see the input field for the **OUTPUTMSG runPhoneBook()** method. This is a complex object that has to be created. In the Parameter Value table, click the **Expand** link beside **INPUTMSG** to expand the **INPUTMSG** parameter values.
11. Type the following values in the table:
 - **in_cmd**: DISPLAY
 - **in_ll**: 59
 - **in_name1**: LAST2
 - **in_trcd**: IVTNO
 - **in_zz**: 0 (the default is 0)
12. Click **Invoke**. You might need to scroll down to see the **Work with Object** button.
13. Click **Work with Object**. In the References pane under Object References, the objects are displayed. If you expand the objects, a list of methods that can be performed in the object are displayed.
14. By selecting different get methods, you can retrieve customer information. To do so, click one of the get methods from the list. Click **Invoke** to create the object. The results are displayed. Repeat this for every method you want to test.
15. When you have finished testing the methods, close the Universal Test Client.
Note: After using the Universal Test Client, you need to stop the server if you plan to use a test class to perform any additional testing. To stop the server, click the **Servers** tab. Right-click the server instance and select **Stop**. If you do not restart the server after using the Universal Test Client, you may encounter errors when you use a test class to test a proxy.

Option 2: Creating a client application that uses a SOAP proxy

Complete the following steps to build a client application that uses a SOAP proxy:

1. Generating the SOAP proxy.
2. Creating JSP files for the client application.
3. Testing the SOAP proxy.

Step 1: Generating the SOAP proxy

Earlier in this sample, you created the EAR file that includes the SOAP inbound binding type. In fact, you generated the **myPhoneBookSOAPService.wsdl** and binding file **myPhoneBookSOAPBinding.wsdl** in the **myIMSPhoneBookServiceWeb** project. Now you are going to generate a SOAP proxy for the client.

To generate the SOAP proxy, complete the following steps:

1. Click the **Services** tab of the Business Integration perspective. Expand **Deployable Services > myIMSPhoneBookServiceWeb > WebContent/wsdl > sample.ims**.
2. Right-click **myPhoneBookSOAPService.wsdl** and select **Enterprise Services > Generate Service Proxy**. The Generate Service Proxy wizard opens.
3. In the **Proxy Selection** page, select **Web Services Invocation Framework** for the type of proxy to generate. Click **Next**.
4. In the Service Proxy page, the fields contain default values that are based on the service file you selected. You need to change the **Source folder** field to place the proxy in the java source folder. Click **Browse** or type `/myIMSPhoneBookServiceWeb/JavaSource` for the **Source folder**.
5. Ensure that the package name is `sample.ims.client`.
6. Ensure that the **Generate helper classes** check box is selected.
7. Change the **Class name** to `myPhoneBookSOAPProxy.java`. Click **Next**.
8. In the Service Proxy page, select **client stub** for the **Proxy style** and then select the **runPhoneBook** check box.
9. Click **Finish**.

The SOAP proxy, `myPhoneBookSOAPProxy`, is generated in the `/myIMSPhoneBookServiceWeb/JavaSource` folder.

Now you can create JSP files for the client application.

Step 2: Creating JSP files for the client application

You write JSP files to bind the client to the service and to invoke the service. Within your Java package, you will create three new JSP files to test the SOAP proxy:

- A JSP file for input named `Controller.jsp`
- A JSP file to display results named `Display.jsp`
- A JSP file to display errors named `error.jsp`

To create the JSP files, complete the following steps:

1. Create a JSP file to accept user input.
 - a. In the Package explorer, right-click **myIMSPhoneBookServiceWeb** and select **New > Other**.
 - b. Select **Web** and then select **JSP File**. Click **Next**. The New JSP File wizard opens.
 - c. Ensure that the folder is `/myIMSPhoneBookServiceWeb/WebContent` and type `Controller.jsp` for the File Name. Click **Finish**. The `controller.jsp` file opens in the editor.
 - d. Click the **Source** tab and replace the code in the editor with the following code for the JSP:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN">
<HTML>
<HEAD>
<%@ page
language="java"
contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"
%>
<META http-equiv="Content-Type" content="text/html;
```

```

charset=ISO-8859-1">
<META name="GENERATOR" content="IBM WebSphere Studio">
<META http-equiv="Content-Style-Type" content="text/css">

<TITLE>Controller.jsp</TITLE>
</HEAD>
<BODY>
<FORM METHOD="POST" ACTION="Display.jsp"><BR>
<H1>Query the PhoneBook application!</H1>
<P>To display a phone book entry, type DISPLAY in the
command field and type a last name.
To add or delete an entry, type ADD or DELETE in the command field
and
fill in other fields with the required information. </P>
<TABLE>
  <TR><TD>Command: </TD>
  <TD><INPUT TYPE="TEXT" NAME="Cmd" VALUE="" SIZE="10"
MAXLENGTH="50"></TD>
</TR>
  <TR><TD>Last name: </TD>
  <TD><INPUT TYPE="TEXT" NAME="Name1" VALUE="" SIZE="10"
MAXLENGTH="50"></TD>
</TR>
  <TR><TD>First name: </TD>
  <TD><INPUT TYPE="TEXT" NAME="Name2" VALUE="" SIZE="10"
MAXLENGTH="50"></TD>
</TR>
  <TR><TD>Extension: </TD><TD>
  <TD><INPUT TYPE="TEXT" NAME="Extn" VALUE="" SIZE="10"
MAXLENGTH="50"></TD>
</TR>
  <TR><TD>Zip code: </TD>
  <TD><INPUT TYPE="TEXT" NAME="Zip" VALUE="" SIZE="10"
MAXLENGTH="50">
</TD></TR>
</TABLE><BR>
<INPUT TYPE="SUBMIT" NAME="Submit">
</FORM>
</BODY>
</HTML>

```

- e. Press **Ctrl-S** to save the file and then close the editor.

Note: You can ignore the broken link message because this problem will be resolved when you build the next two JSP files.

2. Create a JSP file to display the output.

- a. In the Package explorer, right-click **myIMSPhoneBookServiceWeb** and select **New > Other**.
- b. Select **Web** and then select **JSP File**. Click **Next**. The New JSP File wizard opens.
- c. Ensure that the folder is **/myIMSPhoneBookServiceWeb/WebContent** and type **Display.jsp** for the File Name. Click **Finish**. The display.jsp file opens in the editor.
- d. Click the **Source** tab and replace the code in the editor with the following code for the JSP:

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN">
<HTML>
<HEAD>
<%@ page
language="java"
import="javax.naming.*, sample.ims.*, sample.ims.client.*"
contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"

```

```

errorPage="error.jsp"
%>

<META HTTP-EQUIV="PRAGMA" CONTENT="NO-CACHE">
<TITLE>Display.jsp</TITLE>
</HEAD>
<BODY>
<%
response.setHeader("Cache-Control","no_store"); //HTTP 1.1
response.setHeader("Cache-Control","must-revalidate"); //HTTP 1.1
response.setHeader("Pragma","no-cache"); //HTTP 1.0
response.setDateHeader ("Expires", 0); //prevents caching at the
proxy server
response.setHeader("Cache-Control","no-store"); //HTTP 1.1

    INPUTMSG input = new INPUTMSG();
    input.setIn__ll((short) 59);
    input.setIn__zz((short) 0);
    input.setIn__trcd("IVTNO");
    input.setIn__cmd(request.getParameter("Cmd"));
    input.setIn__name1(request.getParameter("Name1"));
    input.setIn__name2(request.getParameter("Name2"));
    input.setIn__extn(request.getParameter("Extn"));
    input.setIn__zip(request.getParameter("Zip"));

    myPhoneBookSOAPProxy proxy = new myPhoneBookSOAPProxy();
    OUTPUTMSG output = proxy.runPhoneBook(input);
    %>

    <H1>Query Results</H1><hr WIDTH="50%"
ALIGN=LEFT>

<TABLE BORDER=0 CELLSPACING=0 CELLPADDING=0 WIDTH="50%">
<TR ALIGN="left" VALIGN="middle">
<TH>Field</TH>
<TH>Value</TH></TR>
<TR ALIGN="left" VALIGN="middle">
<TD>Last name</TD>
<TD>
<%= output.getOut__name1() %>
</TD></TR>
<TR ALIGN="left" VALIGN="middle">
<TD>First name</TD>
<TD>
<%= output.getOut__name2() %>
</TD></TR>
<TR ALIGN="left" VALIGN="middle">
<TD>Extension</TD>
<TD>
<%= output.getOut__extn() %>
</TD></TR>
<TR ALIGN="left" VALIGN="middle">
<TD>Zip code</TD>
<TD>
<%= output.getOut__zip() %>
</TD></TR>
</TABLE>
<hr WIDTH="50%" ALIGN="LEFT">
<P>Status: <%= output.getOut__msg() %></P>

</BODY>

```

```

<HEAD>
  <META HTTP-EQUIV="PRAGMA" CONTENT="NO-CACHE">
</HEAD>
</HTML>

```

- e. Press **Ctrl-S** to save the file and then close the editor.
3. Create a JSP file to display any errors encountered during the query.
 - a. In the Package explorer, right-click **myIMSPhoneBookServiceWeb** and select **New > Other**.
 - b. Select **Web** and then select **JSP File**. Click **Next**. The New JSP File wizard opens.
 - c. Ensure that the folder is `/myIMSPhoneBookServiceWeb/WebContent` and type `error.jsp` for the File Name. Click **Finish**. The `error.jsp` file opens in the editor.
 - d. Click the **Source** tab and replace the code in the editor with the following code for the JSP:

```

<%@ page isErrorPage="true" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN">
<HTML>
<HEAD>
  <title>Error Page</title>
</HEAD>
<BODY text="#000000" bgcolor="#FFFFFF" link="#0000FF"
vlink="#800080" alink="#FF00FF">

<CENTER>
<h2>Error</h2></CENTER>

<P>Application <B>PhoneBook</B> reported the
following error:
<P>
<FONT color="#3333FF"><%=exception.toString()
%></FONT>

<P>This problem occurred in the following place:
<P>
<PRE>
<%= exception.printStackTrace(new java.io.PrintWriter(out));
%>
</PRE>
</BODY>
</HTML>

```

- e. Press **Ctrl-S** to save the file and then close the editor.
Now you can use the JSP files to test the SOAP proxy.

Step 3: Testing the SOAP proxy

Next, you test the SOAP proxy by using the input JSP to send requests to the PhoneBook application. To test the SOAP proxy, complete the following steps:

1. In the **Package** explorer, expand **myIMSPhoneBookServiceWeb > Web Content**.
2. Right-click **Controller.jsp** and click **Run on server**.
3. If the Server Selection wizard displays, select **Use an existing server** and select the server that you configured for this sample. Click **Finish**.

The server starts and launches the JSP. You should see the following in the display:

Query the PhoneBook application!

To display a phone book entry, type **DISPLAY** in the command field and type a last name. To add or delete an entry, type **ADD** or **DELETE** in the command field and fill in other fields with the required information.

Command:
Last name:
First name:
Extension:
Zip code:

Submit Query

4. Issue a query to display a phone book entry. Type the following values in the fields on the input JSP:

Field	Value
Command	Display
Last name	LAST1

5. Click **Submit Query**. You should see the following results of the query:

Query Results

Field	Value
Last name	LAST1
First name	FIRST1
Extension	8-111-2222
Zip code	D02/R02

Status: ENTRY WAS DISPLAYED

6. Click the back button in the browser to return to the input JSP.
7. Issue a query to add a phone book entry. Type the following values in the fields on the input JSP:

Field	Value
Command	Add
Last name	Doe
First name	Jane
Extension	5-5555
Zip code	55555

8. Click **Submit Query**. You should see the following results of the query:

Query Results

Field	Value
Last name	DOE
First name	JANE
Extension	5-5555
Zip code	55555

Status: ENTRY WAS ADDED

- Click the back button in the browser to return to the input JSP.
- Issue a query to delete the phone book entry that you just created. Type the following values in the fields on the input JSP:

Field	Value
Command	Delete
Last name	Doe

- Click **Submit Query**. You should see the following results of the query:

Query Results

Field	Value
Last name	DOE
First name	JANE
Extension	5-5555
Zip code	55555

Status: ENTRY WAS DELETED

What's next?

Now that you the have successfully completed this sample, what can you do next?

- You can deploy the service to a production version of WebSphere Application Server by following the tasks described in [Deploying an IMS enterprise service to a production server](#).
- You might be interested in reading the following documentation, which has step-by-step instructions on:
 - Bottom-up development: [Creating an enterprise service](#)
 - Top-down development: [Creating an enterprise service](#)
- For a general understanding of the WebSphere Studio product and the programming and development model for enterprise services, read the [Product Overview](#) (also available in PDF).
- You can read the other enterprise services scenarios and samples available in the product:
 - Sample: [Deploying an IMS enterprise service to a production server](#).
 - Sample: [Building a service that submits commands to IMS](#).
 - Sample: [Running an enterprise service for an IMS transaction](#).
 - Sample: [Building container-managed and component-managed transactional EJBs to run IMS transactions](#).

- Auction (Business Integration) scenario
- eMerged Financial Portal scenario
- Travel agency scenario
- Creating an enterprise service from an EJB
- Creating a Java skeleton service
- Other enterprise services scenarios and samples

Sample: Deploying an IMS enterprise service to a production server

Objectives

In this sample, you deploy an IMS service to a WebSphere Application Server Enterprise Edition production server. Before performing the steps in this sample, you must complete the steps described in Sample: Creating an enterprise service for an IMS transaction because you will deploy the same service in this sample.

Time required

Allow 60 minutes. This will give you enough time to deploy the enterprise service to a production server.

Before you begin

This sample assumes that WebSphere Application Server Enterprise Edition Version 5.1 is installed on your machine and that you are familiar with using the product. It also assumes that you have installed and configured the IMS resource adapter (also known as IMS Connector for Java) on the WebSphere Application Server. For more information about installing and configuring the IMS resource adapter, see the "How To" file that ships with the IMS Connector for Java run-time code.

Important: If you want to run your application on a remote server, see **Application Server Toolkit** in the WebSphere Application Server Version 6.0 Information Center. The unit test environment in WebSphere Studio does not support running remote WebSphere Application Servers for z/OS.

Description

To deploy the IMS service, complete the following tasks:

1. Within the WebSphere Studio Integration Edition, package the service for the production server:
 - Edit the module dependencies.
 - Export the EAR file.
2. Deploy the service to a production server:
 - Start the Administrative console.
 - Add a J2C connection factory.
 - Install the EAR file.
3. Test the service on a production server.

Part 1: Packaging the service for the production server

To package the service for the production server, complete the following steps:

1. Edit the module dependencies.

2. Export the EAR file.

Step 1: Editing the module dependencies

Because you are going to use a SOAP proxy to access the service on the production server, you must edit the Web module dependencies.

Note: If your production version of WebSphere Application Server Enterprise Edition has the module visibility set to "Application" rather than to the default value of "Module", then you do not need to complete the steps in this section.

To edit the Web module dependencies, complete the following steps:

1. Within WebSphere Studio, click the **J2EE Hierarchy** tab and expand **Web Modules**.
2. Right-click **myIMSPhoneBookServiceWeb** and select **Open With > Jar Dependency Editor**.
3. In the Jar Dependencies page, select the check boxes for **myIMSPhoneBookService.jar**, which contains the service interface definition file, and **myIMSPhoneBookServiceEJB.jar**, which contains the EJB project named myIMSPhoneBookServiceEJB.
4. Press **Ctrl-S** to save your changes and then close the editor.

Step 2: Exporting the EAR file

In this step, you export the EAR file that contains the service. To export the EAR file, complete the following steps:

1. Select **File > Export**. The Export wizard opens.
2. Select **EAR file** and click **Next**.
3. Select **myIMSPhoneBookServiceEAR** from the **What resources do you want to export?** list.
4. Click **Browse** beside the **Where do you want to export resources to?** field and select the folder where you want to export the resources: for example, C:\PhoneBook.ear. Click **Open**.
5. Click **Finish**.

Part 2: Deploying the service to a production server

To deploy the service to the production server, complete the following tasks:

1. Start the Administrative console.
2. Add a J2C connection factory.
3. Install the EAR file.
4. Start the application.

Note: You must install the IMS resource adapter (also known as IMS Connector for Java) on the production server before deploying the service.

Step 1: Starting the Administrative console

To start the Administrative console, complete the following steps:

1. From the Windows desktop, click **Start > Programs > IBM WebSphere > Application Server v5.0 > Start the Server**. You must start the server before you start the Administrative Console.

2. When the server is started, click **Start > Programs > IBM WebSphere > Application Server v5.0 > Administrative Console**.
3. In the WebSphere Administrative Console, enter a valid user ID and password, if necessary.

Step 2: Adding a J2C connection factory

Before adding the J2C connection factory, ensure that the IMS resource adapter has been installed and configured with the correct connection properties for your environment. To view installed resource adapters, complete the following steps:

1. In the left frame of the Administrative Console, expand **Resources** and click **Resource Adapters**.
2. In the Resource Adapters page, look for the name of the IMS resource adapter, for example IMS Connector for Java. The name of the resource adapter is defined when you deploy the adapter. If the IMS resource adapter is not deployed, you need to deploy it before completing the steps described in this sample. For more information about deploying the IMS resource adapter, see the "How To" file that ships with the IMS Connector for Java run-time code.
3. If the resource adapter appears in the list, click the name of the resource adapter to view the properties for the adapter.

After verifying that the IMS resource adapter is correctly installed, complete the following steps to add a J2C connection factory:

1. In the left frame of the Administrative Console, expand **Resources** and click **Resource Adapters**. Click the link for the installed resource adapter. For example, IMS Connector for Java.
2. Under Additional Properties at the bottom of the page, click **J2C Connection Factories**. You might need to scroll down to see this link.
3. Click **New** to add a new J2C connection factory for this resource adapter.
4. In the **Name** field, type a display name for the IMS resource adapter, for example, type `ims_cf`.
5. In the JNDI name field, type `myIMSTarget`. Click **Apply** to apply your changes. **Note:** Depending on your security configuration, you might need to select one of the previously defined authentication aliases. To view or create J2C authentication data entries, expand **Security > JAAS Configuration** and click **J2C Authentication Data**.
6. Next, under Additional Properties at the bottom of the page, click **Custom Properties** and specify the connection properties for the connection factory. See Connection properties for a description of these properties. For example:
 - In the **HostName** field, type `MYHOST.ABC.XYZ.COM`
 - In the **DataStoreName** field, type `MYDSTOR`
 - In the **PortNumber** field, type `9999`Click the link for each property that you need to set and type the value appropriate for your environment. After setting the value for each property, click **OK**.
7. When you are finished typing values for connection properties, click **Save** in the Message(s) box at the top of the Custom Properties page. Click **Save** again to apply your changes to the master configuration.

Step 3: Installing the EAR file

To install the EAR file, complete the following steps:

1. From the Administrative console, expand **Applications** and click **Install New Application**.
2. Click **Browse** and select the EAR file that you created in the previous step: for example, C:\PhoneBook.ear. Click **Open** and then click **Next**.
3. In the Preparing for the application installation page, accept the defaults and click **Next**.
4. For Step 1, you can accept all the defaults and click **Next**. (You may change any of the defaults.)
5. For Step 2, accept the defaults and click **Next**.
6. For Step 3, accept the defaults. (The JNDI name must match the JNDI name of the Connection Factory you will be using.) Select the checkbox next to myIMSPhoneBookServiceEJB and click **Apply**. Click **Next**.
7. For Step 4, accept the defaults and click **Next**.
8. For Step 5, accept the defaults and click **Next**.
9. For Step 6, accept the defaults and click **Next**.
10. For Step 7, click **Finish**.
11. Click **Save to Master Configuration** and then click **Save** again to save your changes.

Now you can start the application server and test the deployed service.

Step 4: Starting the application

To start the application, complete the following steps:

1. In the left frame of the WebSphere Application Server Administrative console, expand **Applications** and click **Enterprise Applications**.
2. Select the check box beside **myIMSPhoneBookServiceEAR** and click **Start**.

Part 3: Testing the service on a production server

To test the service on a production server, you use a SOAP proxy, as described in this section.

Here, you reuse the SOAP proxy you created in Sample: Creating an enterprise service for an IMS transaction. To test the service on a production server by using a SOAP proxy, complete the following steps:

1. Launch a browser and type the following URL:
`http://host_name:9080/myIMSPhoneBookServiceWeb/Controller.jsp`. You should see the following in the browser window:

Query the PhoneBook application!

To display a phone book entry, type **DISPLAY** in the command field and type a last name. To add or delete an entry, type **ADD** or **DELETE** in the command field and fill in other fields with the required information.

Command:
Last name:
First name:
Extension:
Zip code:

Submit Query

- Issue a query to display a phone book entry. Type the following values in the fields on the input JSP:

Field	Value
Command	Display
Last name	LAST1

- Click **Submit Query**. You should see the following results of the query:

Query Results

Field	Value
Last name	LAST1
First name	FIRST1
Extension	8-111-2222
Zip code	D02/R02

Status: ENTRY WAS DISPLAYED

- Click the back button in the browser to return to the input JSP.
- Issue a query to add a phone book entry. Type the following values in the fields on the input JSP:

Field	Value
Command	Add
Last name	Doe
First name	Jane
Extension	5-5555
Zip code	55555

- Click **Submit Query**. You should see the following results of the query:

Query Results

Field	Value
Last name	DOE
First name	JANE
Extension	5-5555
Zip code	55555

Status: ENTRY WAS ADDED

- Click the back button in the browser to return to the input JSP.
- Issue a query to delete the phone book entry that you just created. Type the following values in the fields on the input JSP:

Field	Value
Command	Delete
Last name	Doe

- Click **Submit Query**. You should see the following results of the query:

Query Results

Field	Value
Last name	DOE
First name	JANE
Extension	5-5555
Zip code	55555

Status: ENTRY WAS DELETED

What's next?

Congratulations! You have successfully deployed and tested your service. If you are interested in learning more about building enterprise services, read other enterprise services scenarios and samples available in the product.

Sample: Running an enterprise service for an IMS transaction

Objectives

There are two ways to become familiar with the business integration tools used to build a service that runs an IMS transaction. One way is to use the Samples Wizard to install the ready-to-run IMS Connector for Java PhoneBook sample in the WebSphere Studio Integration Edition workspace. This allows you to quickly execute and become familiar with an IMS service implementation. This document describes how to use the ready-to-run PhoneBook sample.

Another way to become familiar with the business integration tools is to follow the detailed steps in the Sample: Creating an enterprise service for an IMS transaction for developing and running the PhoneBook sample IMS service. This approach

provides an in-depth look at how the PhoneBook sample IMS service can be created from the COBOL data structures, then deployed and executed using the Integration Edition tools.

To create the PhoneBook sample IMS service yourself, follow the instructions in *Sample: Creating an enterprise service for an IMS transaction*. Note that the names used in the ready-to-run PhoneBook IMS service and those used in the documentation describing how to develop and run your own PhoneBook IMS service, differ slightly so that you can have both projects in your workspace without any naming conflicts.

Time required

Allow 60 minutes. This will give you enough time to run your enterprise service for an IMS transaction. The 60 minutes does not include time for completing the actions specified in the **Before you begin** section.

Before you begin

In this sample, your application uses the IMS resource adapter (also called IMS Connector for Java) to interact with IMS through IMS Connect. The sample service runs the PhoneBook IMS transaction on an IMS system you specify. Because the PhoneBook IMS transaction is one of the IMS Installation Verification Programs, it is probably already installed on your IMS test system. Before running this sample:

- Contact your IMS system programmer to verify that the IVTNO transaction which is part of the IMS Installation Verification Program is installed and working on your target IMS system.
- Ensure that your environment meets the prerequisites for using the IMS resource adapter. See *Prerequisites for using the IMS resource adapter*.
- Import the IMS resource adapter, **ims.rar** file. To import the RAR file, see *Importing a resource adapter*.

You may find it convenient to stay in the Business Integration perspective while installing and running the sample. If you would like to do this and have not already done so, follow these steps:

1. Select **Window > Preferences**.
2. In the left-hand column, expand **Workbench** and select **Perspectives**.
3. In the Perspective pane, select **Never switch** under **Switch to associated perspective when creating a new project**.
4. Click **OK**.

Description

The ready-to-run PhoneBook sample is installed in the WebSphere Studio Application Developer Integration Edition workspace. To run the installed PhoneBook sample, you can use the Samples Wizard to install the PhoneBook sample and then run it as a stand-alone Java application or as a deployed service invoked on an application server through a SOAP proxy by following the detailed steps in this sample.

Installing the ready-to-run PhoneBook sample

1. In the workbench, select **File > New > Other**.
2. In the left pane, expand **Examples > Business Integration > Services**.

3. In the right pane, select **Service for IMS transaction (COBOL)** and then click **Next**.
4. Accept all the defaults in the Service for IMS transaction (COBOL) window. (The default **EAR Import Project** name is IMSPhoneBookEAR.) Click **Finish**.

Configuring the ready-to-run PhoneBook sample to run as a stand-alone Java application

After installing the PhoneBook sample IMS service, you must configure the PhoneBookIMSService.wsdl file for your target IMS system.

1. In the Business Integration perspective Services view, under **IMSPhoneBookService > sample.ims**, double-click the **PhoneBookIMSService.wsdl** file. This opens the file in the Graph tab in the WSDL editor.
2. In the Services container, expand **PhoneBookIMSService > PhoneBookIMSPort** and select **ims:address**.
3. In the **ims:address** section, enter the appropriate property values required for your IMS test environment in the table. At a minimum, you must enter the property values for the **dataStoreName**, **hostName** and **portNumber** properties.
Note: After entering the value for any of the properties, you must click somewhere else in the frame (for example, on the value field of another property) to move the cursor out of the field in which you have made a change. This must be done so the change can be recognized. If you do not do so, the new value for the last change you made will not be saved in the next step. See **Connection Properties** for a description of these properties.
4. Press **Ctrl-S** to save your changes and close the WSDL editor by clicking the **X** next to PhoneBookIMSService.wsdl.

Running the ready-to-run PhoneBook sample as a stand-alone Java application

After installing and configuring the PhoneBook sample IMS service, you can run the test Java application that is provided as part of the PhoneBook sample. You do not have to create a new server instance and configuration prior to running the PhoneBook sample as a Java application.

1. In the Services view, go to the **IMSPhoneBookService** project and expand the **sample.ims** package.
2. Select the **PhoneBookIMSProxyTestApp.java** file in the **Packages** view.
3. Expand the **Run** pulldown menu by selecting the arrow beside the **Run** icon on the toolbar.
4. From the pulldown menu, select **Run as > Java Application**.
5. The stand-alone test java application runs and in the console, you will see the following message:

```
Message: ENTRY WAS DISPLAYED
Name: LAST1      FIRST1
Extension: 8-111-1111
Zipcode: D01/R01
```

Note: This sample assumes that your target IMS system has the non-conversational COBOL version of the IMS INSTALL/IVP program installed and that the pre-loaded entries in the IVPDB2 database have not been modified during previous testing.

Running the ready-to-run PhoneBook sample on a server

You can also make this service available by deploying the service to WebSphere Application Server as a SOAP service. In this case, the server will run within WebSphere Studio. Following are the steps you need to complete in order to set up an application server in the WebSphere Test Environment in WebSphere Studio Application Developer and then deploy the service to that server:

- Configure a server
- Add a Connection Factory to the server configuration
- Add an EAR project to the server configuration and start or restart the server
- Execute the IMS PhoneBook sample IMS service using a SOAP proxy

Configuring a server

Prior to executing the SOAP program provided by the ready-to-run PhoneBook sample, the server must be configured. Complete the steps in Adding a server instance and server configuration to do so.

Adding a Connection Factory to the server configuration

To run the ready-to-run PhoneBook sample on a server, you need to add an instance of the JCA connection factory to the server configuration and configure its properties. To do so, complete the steps in Adding a Connection Factory to the server configuration in Sample: Creating an enterprise service for an IMS transaction.

Adding an EAR project to the server configuration and starting or restarting the server

Next you need to add the enterprise application project (IMSPhoneBookServiceEAR) to the server configuration that you created earlier, which is required to correctly start the server and test your service. To add the project and then start or restart the server, follow these steps:

1. In the Server Configuration view, expand **Server Configurations** and right-click **myIMSServicesServer**.
2. Select **Add and remove projects**. Ensure you have IMSPhoneBookServiceEAR installed.
3. In the Servers view, check the status of **myIMSServicesServer**. If the server is in "Stopped" status, you will need to start it. However, if the server is in "Started" status, you will need to restart it in order to pick up the binding information and the IMSServicesEAR project. For information about starting or restarting the server, see Starting or restarting the server.

Executing the IMS PhoneBook sample IMS service using a SOAP proxy

To execute the ready-made IMS PhoneBook sample IMS service using a SOAP proxy, execute the following instructions:

1. Start the server if it is not already started following the instructions in Starting or restarting the server.
2. In the Package Explorer view, expand **IMSPhoneBookServiceWeb > Web Content**.
3. Right-click **Controller.jsp** and click **Run on server**.

- If the Server Selection wizard displays, select **Use an existing server** and select the server that you configured for this sample. Click **Finish**.
The server starts and launches the JSP. You should see the following in the display:

Query the PhoneBook application!

To display a phone book entry, type DISPLAY in the command field and type a last name. To add or delete an entry, type ADD or DELETE in the command field and fill in other fields with the required information.

Command:

Last name:

First name:

Extension:

Zip code:

- Issue a query to display a phone book entry. Type the following values in the fields on the input JSP:

Field	Value
Command	Display
Last name	LAST1

- Click **Submit Query**. You should see the following results of the query:

Query Results

Field	Value
Last name	LAST1
First name	FIRST1
Extension	8-111-2222
Zip code	D02/R02

Status: ENTRY WAS DISPLAYED

- Click the back button in the browser to return to the input JSP.
- Issue a query to add a phone book entry. Type the following values in the fields on the input JSP:

Field	Value
Command	Add
Last name	Doe
First name	Jane
Extension	5-5555
Zip code	55555

- Click **Submit Query**. You should see the following results of the query:

Query Results

Field	Value
Last name	DOE
First name	JANE
Extension	5-5555
Zip code	55555

Status: ENTRY WAS ADDED

10. Click the back button in the browser to return to the input JSP.
11. Issue a query to delete the phone book entry that you just created. Type the following values in the fields on the input JSP:

Field	Value
Command	Delete
Last name	Doe

12. Click **Submit Query**. You should see the following results of the query:

Query Results

Field	Value
Last name	DOE
First name	JANE
Extension	5-5555
Zip code	55555

Status: ENTRY WAS DELETED

What's next?

If you have successfully executed these ready-to-run sample applications and would like to take the next step of creating a similar set of applications on your own, see [Creating an enterprise service for an IMS transaction](#) for instructions on how to do so. It is also possible to create an IMS service that runs an IMS command. To create an IMS service that runs an IMS command rather than an IMS transaction, click on this link: [Building a service that submits commands to IMS](#). For instructions on deploying an IMS service to a production server, click on this link: [Deploying an IMS Enterprise service to a production server](#).

Good luck and have fun!

Sample: Building a service that submits commands to IMS

The IMS resource adapter (also known as IMS Connector for Java) included with WebSphere Studio is primarily intended for use in running transactions on a host IMS system. However, it can also be used by WebSphere Studio applications to submit certain IMS commands. The IMS resource adapter uses the host product, IMS Connect, to access IMS. IMS Connect uses the Cross-system Coupling Facility (XCF) to access IMS through OTMA. IMS allows only certain commands to be submitted through the IMS OTMA interface. Since the IMS resource adapter

accesses IMS through OTMA, these are the only commands that can be submitted to IMS by an application that uses the IMS resource adapter. For a list of these commands, see the IMS Version 6 publication *Operator's Reference*, section *Commands Supported from OTMA*.

The output of an IMS command is a message consisting of one or more segments of data. The output of some IMS commands is a "DFS" message. For example, the output of most /START commands is usually the message "DFS058I START COMMAND COMPLETED". Other IMS commands do not return "DFS" messages. For example, /DISPLAY commands return multiple segments of data representing lines of display information. In order to treat both types of output the same, you must set the property **imsRequestType** of class **com.ibm.connector2.ims.ico.IMSInteractionSpec** to **2**. This value indicates to the IMS resource adapter that the interaction is an IMS command and that "DFS" messages should be treated as normal output and not as Java exceptions.

This sample illustrates how you can create a standalone Java application to invoke a service that submits a command to a host IMS system. Other types of Java applications can also be used. A detailed description of how to create an IMS service and invoke the service using various types of Java applications can be found in Sample: Creating an enterprise service for an IMS transaction.

Before running the sample, ensure that your COBOL copybook contains valid COBOL definitions for the input and output messages:

- The input definition should include an LL field, a ZZ field, and a text field large enough for the command(s) you wish to submit. For example:
 - ```
01 INMSG.

 02 INLL PICTURE S9(3) COMP.

 02 INZZ PICTURE S9(3) COMP.

 02 INCMD PICTURE X(30).
```
- The output definition will be used only as a mechanism for retrieving the byte array containing the command output, so it only requires an LL field, a ZZ field, and a text field of non-zero size. For example:
  - ```
01 OUTMSG.  
  
    02 OUTLL         PICTURE S9(3) COMP VALUE +0.  
  
    02 OUTZZ         PICTURE S9(3) COMP VALUE +0.  
  
    02 DATA         PICTURE X(30) VALUE SPACES.
```

Much of the code described below can be found in:

`WS_installdir\wstools\eclipse\plugins\com.ibm.etools.ctc.samples.ims_5.1.0\sampleparts`

where `WS_installdir` is the WebSphere Studio installation directory. If you want to use this code you should follow all of the steps below, using the suggested names.

To build the sample Java application that submits commands to IMS, complete the following tasks:

Step 1. Creating a service project

The service project stores all of the files for your project, including imported source files and files generated by wizards. To create a service project, complete the following steps:

1. Ensure you are in the Business Integration perspective.
2. Click the **Create a service project** icon in the toolbar to create a service project. The Service Project wizard opens.
3. Type `IMSCmdSample` for the name of the project. Under **Project contents** accept the default and click **Finish**.

Step 2. Importing a COBOL file

In this step, you import the COBOL copybook file that is needed to create your service definition. The COBOL file defines the structure of the input and output messages.

Before importing the COBOL files into the workbench, create a Java package to hold the file:

1. Create a Java package for the file you are about to import. Select the project you just created (`IMSCmdSample`) and click the **New Java package** icon.
2. In the Java Package page of the wizard, ensure that `IMSCmdSample` is the name of the folder.
3. Type `sample.ims.ims cmd` for the name of the package. Click **Finish**.

Next, you need to import the COBOL file into the `sample.ims.ims cmd` package.

1. Expand the `IMSCmdSample` project, right-click the package you just created (`sample.ims.ims cmd`), and select **Import** to open the Import wizard.
2. Select **File system** to import the resources from the local file system. Click **Next**.
3. Click **Browse** to locate and select the following directory:

```
WS_install_dir\wstools\eclipse\plugins\com.ibm.etools.ctc.samples.ims_5.1.0\sampleparts
```

Click **OK**.

4. In the File system page of the Import wizard, select the **IMS** folder without selecting the check box.
5. Select the `IMSCmd.ccp` check box to import the COBOL copybook.
6. Ensure that `IMSCmdSample/sample/ims/ims cmd` is the name of the destination folder for the imported resource. Click **Finish** to import the file and close the wizard.
7. Look at the Tasks view to see if the import resulted in any errors.
8. Verify that the `sample.ims.ims cmd` package in the Package Explorer view contains the imported file.

Step 3. Creating a service definition

The service definition is described in Web Services Description Language (WSDL), which is standard for describing networked, XML-based services. WSDL provides a simple way to describe the basic format of system requests regardless of the underlying run-time implementation. A WSDL document describes where the service is deployed and what operations the service provides. WebSphere Studio

tools for building enterprise services use WSDL as the model for describing any kind of service. To create the service definition, complete the following steps:

1. Select **File > New > Other**.
2. In the Select window, select **Business Integration** and choose **Services built from**. Click **Next**.
3. In the Create Service page, select **IMS** and click **Next**.
4. In the Connection Properties page, type the property values appropriate for your environment. See Connection properties for a description of these properties. For example:
 - In the **Host name** field, type MYHOST.ABC.XYZ.COM
 - In the **Port number** field, type 9999
 - In the **Data store name** field, type MYDSTOR
5. Click **Next**. **Note:** Because the connection properties are not encrypted, you should remove at minimum the User name and password from the port definition after you have completed testing.
6. In the Service Binding page, ensure that the following values are set:
 - **Source folder** name: /IMSCmdSample.
 - **Package** name: sample.ims.imscmd.
 - **Target namespace:** http://imscmd.ims.sample/.

In the **Interface file name** field, type IMSCmd. The remaining fields are automatically filled. Ensure that the Port type name is IMSCmd.
7. Click **Finish** to accept all other default names. Click **OK** to proceed to the page where you will create the binding operations. A WSDL editor opens with the overview of the IMSCmdIMSBinding.wsdl file.
8. In the Bindings container of the Graph view, right-click **IMSCmdIMSBinding** and select **Generate Binding Content**.
9. The Specify Binding Details page opens. In the **Protocol** field, select **IMS**.
10. Click **Add** next to the **Add binding operations** field.
11. In the Operation Binding page, you create new input and output messages based on the existing input and output operations. In the **Operation name** field, type runIMSCmd. Leave the type of operation as REQUEST_RESPONSE because there will be two messages, one for the request to run the IMS transaction and one for the response from the IMS transaction. Click **Next**.
12. The properties of the interaction with the IMS application program are shown. See Operation binding properties for a description of these properties. Type the property values appropriate for your application. For example:
 - In the **imsRequestType** field, type 2.
 - Accept all the other defaults.

Click **Next**.
13. In the Operation Binding page, click **Import** next to the input message. The File Selection page opens. Import the IMSCmd.cpp file to create the XML schema definition for the input part.
 - a. Expand **IMSCmdSample > sample > ims > imscmd** and select **IMSCmd.cpp**. Click **Next**.
 - b. In the COBOL Import Properties page, enter the following values:

Note: If you choose the z/OS platform, the values for all the fields except TRUNC are automatically filled in as shown in the following table. Because most IMS programs are compiled with the TRUNC(BIN) option, it is recommended that you change the value of TRUNC from STD to BIN.

Platform	z/OS
Codepage	037
Floating point format	IBM 390 Hexadecimal
Endian	Big
Remote integer endian	Big
External decimal sign	EBCDIC
QUOTE	DOUBLE
TRUNC	BIN
NSYMBOL	DBCS

Other values in the table above might differ for your environment. For example, you might need to specify a different value for the **Codepage** field if your IMS data is in a code page other than U.S. English (037). The value for the **QUOTE** field also might differ, depending on your COBOL source. Click **Next**.

14. In the COBOL Importer page, the data structures from the file are displayed. Select **INPUTMSG**, which will populate the XSD type name with **INPUTMSG**. You can accept the default to overwrite the XSD types. Click **Finish**.
15. In the Operation Binding page, click **Import** next to the output message. The File Selection page opens. You import the `IMSCmd.ccp` file to specify the XML schema definition for the output part.
 - a. Select the same **IMSCmd.ccp** from the `IMSCmdSample` package. In this sample, both the input and output message definitions are contained in the same COBOL source file, `IMSCmd.ccp`. Click **Next**.
 - b. In the COBOL Import Properties page, specify the same values that you entered in step 11b for input. Click **Next**.
 - c. In the COBOL Import window, select **OUTPUTMSG** in the data structures list, which will populate the XSD type name with **OUTPUTMSG**. You can accept the default to overwrite the XSD types. Click **Finish** to return to the Operation Binding page.
16. On the Operation Binding page, click **Next**.
17. In the Operation Binding summary page, the new operation information is displayed. Click **Finish**. The wizard populates WSDL files with the operation information.
18. Press **Ctrl-S** to save the file and close the editor by clicking the **X** next to `IMSCmdIMSBinding.wsdl`. Note that you must save the changes in order to successfully proceed with the sample. Also, close the Service Provider page in the editor. The service interface file `IMSCmdIMSBinding.wsdl` and the service binding file `IMSCmdIMSService.wsdl` are updated in the `sample.ims.ims cmd` package of the `IMSCmdSample` project.

Now you can create a proxy to directly access the service you just created.

Step 4: Creating a Java service proxy

The Java service proxy provides a remote procedure call interface to the service. Using the proxy, the application calls a remote method on the service as if the

method were a local one. Once the application makes the remote call, the proxy handles all of the communication details between the application and service.

To create the Java service proxy, complete the following steps:

1. Expand **IMSCmdSample** > **sample.ims.ims cmd**, right-click the service binding file **IMSCmdIMSService.wsdl**, and select **Enterprise Services** > **Generate a service proxy**. The Generate Service Proxy wizard opens.
2. In the Proxy Selection page, ensure **Web Services Invocation Framework (WSIF)** is selected and click **Next**.
3. In the Service Proxy page, verify that the service you want to create the proxy for is shown. Verify that the proxy class properties are correct. Because you selected a file in the previous step, most fields are populated with default values. These default values are generated based on the contents of the selected service binding file.
 - a. **Generate helper classes** is selected by default. Your service will need these Java helper classes because you are creating a service that includes complex types.
 - b. Verify that the source folder is `/IMSCmdSample`.
 - c. Verify that the package name is `sample.ims.ims cmd`.
 - d. Type `IMSCmdIMSProxy` for the class name.
 - e. Click **Next**.
4. In the Service Proxy Style page, specify the style of the proxy and the operations to expose in the proxy:
 - a. Select the **Command bean** proxy style.
 - b. Select the `runIMSCmd` method.
5. Click **Finish**. The Java service proxy `IMSCmdIMSProxy` is generated in the `IMSCmdSample` project.

Next, write a Java class to test the proxy.

Step 5: Testing the Java service proxy

To test the Java service proxy, you need to write client code that executes the proxy. This code sets parameters for the input message, invokes the proxy, passes the input message to the proxy, receives an output message back from the proxy, and then displays the message on the console.

To write the Java class, complete the following steps:

1. Import the file `IMSCmdOutput.java` into the package `sample.ims.ims cmd` from the directory:

```
WS_installdir\wstools\eclipse\plugins\com.ibm.etools.ctc.samples.ims_5.1.0  
\sampleparts
```

where `WS_installdir` is the directory where you installed WebSphere Application Developer Integration Edition.

2. Expand the **IMSCmdSample** project and then select the **sample.ims.ims cmd** package. From the toolbar, click the **New Java class** icon. You will create a Java class that uses the IMS proxy to invoke the IMS service. The service submits a command to IMS, then returns the output of the command to the application.
3. Type in the new class name: `TestIMSCmdIMS`.
4. Click the **public static void main(String[]args)** check box. Accept all other defaults and click **Finish**.

5. In the editor that opens, replace the code with the following Java code:

6. package sample.ims.ims cmd;

```
import com.ibm.connector2.ims.ico.*;
import java.io.*;

public class TestIMSCmdIMS
{
    public static void main(String[] args)
    {
        try
        {
            INMSG input = new INMSG();
            input.setInll((short) 34);
            input.setInzz((short) 0);
            input.setIncmd("/START OTMA");
            IMSCmdIMSProxy proxy = new IMSCmdIMSProxy();
            proxy.setINMSG(input);

            // Run the IMS service.
            proxy.execute();

            // Retrieve the command output as a byte array.
            OUTMSG output = proxy.getOUTMSGPart();
            ByteArrayOutputStream outputStream = new
            ByteArrayOutputStream();

            ((org.apache.wsif.providers.jca.WSIFFormatHandler_JCA)
            output._getFormatHandler()).write(outputStream);
            byte[] b = outputStream.toByteArray();

            // Use byte array to populate
            IMSCommandOutput object;.
            IMSCommandOutput cmdOut = new
            IMSCommandOutput(b);

            // Print command output.
            System.out.println(
                "\nCommand output as concatenated string:"
                + cmdOut.getMessage());

            java.util.Enumeration en = (cmdOut.
            getMessageSegments()).elements();
            System.out.println("\nCommand output as segments:");
            while (en.hasMoreElements())
            {
                System.out.println(en.nextElement());
            }
        }
        catch (Exception e)
        {
            if (e instanceof org.apache.wsif.WSIFException)
            {
                Throwable ic4jEx =
                    ((org.apache.wsif.WSIFException) e).
                    getTargetException();
                System.out.println(
                    "\nIMS Connector threw exception: " + ic4jEx);
            }
            else
            {
                System.out.println("\nCaught exception is: " + e);
            }
        }
    }
}
```

7. Close the editor and click **Yes** to save the changes.
8. Select the **TestIMSCmdIMS.java** class from the package explorer. Expand the **Run** icon on the toolbar by selecting the arrow beside it. From the pop-up menu, select **Run As > Java Application**.

In the console view, you will see the following command output:

```
Command output as concatenated string:  
DFS058I 17:45:13 START COMMAND COMPLETED
```

```
Command output as segments:  
DFS058I 17:45:13 START COMMAND COMPLETED
```

Explanation:

- Because the output of IMS commands vary in length and number of segments, the Java application that invokes the service must contain additional logic. The output of a service can be retrieved as a byte array. In the case of an IMS command, the byte array consists of one or more segments of EBCDIC characters, each segment starting with a two byte length (LL), followed by two flag bytes. In the sample code, the `_getFormatHandler()` method of the class, `com.ibm.wsif.format.jca.WSIFFormatPart_JCA`, is used in retrieving the command output as a `ByteArrayOutputStream` object, which is then converted to a byte array.
- The byte array is used to populate an `IMSCommandOutput` object. The `IMSCommandOutput` class is provided as a sample class in:

```
WS_installdir\wstools\eclipse\plugins\com.ibm.etools.ctc.samples.ims_5.1.0  
\sampleparts
```

. It provides methods for retrieving the command output as a concatenated string of all the segments or as individual segments. Since the output of an IMS command is in U.S. English, the `IMSCommandOutput` class converts the byte array from EBCDIC to unicode using code page 037.

Sample: Building container-managed and component-managed transactional EJBs to run IMS transactions

Objectives

The main objective of this sample is to describe how to build two transactional EJBs, container-managed and component-managed, that use IMS Connector for Java to run a single resource global transaction using two-phase commit.

This sample demonstrates how to:

- Create single action container-managed and component-managed transactional methods (methods that perform one add, delete, update, or display command against an IMS database).
- Implement multi-action methods bundled within container-managed and component-managed transactions.
- Create a servlet to call a container-managed or component-managed bean.
- Deploy and run the application using the HTTP server and WebSphere Application Server for z/OS.

Time required

Allow 90 minutes. This will give you enough time to create the enterprise service, the EJB sessions beans, and the EJB methods, generate web pages from the Java bean, and run the sample.

Before you begin

This sample assumes that WebSphere Studio Application Developer Integrated Edition, Version 5.1 is installed on your machine and that you are familiar with using the product. It also assumes that the IMS resource adapter (also known as IMS Connector for Java) is correctly installed and configured on the WebSphere Application Server. See Importing a resource adapter for more information about importing the IMS resource adapter. Before performing the steps in this sample, you must have access to Sample: Creating an enterprise service for an IMS transaction.

Description

This sample simulates a joint-policy insurance customer database (a phone book) in which the insurance agent stores information about couples and individuals. The insurance agent uses methods such as addCouple, deleteCouple, updateCouple, and displayCouple. Each couple in the database consists of a pair of individual entries and a set of status records that link each individual to the other individual in the couple. Each method to add, delete, update, or display a couple is bundled into a single transaction. This means that every aspect of this transaction must be successful or the entire transaction will be rolled back.

This sample leads you through detailed steps that describe how to generate transactional EJBs to run an IMS transaction. Within WebSphere Studio, you will use wizards to generate code for the service, and then deploy the code to the WebSphere test environment that is shipped with WebSphere Studio. You will also generate session beans to access the service, and you will add EJB methods to remote interfaces to be used by client applications. The client application provides input data for the IMS input message, which the service passes to the IMS system. The IMS transaction runs and returns an output message, the contents of which are returned to the client by the service. For this sample, you run all of the server and client applications on the same machine. The steps in this sample are:

1. Preparing the sample
 - a. Generating the enterprise service
 - b. Generating an EJB proxy
 - c. Creating exception and output classes
2. Creating the sample
 - a. Creating EJB session beans for container-managed and component-managed EIS sign-on
 - b. Creating EJB business methods
 - c. Adding the EJB methods to the remote interfaces
 - d. Setting the EJB transaction attributes for the container-managed bean
 - e. Generating deployed code
 - f. Creating an error page
 - g. Generating web pages from a Java bean
3. Configuring the server and deploying the sample

- a. Configuring the server and deploying the EAR project
- b. Running the sample

Part 1: Preparing the sample

Before you build container-managed and component-managed EJB beans, you need to prepare the sample by first creating the enterprise service, then generating an EJB proxy, and finally creating exception and output classes.

Step 1: Creating the enterprise service

To create an enterprise service, you must complete all of the steps described in *Part 1: Creating the enterprise service* in the Sample: Creating an enterprise service for an IMS transaction with one exception. Replace *Step 6: Binding the resource reference* in the sample and follow these steps instead:

1. Click the **J2EE Hierarchy** tab in the Business Integration perspective and expand **EJB Modules**.
2. Double-click **myIMSPhoneBookServiceEJB** to open the Deployment Descriptor Editor. Click the **References** tab.
3. In the References page, expand **MyPhoneBookService** and select the **ResourceRef** element.
4. Select **Application** in the Authentication field.
5. Under **WebSphere Bindings**, type **myIMSTarget** for the JNDI name.
6. Click the **Assembly Descriptor** tab. Under Container Transactions, click **Add**. The Add Container Transaction wizard opens.
7. On the Enterprise Bean Selection page, select the check box next to the **MyPhoneBookService** bean and click **Next**.
8. On the Container Transaction Type and Method Elements page, select **Required** in the Container transaction type field. Expand **MyPhoneBookService** and select the **runPhoneBook (sample.ims.INPUTMSG)** method. Click **Finish**.
9. Press **Ctrl-S** to save your changes and then close the EJB Deployment Descriptor editor.

Note: Do **not** complete the entire *Creating an enterprise service for an IMS transaction* sample. Complete **only** *Part 1: Creating the enterprise service* and replace *Step 6: Binding the resource reference* with the steps listed above.

Step 2: Generating an EJB proxy

Now you will create an EJB proxy so that later the EJB proxy can access the deployed services. The deployed services are then used by the container-managed and component-managed session beans. To generate an EJB proxy, complete the following steps:

1. Click the **Services** tab of the Business Integration perspective.
2. Expand **Deployable Service > myIMSPhoneBookServiceEJB > ejbModule > sample.ims** and select **myPhoneBookEJBService.wsdl**.
3. Right-click the file and select **Enterprise Services > Generate Service Proxy**. The Generate Service Proxy wizard opens.
4. In the Proxy selection page, select **Web Services Invocation Framework** for the type of proxy to generate. Click **Next**.
5. In the Service Proxy page, the fields contain default values which are based on the service file you selected.

6. Ensure that the **Source folder** is /myIMSPhoneBookServiceEJB/ejbModule.
7. Ensure that the Package name is sample.ims.
8. Change the class name to myPhoneBookEJBProxy.java. Click **Next**.
9. In the Service Proxy style page, select **Client stub** for the Proxy style and then select the **runPhoneBook** check box.
10. Click **Finish**.

The EJB proxy, myPhoneBookEJBProxy, is generated in the myIMSPhoneBookServiceEJB project. The enterprise service WSDL files are also copied into this service project.

Step 3: Creating exception and output classes

An exception class is created to handle transactional logic problems, such as when an attempt is made to add a new couple to the phone book when an individual of that couple already exists in the database. An output class is created to group multiple output messages.

To create a new exception type, complete the following steps:

1. In the Package Explorer view, expand **myIMSPhoneBookServiceEJB > ejbModule** and select the **sample.ims** package.
2. Click the **Create a Java class** icon on the toolbar. The New Java Class wizard opens.
3. In the New Java Class wizard, ensure that the **Source folder** is myIMSPhoneBookServiceEJB/ejbModule and the package is sample.ims.
4. In the **Name** field, type PhoneBookException for the name of the class that you are creating.
5. In the **Superclass field**, type java.lang.Exception.
6. Ensure **Constructors from superclass** and **Inherited abstract methods** methods are selected for which methods to create.
7. Accept the other defaults and click **Finish**. The java class, **PhoneBookException.java**, is created.

To create an output class, complete the following steps:

1. In the Package Explorer view, expand **myIMSPhoneBookServiceEJB > ejbModule** and select the **sample.ims** package.
2. Click the **Create a Java class** icon on the toolbar. The New Java Class wizard opens.
3. In the New Java Class wizard, ensure that the **Source folder** is myIMSPhoneBookServiceEJB/ejbModule and the package is sample.ims.
4. In the **Name** field, type CoupleOutput for the name of the class that you are creating.
5. Click on the **Add** button next to the Interfaces field. The Implemented Interfaces Selection window opens. In the **Choose Interfaces** field, type serializable and click **Ok**. Ensure that java.io.Serializable is in the Interfaces field.
6. Accept all other defaults, and click **Finish**. The java class, **CoupleOutput.java**, is created and is opened in the editor view.
7. Replace the code in the editor with the following Java code:

```
package sample.ims;

import java.io.Serializable;
```

```

public class CoupleOutput implements Serializable {

    OUTPUTMSG output1;
    OUTPUTMSG output2;

    public CoupleOutput() {
        super();
    }

    public CoupleOutput(OUTPUTMSG initOutput1, OUTPUTMSG initOutput2) {
        super();
        this.output1 = initOutput1;
        this.output2 = initOutput2;
    }

    public OUTPUTMSG getOutput1() {
        return output1;
    }

    public OUTPUTMSG getOutput2() {
        return output2;
    }

    public void setOutput1(OUTPUTMSG output1) {
        this.output1 = output1;
    }

    public void setOutput2(OUTPUTMSG output2) {
        this.output2 = output2;
    }

}

```

8. Press **Ctrl-S** to save the file and then close the CoupleOutput.java file.

Part 2: Creating the sample

In this section, you will create two EJB session beans which uses container-managed EIS sign-on and the other for component-managed EIS sign-on. Each session bean will have eight business methods. After you create the EJB session beans, you will create and test methods that perform container-managed or component-managed transactions and verify the transaction output.

Note: In this sample, you will see the terms **container-managed** and **component-managed** used to describe both EIS sign-on and transactions. It is important that you read these instructions carefully to avoid confusion over the use of these terms. For more information about container-managed and component-managed EIS sign-on, see Container-managed EIS sign-on and Component-managed EIS sign-on.

Step 1: Creating EJB session beans for container-managed and component-managed EIS sign-on

To create a container-managed EJB session bean, complete the following steps:

1. In the Business Integration Perspective, click on the **Package Explorer** tab.
2. Select the **myIMSPhoneBookServiceEJB** project.
3. Click the **Create an Enterprise Bean** icon.
4. The New Enterprise Bean wizard opens. Ensure that the EJB project name is myIMSPhoneBookServiceEJB and click **Next**.
5. In the Create a 2.0 Enterprise Bean window:

- a. Ensure that **Session bean** is selected.
 - b. In the **Bean name** field, type `ContPhoneBook`.
 - c. Ensure that the **Source folder** is `/ejbModule`.
 - d. Ensure that the **Default package name** is `sample.ims`. Click **Next**.
6. In the Enterprise Bean Details window:
- a. For the **Session type**, select **stateless**.
 - b. For the **Transaction type**, select **Container**.
 - c. Accept all other defaults and click **Finish**.

To create a component-managed EJB session bean, complete the following steps:

1. Repeat steps 1 through 4 that are listed for creating the container-managed session bean.
2. For steps 5 and 6, follow these steps:
 - In the Create a 2.0 Enterprise Bean window:
 - a. Ensure that **Session bean** is selected.
 - b. In the **Bean name** field type `CompPhoneBook`.
 - c. Ensure that the **Source folder** is `/ejbModule`.
 - d. Ensure that the **Default package name** is `sample.ims`. Click **Next**.
 - In the Enterprise Bean Details window:
 - a. For the **Session type**, select **stateless**.
 - b. For the **Transaction type**, select **bean**.
 - c. Accept all other defaults and click **Finish**.

The `ContPhoneBook` and `CompPhoneBook` session beans have been created and are generated in the **sample.ims** package under the `myIMSPhoneBookServiceEJB` project.

Step 2: Creating EJB business methods

In this section, you will create a total of 16 EJB business methods. Eight business methods are for container-managed beans and eight business methods are for component-managed beans. Some of these business methods verify the output message and some run the EJB proxy to perform a single or multi-action container or component-managed transaction and verify the transaction output. The business methods that you will create for each bean are:

- `add()`
- `delete()`
- `display()`
- `update()`
- `addCouple()`
- `deleteCouple()`
- `displayCouple()`
- `updateCouple()`

To create the EJB business methods for the component-managed bean, complete the following steps:

1. In the Package Explorer view, expand **myIMSPhoneBookServiceEJB > ejbModule > sample.ims** and double-click the component-managed java bean, **CompPhoneBookBean.java**, to open the Deployment Descriptor editor.

2. In the editor view, replace the code in the editor with the following Java code:

```
package sample.ims;
import javax.transaction.*;

public class CompPhoneBookBean implements javax.ejb.SessionBean {
    private javax.ejb.SessionContext mySessionCtx;

    public javax.ejb.SessionContext getSessionContext() {
        return mySessionCtx;
    }
    public void setSessionContext(javax.ejb.SessionContext ctx){
        mySessionCtx = ctx;
    }
    public void ejbActivate() {
    }
    public void ejbCreate() throws javax.ejb.CreateException {
    }
    public void ejbPassivate() {
    }
    public void ejbRemove() {
    }
    public void verifyOutput(OUTPUTMSG output, String cmd)
        throws PhoneBookException {
        try {
            if (output == null)
                throw new PhoneBookException("NULL
                OUTPUT");
            String outMsg = output.getOut__msg().trim();
            if (cmd.equals("DISPLAY") ||
                cmd.equals("ADD")) {
                if (!outMsg.equals
                    ("ENTRY WAS " + cmd + "ED"))
                    throw new
                        PhoneBookException
                            (output.get
                                Out__msg());
            } else {
                if (!outMsg.equals("ENTRY WAS " +
                    cmd + "D"))
                    throw new
                        PhoneBookException
                            (output.get
                                Out__msg());
            }
        } catch (PhoneBookException pe) {
            throw pe;
        } catch (Exception e) {
            throw new PhoneBookException
                (e.toString());
        }
    }
    public void verifyOutput(OUTPUTMSG output, String cmd,
        String state)
        throws PhoneBookException {
        try {
            if (output == null)
                throw new PhoneBookException("NULL
                OUTPUT");
            String outMsg = output.getOut__msg().trim();
            if (cmd.equals("DISPLAY") ||
                cmd.equals("ADD")) {
                if (!outMsg.equals("ENTRY WAS " +
                    cmd + "ED"))
                    throw new PhoneBookException(
                        output.getOut__msg() +
                            ": " + state);
            } else {

```

```

        if (!outMsg.equals("ENTRY WAS " +
            cmd + "D"))
            throw new PhoneBookException(
                output.getOut_msg() +
                ": " + state);
    }
} catch (PhoneBookException pe) {
    throw pe;
} catch (Exception e) {
    throw new PhoneBookException(e.toString());
}
}

public OUTPUTMSG phoneBookAction(String lastName,String firstName,
    String zip,String ext,String cmd) throws PhoneBookException {
    UserTransaction transaction = mySessionCtx.
        getUserTransaction();
    try {
        transaction.begin();
        myPhoneBookEJBProxy ejbProxy = new
            myPhoneBookEJBProxy();
        INPUTMSG input = new INPUTMSG();
        input.setIn__ll((short) 59);
        input.setIn__zz((short) 0);
        input.setIn__trcd("IVTNO");
        input.setIn__name1(lastName);
        input.setIn__name2(firstName);
        input.setIn__zip(zip);
        input.setIn__extn(ext);
        input.setIn__cmd(cmd);
        OUTPUTMSG output =
            ejbProxy.runPhoneBook(input);
        verifyOutput(output, cmd);
        transaction.commit();
        return output;
    } catch (Exception ex) {
        try {
            transaction.rollback();
            Exception storedEx;
            throw new
                PhoneBookException(ex.toString());
        } catch (SystemException se) {
            throw new PhoneBookException(
                ex.toString() + ": ROLLBACK
                FAILED:
                " + se.toString());
        }
    }
}

public OUTPUTMSG phoneBookService(String lastName,String firstName,
    String zip,
    String ext, String cmd, String state) throws Exception {
    myPhoneBookEJBProxy ejbProxy = new
        myPhoneBookEJBProxy();
    INPUTMSG input = new INPUTMSG();
    input.setIn__ll((short) 59);
    input.setIn__zz((short) 0);
    input.setIn__trcd("IVTNO");
    input.setIn__name1(lastName);
    input.setIn__name2(firstName);
    input.setIn__zip(zip);
    input.setIn__extn(ext);
    input.setIn__cmd(cmd);
    OUTPUTMSG output = ejbProxy.runPhoneBook(input);
    verifyOutput(output,cmd,state);
    return output;
}

public CoupleOutput phoneBookAction(String lastName1,

```

```

String firstName1,String zip1,
String ext1,String lastName2,String firstName2,String zip2,
String ext2,String cmd)throws PhoneBookException {
    OUTPUTMSG output1,output2,coupleStatus1,coupleStatus2;
    String hashCode1, hashCode2;
    UserTransaction transaction =
    mySessionCtx.getUserTransaction();
    try {
        transaction.begin();
        output1 = phoneBookService(lastName1,firstName1,
        zip1,ext1,cmd,
            "FIRST INDIVIDUAL");
        lastName1 = lastName1.toUpperCase();
        int h = lastName1.hashCode();
        if (h<0) h=0-h;
        hashCode1 = String.valueOf(h);
        coupleStatus1 = phoneBookService(hashCode1,
        lastName2,zip1,
            ext1,cmd,"FIRST COUPLE STATUS");
        if (cmd.compareTo("DISPLAY")==0)
            lastName2 = coupleStatus1.getOut__name2();
        output2 = phoneBookService(lastName2,firstName2,
        zip2,ext2,cmd,
            "SECOND INDIVIDUAL");
        if (cmd.compareTo("DISPLAY")!=0) {
            lastName2 = lastName2.toUpperCase();
            h = lastName2.hashCode();
            if (h<0) h=0-h;
            hashCode2 = String.valueOf(h);
            coupleStatus2 = phoneBookService(hashCode2,
            lastName1,zip2,ext2,cmd,"SECOND COUPLE
            STATUS");
        }
        transaction.commit();
        return new CoupleOutput(output1,output2);
    } catch (Exception ex) {
        try {
            transaction.rollback();
            System.out.println("EXCEPTION CAUGHT");
            Exception storedEx;
            throw new
            PhoneBookException(ex.toString());
        } catch (SystemException se) {
            throw new PhoneBookException(
                ex.toString() + ": ROLLBACK
                FAILED: "
                + se.toString());
        }
    }
}

public OUTPUTMSG display(String lastName, String firstName, String zip,
String ext) throws PhoneBookException {
    return phoneBookAction(lastName, firstName, zip, ext,
    "DISPLAY");
}

public OUTPUTMSG add(String lastName, String firstName, String zip,
String ext) throws PhoneBookException {
    return phoneBookAction(lastName, firstName, zip, ext,
    "ADD");
}

public OUTPUTMSG delete(String lastName, String firstName, String zip,
String ext) throws PhoneBookException {
    return phoneBookAction(lastName, firstName, zip, ext,
    "DELETE");
}

public OUTPUTMSG update(String lastName, String firstName,String zip,
String ext) throws PhoneBookException {

```



```

        return phoneBookAction(lastName, firstName, zip, ext,
                                "UPDATE");
    }
    public CoupleOutput displayCouple(String lastName1,
    String firstName1,String zip1,
    String ext1) throws PhoneBookException {
        return phoneBookAction(lastName1,firstName1,
                                zip1,ext1,"","","","DISPLAY");
    }
    public CoupleOutput addCouple(String lastName1,
    String firstName1,String zip1,
    String ext1,String lastName2,String firstName2,
    String zip2,String ext2)
    throws PhoneBookException {
        return phoneBookAction(lastName1,firstName1,
                                zip1,ext1,
                                lastName2,
                                firstName2,
                                zip2,ext2,"ADD");
    }
    public CoupleOutput deleteCouple(String lastName1,
    String firstName1,String zip1,
    String ext1,String lastName2,String firstName2,
    String zip2,String ext2)
    throws PhoneBookException {
        return phoneBookAction(lastName1,firstName1,
                                zip1,ext1,
                                lastName2,
                                firstName2,
                                zip2,ext2,"DELETE");
    }
    public CoupleOutput updateCouple(String lastName1,
    String firstName1,String zip1,
    String ext1,String lastName2,String firstName2,
    String zip2,String ext2)
    throws PhoneBookException {
        return phoneBookAction(lastName1,firstName1,
                                zip1,ext1,
                                lastName2,
                                firstName2,
                                zip2,ext2,"UPDATE");
    }
}

```

3. Press **Ctrl-S** to save the file and then close the `CompPhoneBookBean.java` file.

To create the EJB business methods for the container-managed bean, complete the following steps:

1. In the Package Explorer view, expand **myIMSPhoneBookServiceEJB > ejbModule > sample.ims** and double-click the container-managed java bean, **ContPhoneBookBean.java**, to open the Deployment Descriptor editor.
2. In the editor view, replace the code in the editor with the following Java code:

```

package sample.ims;

public class ContPhoneBookBean implements javax.ejb.SessionBean {
    private javax.ejb.SessionContext mySessionCtx;
    public javax.ejb.SessionContext getSessionContext() {
        return mySessionCtx;
    }
    public void setSessionContext(javax.ejb.SessionContext ctx){
        mySessionCtx = ctx;
    }
    public void ejbActivate() {
    }
    public void ejbCreate() throws javax.ejb.CreateException {

```

```

}
public void ejbPassivate() {
}
public void ejbRemove() {
}
public void verifyOutput(OUTPUTMSG output, String cmd)
    throws PhoneBookException {
    try {
        if (output == null)
            throw new PhoneBookException("NULL
            OUTPUT");
        String outMsg = output.getOut__msg().trim();
        if (cmd.equals("DISPLAY") ||
            cmd.equals("ADD")) {
            if (!outMsg.equals("ENTRY WAS " +
                cmd + "ED"))
                throw new PhoneBookException(
                    output.getOut__msg());
        } else {
            if (!outMsg.equals("ENTRY WAS " +
                cmd + "D"))
                throw new PhoneBookException(
                    output.getOut__msg());
        }
    } catch (PhoneBookException pe) {
        throw pe;
    } catch (Exception e) {
        throw new PhoneBookException(e.toString());
    }
}

public void verifyOutput(OUTPUTMSG output, String cmd, String
state)
    throws PhoneBookException {
    try {
        if (output == null)
            throw new PhoneBookException("NULL
            OUTPUT");
        String outMsg = output.getOut__msg().trim();
        if (cmd.equals("DISPLAY") ||
            cmd.equals("ADD")) {
            if (!outMsg.equals("ENTRY WAS " +
                cmd + "ED"))
                throw new PhoneBookException(
                    output.getOut__msg() +
                    ": " + state);
        } else {
            if (!outMsg.equals("ENTRY WAS " + cmd +
                "D"))
                throw new PhoneBookException(
                    output.getOut__msg() + ":
                    " + state);
        }
    } catch (PhoneBookException pe) {
        throw pe;
    } catch (Exception e) {
        throw new PhoneBookException(e.toString());
    }
}

public OUTPUTMSG phoneBookAction(
    String lastName,
    String firstName,
    String zip,
    String ext,
    String cmd)
    throws PhoneBookException {
    try {
        myPhoneBookEJBProxy ejbProxy = new

```

```

        myPhoneBookEJBProxy();
        INPUTMSG input = new INPUTMSG();
        input.setIn__ll((short) 59);
        input.setIn__zz((short) 0);
        input.setIn__trcd("IVTNO");
        input.setIn__name1(lastName);
        input.setIn__name2(firstName);
        input.setIn__zip(zip);
        input.setIn__extn(ext);
        input.setIn__cmd(cmd);
        OUTPUTMSG output =
            ejbProxy.runPhoneBook(input);
        verifyOutput(output, cmd);
        return output;
    } catch (Exception ex) {
        mySessionCtx.setRollbackOnly();
        Exception storedEx;
        throw new
            PhoneBookException(ex.toString());
    }
}

public OUTPUTMSG phoneBookService(
    String lastName,
    String firstName,
    String zip,
    String ext,
    String cmd,
    String state)
    throws Exception {
    myPhoneBookEJBProxy ejbProxy = new myPhoneBookEJBProxy();
    INPUTMSG input = new INPUTMSG();
    input.setIn__ll((short) 59);
    input.setIn__zz((short) 0);
    input.setIn__trcd("IVTNO");
    input.setIn__name1(lastName);
    input.setIn__name2(firstName);
    input.setIn__zip(zip);
    input.setIn__extn(ext);
    input.setIn__cmd(cmd);
    OUTPUTMSG output = ejbProxy.runPhoneBook(input);
    verifyOutput(output, cmd, state);
    return output;
}

public CoupleOutput phoneBookAction(
    String lastName1,
    String firstName1,
    String zip1,
    String ext1,
    String lastName2,
    String firstName2,
    String zip2,
    String ext2,
    String cmd)
    throws PhoneBookException {
    OUTPUTMSG output1, output2, coupleStatus1, coupleStatus2;
    String hashCode1, hashCode2;
    try {
        output1 =
            phoneBookService(
                lastName1,
                firstName1,
                zip1,
                ext1,
                cmd,
                "FIRST INDIVIDUAL");
        lastName1 = lastName1.toUpperCase();
        int h = lastName1.hashCode();

```

```

        if (h < 0)
            h = 0 - h;
        hashCode1 = String.valueOf(h);
        coupleStatus1 =
            phoneBookService(
                hashCode1,
                lastName2,
                zip1,
                ext1,
                cmd,
                "FIRST COUPLE STATUS");
        if (cmd.compareTo("DISPLAY") == 0)
            lastName2 = coupleStatus1.getOut__name2();
        output2 =
            phoneBookService(
                lastName2,
                firstName2,
                zip2,
                ext2,
                cmd,
                "SECOND INDIVIDUAL");
        if (cmd.compareTo("DISPLAY") != 0) {
            lastName2 = lastName2.toUpperCase();
            h = lastName2.hashCode();
            if (h < 0)
                h = 0 - h;
            hashCode2 = String.valueOf(h);
            coupleStatus2 =
                phoneBookService(
                    hashCode2,
                    lastName1,
                    zip2,
                    ext2,
                    cmd,
                    "SECOND COUPLE STATUS");
        }
        return new CoupleOutput(output1, output2);
    } catch (Exception ex) {
        mySessionCtx.setRollbackOnly();
        Exception storedEx;
        throw new PhoneBookException(ex.toString());
    }
}

public OUTPUTMSG display(
    String lastName,
    String firstName,
    String zip,
    String ext)
    throws PhoneBookException {
    return phoneBookAction(lastName, firstName, zip,
        ext, "DISPLAY");
}

public OUTPUTMSG add(
    String lastName,
    String firstName,
    String zip,
    String ext)
    throws PhoneBookException {
    return phoneBookAction(lastName, firstName, zip,
        ext, "ADD");
}

public OUTPUTMSG delete(
    String lastName,
    String firstName,
    String zip,
    String ext)
    throws PhoneBookException {

```

```

        return phoneBookAction(lastName, firstName, zip,
                                ext, "DELETE");
    }
    public OUTPUTMSG update(
        String lastName,
        String firstName,
        String zip,
        String ext)
        throws PhoneBookException {
        return phoneBookAction(lastName, firstName, zip,
                                ext, "UPDATE");
    }
    public CoupleOutput displayCouple(
        String lastName1,
        String firstName1,
        String zip1,
        String ext1)
        throws PhoneBookException {
        return phoneBookAction(
            lastName1,
            firstName1,
            zip1,
            ext1,
            "",
            "",
            "",
            "",
            "DISPLAY");
    }
    public CoupleOutput addCouple(
        String lastName1,
        String firstName1,
        String zip1,
        String ext1,
        String lastName2,
        String firstName2,
        String zip2,
        String ext2)
        throws PhoneBookException {
        return phoneBookAction(
            lastName1,
            firstName1,
            zip1,
            ext1,
            lastName2,
            firstName2,
            zip2,
            ext2,
            "ADD");
    }
    public CoupleOutput deleteCouple(
        String lastName1,
        String firstName1,
        String zip1,
        String ext1,
        String lastName2,
        String firstName2,
        String zip2,
        String ext2)
        throws PhoneBookException {
        return phoneBookAction(
            lastName1,
            firstName1,
            zip1,
            ext1,
            lastName2,
            firstName2,

```

```

        zip2,
        ext2,
        "DELETE");
    }
    public CoupleOutput updateCouple(
        String lastName1,
        String firstName1,
        String zip1,
        String ext1,
        String lastName2,
        String firstName2,
        String zip2,
        String ext2)
        throws PhoneBookException {
        return phoneBookAction(
            lastName1,
            firstName1,
            zip1,
            ext1,
            lastName2,
            firstName2,
            zip2,
            ext2,
            "UPDATE");
    }
}

```

3. Press **Ctrl-S** to save the file and then close the ContPhoneBookBean.java file.

Step 3: Adding the container-managed and component-managed EJB methods to the remote interfaces

The remote interfaces specify which business methods of an EJB can be used by client applications. Generating the EJB automatically builds an empty Remote Interface and a Home Interface with a default *create* method. To make the EJB business methods you created in Part 1, Step 2, accessible to client applications, add the methods to the Remote Interface by completing the following steps:

1. In the Package Explorer view, expand **myIMSPhoneBookServiceEJB > ejbModule > sample.ims** and double-click the **CompPhoneBookBean.java** file to open the file in the Deployment Descriptor editor view.
2. In the Outline pane, expand the **CompPhoneBookBean.java** class. Select the following methods:
 - display()
 - add()
 - delete()
 - update()
 - displayCouple()
 - addCouple()
 - deleteCouple()
 - updateCouple()
3. Right-click any of the selected files and select **Enterprise Bean > Promote to Remote Interface**. This adds a declaration for each of the selected methods to the Remote Interface. The Remote Interface declarations you created are stored in the file, CompPhoneBook.java.
4. Repeat steps 1 through 3 for the **ContPhoneBookBean.java** file. The Remote Interface declarations you create are stored in the file, ContPhoneBook.java.

Step 4: Setting the EJB transaction attributes for the container-managed bean

Setting the transaction attributes specifies how a container should manage a specific method or all of the methods within an EJB's remote interface. To set the EJB transaction attributes, complete the following steps:

1. Click the **J2EE Hierarchy** tab in the Business Integration perspective and expand **EJB Modules**. Double-click **myIMSPhoneBookServiceEJB** to open the Deployment Descriptor editor.
2. Click the **Assembly Descriptor** tab.
3. Under Container Transactions, click **Add**. The Add Container Transaction wizard opens. In the Enterprise Bean Selection page, select the **ContPhoneBook** check box and click **Next**.
4. In the Container Transaction Type and Method Elements page, select **Required** for the Container Transaction type.
5. Expand the ContPhoneBook bean and select all of the methods found in the remote interface. The methods include:
 - add()
 - addCouple()
 - delete()
 - deleteCouple()
 - display()
 - displayCouple()
 - update()
 - updateCouple()
6. Click **Finish**.
7. Press **Ctrl-S** to save the changes and then close the EJB Deployment Descriptor editor.

Step 5: Generating deployed code

To run the sample, you must first generate the deployed code for the EJB session beans. The deployed classes allow your beans to run on an EJB server. To generate the deployed code, follow these steps:

1. Click on the **J2EE Hierarchy** tab in the Business Integration perspective and expand **EJB Modules**.
2. Right-click **myIMSPhoneBookServiceEJB**, and select **Generate > Deployment and RMIC Code**.
3. The Generate Deployment and RMIC Code wizard opens. Select all the enterprise java beans (**CompPhoneBook**, **ContPhoneBook**, **myPhoneBookService**) and click **Finish**.

Step 6: Generating access beans

This step provides instructions on how to create a java bean wrapper type access bean. The access bean simplifies access to the Home and Remote Interfaces of your enterprise bean and allows a standard java bean approach to using your EJB. To generate an access bean, complete the following steps:

1. Click on the **J2EE Hierarchy** tab of the Business Integration perspective and expand **EJB Modules**.
2. Right-click **myIMSPhoneBookServiceEJB**, and select **New > Access Bean**. The Add an Access Bean wizard opens.
3. Select **Java bean wrapper** for the **access bean type** and click **Next**.

4. Ensure that the EJB project name is myIMSPhoneBookServiceEJB and select the **CompPhoneBook** and **ContPhoneBook** enterprise beans. Click **Finish**.
5. Click on the **Package Explorer** tab and right-click **myIMSPhoneBookServiceWeb**. Select **Properties**.
6. In the Properties for myIMSPhoneBookServiceWeb window, select **Java Build Path** and click the **Projects** tab. Accept all the defaults and select the **myIMSPhoneBookServiceEJB** check box.
7. Click **OK**.

Step 7: Creating an error page

Before you can create a servlet, you must create an error page to capture any exceptions encountered by the servlet. To create an error page, complete the following steps:

1. Click on the **Package Explorer** tab and right-click on **myIMSPhoneBookServiceWeb**. Select **New > Other**.
2. The Select page of the New window opens. In the left-pane, select **Web** and in the right-pane, select **JSP File**.
3. Click **Next**.
4. The New JSP File wizard opens. In the wizard:
 - Ensure that the **Folder** name is /myIMSPhoneBookServiceWeb/WebContent.
 - For the **JSP File name**, type CouplePhoneBookError.
 - Ensure that the **Mark up Language** is HTML.
 - Click **Finish**. The editor opens with your **CouplePhoneBookError.jsp** file.
5. In the editor view, click the **Source** tab and replace the code with the following Java and HTML code:

```
<%@ page isErrorPage="true" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<html>
<title>Error Page</title>
<body>
<form method="post" action="CouplePhoneBookResultsForm.jsp">
<table border="0" width="600">
<tr>
<td colspan="5"><h2
align="center"><br>IMS Connector for Java Joint Insurance
Policy Phonebook <br></h2>
</td>
<tr>
<td colspan="5"><b>RESULT: </b>EXCEPTION WAS THROWN<td>
</tr>
<tr height="10"></tr>
<tr>
<td colspan="5">Please provide the necessary information for your
command.</td>
</tr>
<tr height="10"></tr>
<tr>
<th colspan="2" width="280">FIRST INDIVIDUAL</th>
<td width="40"></td>
<th colspan="2" width="280">SECOND INDIVIDUAL</th>
</tr>
<tr>
<td width="100">Last Name</td>
<td width="180">
<input type="text" name="FIRST_IN__NAME1" id="FIRST_IN__NAME1"
size="20" maxlength="20" >
</td>
```



```

<td width="50"></td>
<td width="100">Last name</td>
<td width="180">
  <input type="text" name="SECOND_IN__NAME1" id="SECOND_IN__NAME1"
    size="20" maxlength="20" >
</td>
</tr>
<tr>
<td>First Name</td>
<td>
  <input type="text" name="FIRST_IN__NAME2" id="FIRST_IN__NAME2"
    size="20" maxlength="20" >
</td>
<td width="50"></td>
<td>First Name</td>
<td>
  <input type="text" name="SECOND_IN__NAME2" id="SECOND_IN__NAME2"
    size="20" maxlength="20" >
</td>
</tr>
<tr>
<td>Extension</td>
<td>
  <input type="text" name="FIRST_IN__EXTN" id="FIRST_IN__EXTN" size="20"
    maxlength="20" >
</td>
<td width="50"></td>
<td>Extension</td>
<td>
  <input type="text" name="SECOND_IN__EXTN" id="SECOND_IN__EXTN"
    size="20" maxlength="20" >
</td>
</tr>
<tr>
<td>Zip code</td>
<td>
  <input type="text" name="FIRST_IN__ZIP" id="FIRST_IN__ZIP"
    size="20" maxlength="20" >
</td>
<td width="50"></td>
<td>Zip code</td>
<td>
  <input type="text" name="SECOND_IN__ZIP" id="SECOND_IN__ZIP"
    size="20" maxlength="20" >
</td>
</tr>
</table>
<br>
<table border="0" width="600">
  <tr>
    <td colspan="4" width="280" valign="center" align="left">Select
a command for one individual:</td>
    <td width="40"></td>
    <td colspan="4" width="280" valign="center" align="left">Select
a command for a couple:</td>
  </tr>
  <tr>
    <td width="40"></td>
    <td width="100"><input type="radio" checked name="CMDBUTTON"
value="DISPLAY">Display</td>
    <td width="100"><input type="radio" name="CMDBUTTON"
value="ADD">Add</td>
    <td width="40"></td>
    <td width="40"></td>
    <td width="40"></td>
    <td width="100"><input type="radio" name="CMDBUTTON"
value="DISPLAY_COUPLE">Display</td>
  </tr>
</table>

```

```

        <td width="100"><input type="radio" name="CMDBUTTON"
        value="ADD_COUPLE">Add</td>
        <td width="40"></td>
    </tr>
    <tr>
        <td width="40"></td>
        <td width="100"><input type="radio" name="CMDBUTTON"
        value="DELETE">Delete</td>
        <td width="100"><input type="radio" name="CMDBUTTON"
        value="UPDATE">Update</td>
        <td width="40"></td>
        <td width="40"></td>
        <td width="40"></td>
        <td width="100"><input type="radio" name="CMDBUTTON"
        value="DELETE_COUPLE">Delete</td>
        <td width="100"><input type="radio" name="CMDBUTTON"
        value="UPDATE_COUPLE">Update</td>
        <td width="40"></td>
    </tr>
    <tr>
        <td colspan="4" width="280" valign="center" align="left">Select
        the type of transaction management:</td>
        <td width="40"></td>
        <td colspan="4" width="280" valign="center" align="center"></td>
    </tr>
    <tr>
        <td width="40"></td>
        <td width="100"><input type="radio" checked name="MANAGEMENT"
        value="CONTAINER">Container</td>
        <td width="100"><input type="radio" name="MANAGEMENT"
        value="COMPONENT">Component</td>
        <td width="40"></td>
        <td width="40"></td>
        <td width="40"></td>
        <td width="100"></td>
        <td width="40"></td>
    </tr>
</table>
<p>
<input type="submit" name="Submit" id="Submit" VALUE="Submit">
<input type="reset" name="Reset" id="Reset" VALUE="Reset">
</p>
</form>
<b>EXCEPTION:</b><br>
<PRE><%=exception.toString() %></PRE>
</body>
</html>

```

6. Press **Ctrl-S** to save the file and then close the **CouplePhoneBookError.jsp** file.

Step 8: Generating web pages from a Java bean

To generate a web page from your Java bean, complete the following steps:

1. Click on the **Package Explorer** tab and right-click **myIMSPhoneBookServiceWeb**. Select **New > Other**.
2. The Select page of the New window opens. In the left-pane, select **Web** and in the right-pane, select **Java Bean Web Pages**.
3. Click **Next**.
4. The Java Bean Web Pages wizard opens. In the page:
 - Ensure that the **Destination folder** is `/myIMSPhoneBookServiceWeb/WebContent`.
 - Ensure that the Java package is `sample.ims`.

- Click **Next**.
5. In the Choose Java Bean page, type `sample.ims.CompPhoneBookAccessBean` in the Bean field and click **Introspect**. The bean properties and methods are displayed.
 6. Select the eight business methods that you created in the EJB, which are:
 - `add()`
 - `display()`
 - `update()`
 - `delete()`
 - `addCouple()`
 - `displayCouple()`
 - `updateCouple()`
 - `deleteCouple()`
 7. Click **Next**.
 8. On the View Bean Data Page, click the check box, **Use Error Page:** and type `/CouplePhoneBookError.jsp` in the field.
 9. Unselect the **Create View Bean(s) to wrapper your data object(s)** check box. Click **Next**.
 10. On the Design the Input Form page, click **Next**.
 11. On the Design the Results Form page, click **Next**.
 12. On the Specify Prefix page, replace the **prefix name** with `CouplePhoneBook` and click **Finish**.
 13. To create the Input Form page in HTML, click on the **Package Explorer** tab. Expand **myIMSPhoneBookServiceWeb > Web Content** and double-click **CouplePhoneBookInputForm.html** to open the HTML file in the editor.
 14. Replace the code in the editor with the following HTML code:

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN">
<html>
<title>Input form</title>
<body>
<form method="post" action="CouplePhoneBookResultsForm.jsp">
<table border="0" width="600">
<tr>
<td colspan="5"><h2 align="center"><br>IMS Connector for Java Joint
Insurance Policy Phonebook <br></h2>
</td>
<tr>
<td colspan="5">Please provide the necessary information for your
command.</td>
</tr>
<tr height="10"></tr>
<tr>
<th colspan="2" width="280">FIRST INDIVIDUAL</th>
<td width="40"></td>
<th colspan="2" width="280">SECOND INDIVIDUAL</th>
</tr>
<tr>
<td width="100">Last Name</td>
<td width="180">
<input type="text" name="FIRST_IN__NAME1" id="FIRST_IN__NAME1"
size="20" maxlength="20" >
</td>
<td width="50"></td>
<td width="100">Last Name</td>
<td width="180">

```

```

        <input type="text" name="SECOND_IN__NAME1" id="SECOND_IN__NAME1"
size="20" maxlength="20" >
    </td>
</tr>
<tr>
    <td>First Name</td>
    <td>
        <input type="text" name="FIRST_IN__NAME2" id="FIRST_IN__NAME2"
size="20" maxlength="20" >
    </td>
    <td width="50"></td>
    <td>First Name</td>
    <td>
        <input type="text" name="SECOND_IN__NAME2" id="SECOND_IN__NAME2"
size="20" maxlength="20" >
    </td>
</tr>
<tr>
    <td>Extension</td>
    <td>
        <input type="text" name="FIRST_IN__EXTN" id="FIRST_IN__EXTN"
size="20" maxlength="20" >
    </td>
    <td width="50"></td>
    <td>Extension</td>
    <td>
        <input type="text" name="SECOND_IN__EXTN" id="SECOND_IN__EXTN"
size="20" maxlength="20" >
    </td>
</tr>
<tr>
    <td>Zip code</td>
    <td>
        <input type="text" name="FIRST_IN__ZIP" id="FIRST_IN__ZIP"
size="20" maxlength="20" >
    </td>
    <td width="50"></td>
    <td>Zip code</td>
    <td>
        <input type="text" name="SECOND_IN__ZIP" id="SECOND_IN__ZIP"
size="20" maxlength="20" >
    </td>
</tr>
</table>
<br>
<table border="0" width="600">
    <tr>
        <td colspan="4" width="280" valign="center" align="left">
            Select a command
            for one individual:</td>
        <td width="40"></td>
        <td colspan="4" width="280" valign="center" align="left">
            Select a command for a couple:</td>
    </tr>
    <tr>
        <td width="40"></td>
        <td width="100"><input type="radio" checked name="CMDBUTTON"
value="DISPLAY">Display</td>
        <td width="100"><input type="radio" name="CMDBUTTON"
value="ADD">Add</td>
        <td width="40"></td>
        <td width="40"></td>
        <td width="40"></td>
        <td width="100"><input type="radio" name="CMDBUTTON"
value="DISPLAY_COUPLE">Display</td>
        <td width="100"><input type="radio" name="CMDBUTTON"
value="ADD_COUPLE">Add</td>
    </tr>
</table>

```

```

        <td width="40"></td>
    </tr>
    <tr>
        <td width="40"></td>
        <td width="100"><input type="radio" name="CMDBUTTON"
value="DELETE">Delete</td>
        <td width="100"><input type="radio" name="CMDBUTTON"
value="UPDATE">Update</td>
        <td width="40"></td>
        <td width="40"></td>
        <td width="40"></td>
        <td width="100"><input type="radio" name="CMDBUTTON"
value="DELETE_COUPLE">Delete</td>
        <td width="100"><input type="radio" name="CMDBUTTON"
value="UPDATE_COUPLE">Update</td>
        <td width="40"></td>
    </tr>
    <tr>
        <td colspan="4" width="280" valign="center" align="left">Select the
type of transaction managment:</td>
        <td width="40"></td>
        <td colspan="4" width="280" valign="center" align="center"></td>
    </tr>
    <tr>
        <td width="40"></td>
        <td width="100"><input type="radio" checked name="MANAGEMENT"
value="CONTAINER">Container</td>
        <td width="100"><input type="radio" name="MANAGEMENT"
value="COMPONENT">Component</td>
        <td width="40"></td>
        <td width="40"></td>
        <td width="40"></td>
        <td width="100"></td>
        <td width="100"></td>
        <td width="40"></td>
    </tr>
</table>
<p>
<input type="submit" name="Submit" id="Submit" VALUE="Submit">

<input type="reset" name="Reset" id="Reset" VALUE="Reset">

</p>
</form>
</body>
</HTML>

```

15. Press **Ctrl-S** to save the file and then close the **CouplePhoneBookInputForm.html** file.
16. To create the Results Form page in HTML, click on the **Package Explorer** tab. Expand **myIMSPhoneBookServiceWeb > Web Content** and double-click **CouplePhoneBookResultsForm.jsp** to open the file in the editor.
17. Replace the code in the editor with the following HTML code:

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN">
<%@ page contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"
errorPage="CouplePhoneBookError.jsp"
%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<meta name="GENERATOR" content="IBM WebSphere Studio">
<title>Results page</title>
</head>
<body>

```

```

<%
sample.ims.ContPhoneBookAccessBean contAccess;
sample.ims.CompPhoneBookAccessBean compAccess;

String lastName1 = request.getParameter("FIRST_IN_NAME1");
String firstName1 = request.getParameter("FIRST_IN_NAME2");
String extension1 = request.getParameter("FIRST_IN_EXTN");
String zipCode1 = request.getParameter("FIRST_IN_ZIP");
String lastName2 = request.getParameter("SECOND_IN_NAME1");
String firstName2 = request.getParameter("SECOND_IN_NAME2");
String extension2 = request.getParameter("SECOND_IN_EXTN");
String zipCode2 = request.getParameter("SECOND_IN_ZIP");
String command = request.getParameter("CMDBUTTON");
String management = request.getParameter("MANAGEMENT");
sample.ims.OUTPUTMSG blank = new sample.ims.OUTPUTMSG();
blank.setOut_name1("");
blank.setOut_name2("");
blank.setOut_extn("");
blank.setOut_zip("");
blank.setOut_msg("");

sample.ims.CoupleOutput coupleOutput = new
sample.ims.CoupleOutput(blank,blank);

    if (management.equals("CONTAINER")) {
        contAccess = new sample.ims.ContPhoneBookAccessBean();
        if (command.equals("DISPLAY")) {
            coupleOutput.setOutput1(contAccess.display(lastName1,
firstName1,zipCode1,extension1));
        }else if (command.equals("ADD"))
            coupleOutput.setOutput1(contAccess.add(lastName1,firstName1,
zipCode1,extension1));
        else if (command.equals("DELETE"))
            coupleOutput.setOutput1(contAccess.delete(lastName1,firstName1,
zipCode1,extension1));
        else if (command.equals("UPDATE"))
            coupleOutput.setOutput1(contAccess.update(lastName1,firstName1,
zipCode1,extension1));
        else if (command.equals("ADD_COUPLE"))
            coupleOutput = contAccess.addCouple(lastName1,firstName1,
zipCode1,extension1,lastName2,
firstName2,zipCode2,extension2);
        else if (command.equals("DELETE_COUPLE"))
            coupleOutput = contAccess.deleteCouple(lastName1,firstName1,
zipCode1,extension1,
lastName2,firstName2,
zipCode2,extension2);
        else if (command.equals("UPDATE_COUPLE"))
            coupleOutput = contAccess.updateCouple(lastName1,firstName1,
zipCode1,extension1,lastName2,firstName2,
zipCode2,extension2);
        else if (command.equals("DISPLAY_COUPLE"))
            coupleOutput = contAccess.displayCouple(lastName1,firstName1,
zipCode1,extension1);
    } else {
        compAccess = new sample.ims.CompPhoneBookAccessBean();
        if (command.equals("DISPLAY"))
            coupleOutput.setOutput1(compAccess.display(lastName1,
firstName1,zipCode1,extension1));
        else if (command.equals("ADD"))
            coupleOutput.setOutput1(compAccess.add(lastName1,firstName1,
zipCode1,extension1));
        else if (command.equals("DELETE"))
            coupleOutput.setOutput1(compAccess.delete(lastName1,firstName1,
zipCode1,extension1));
    }

```

```

        else if (command.equals("UPDATE"))
            coupleOutput.setOutput1(compAccess.update(lastName1,firstName1,
                zipCode1,extension1));
        else if (command.equals("ADD_COUPLE"))
            coupleOutput = compAccess.addCouple(lastName1,firstName1,
                zipCode1,extension1,lastName2,
                firstName2,zipCode2,extension2);
        else if (command.equals("DELETE_COUPLE"))
            coupleOutput = compAccess.deleteCouple(lastName1,firstName1,
                zipCode1,extension1,
                lastName2,firstName2,
                zipCode2,extension2);
        else if (command.equals("UPDATE_COUPLE"))
            coupleOutput = compAccess.updateCouple(lastName1,firstName1,
                zipCode1,extension1,lastName2,firstName2,
                zipCode2,extension2);
        else if (command.equals("DISPLAY_COUPLE"))
            coupleOutput = compAccess.displayCouple(lastName1,firstName1,
                zipCode1,extension1);
    }

%>
<!-- Result Table -->
<form method="post" action="CouplePhoneBookResultsForm.jsp">
<table border="0" width="600">
<tr>
    <td colspan="5"><h2 align="center"><br>IMS Connector for Java Joint
Insurance Policy Phonebook <br></h2>
</td>
<tr>
    <td colspan="5"><b>RESULT: </b>
<%=coupleOutput.getOutput1().getOut__msg() %></td>
</tr>
<tr height="10"></tr>
<tr>
    <td colspan="5">Please provide the necessary information for your
command.</td>
</tr>
<tr height="10"></tr>
<tr>
    <th colspan="2" width="280">FIRST INDIVIDUAL</th>
    <td width="40"></td>
    <th colspan="2" width="280">SECOND INDIVIDUAL</th>
</tr>
<tr>
    <td width="100">Last Name</td>
    <td width="180">
        <input type="text" name="FIRST_IN__NAME1" id="FIRST_IN__NAME1" size="20"
maxlength="20" value=<%=coupleOutput.getOutput1().getOut__name1() %>>
</td>
    <td width="50"></td>
    <td width="100">Last Name</td>
    <td width="180">
        <input type="text" name="SECOND_IN__NAME1" id="SECOND_IN__NAME1" size="20"
maxlength="20" value=<%=coupleOutput.getOutput2().getOut__name1() %>>
</td>
</tr>
<tr>
    <td>First Name</td>
    <td>
        <input type="text" name="FIRST_IN__NAME2" id="FIRST_IN__NAME2" size="20"
maxlength="20" value=<%=coupleOutput.getOutput1().getOut__name2() %>>
</td>
    <td width="50"></td>
    <td>First Name</td>
    <td>
        <input type="text" name="SECOND_IN__NAME2" id="SECOND_IN__NAME2" size="20"
maxlength="20" value=<%=coupleOutput.getOutput2().getOut__name2() %>>

```

```

        </td>
    </tr>
    <tr>
        <td>Extension</td>
        <td>
            <input type="text" name="FIRST_IN_EXTN" id="FIRST_IN_EXTN" size="20"
maxlength="20" value=<%=coupleOutput.getOutput1().getOut_extn() %>>
        </td>
        <td width="50"></td>
        <td>Extension</td>
        <td>
            <input type="text" name="SECOND_IN_EXTN" id="SECOND_IN_EXTN" size="20"
maxlength="20" value=<%=coupleOutput.getOutput2().getOut_extn() %>>
        </td>
    </tr>
    <tr>
        <td>Zip code</td>
        <td>
            <input type="text" name="FIRST_IN_ZIP" id="FIRST_IN_ZIP" size="20"
maxlength="20" value=<%=coupleOutput.getOutput1().getOut_zip() %>>
        </td>
        <td width="50"></td>
        <td>Zip code</td>
        <td>
            <input type="text" name="SECOND_IN_ZIP" id="SECOND_IN_ZIP" size="20"
maxlength="20" value=<%=coupleOutput.getOutput2().getOut_zip() %>>
        </td>
    </tr>
</table>
<br>
<table border="0" width="600">
    <tr>
        <td colspan="4" width="280" valign="center" align="left">Select a command
for one individual:</td>
        <td width="40"></td>
        <td colspan="4" width="280" valign="center" align="left">Select a
command for a couple:</td>
    </tr>
    <tr>
        <td width="40"></td>
        <td width="100"><input type="radio" checked name="CMDBUTTON"
value="DISPLAY">Display</td>
        <td width="100"><input type="radio" name="CMDBUTTON"
value="ADD">Add</td>
        <td width="40"></td>
        <td width="40"></td>
        <td width="40"></td>
        <td width="100"><input type="radio" name="CMDBUTTON"
value="DISPLAY_COUPLE">Display</td>
        <td width="100"><input type="radio" name="CMDBUTTON"
value="ADD_COUPLE">Add</td>
        <td width="40"></td>
    </tr>
    <tr>
        <td width="40"></td>
        <td width="100"><input type="radio" name="CMDBUTTON"
value="DELETE">Delete</td>
        <td width="100"><input type="radio" name="CMDBUTTON"
value="UPDATE">Update</td>
        <td width="40"></td>
        <td width="40"></td>
        <td width="40"></td>
        <td width="100"><input type="radio" name="CMDBUTTON"
value="DELETE_COUPLE">Delete</td>
        <td width="100"><input type="radio" name="CMDBUTTON"
value="UPDATE_COUPLE">Update</td>
        <td width="40"></td>
    </tr>

```



```

        </tr>
    <tr>
        <td colspan="4" width="280" valign="center" align="left">Select
the type of transaction managment:</td>
        <td width="40"></td>
        <td colspan="4" width="280" valign="center" align="center"></td>
    </tr>
    <tr>
        <td width="40"></td>
        <td width="100"><input type="radio" checked name="MANAGEMENT"
value="CONTAINER">Container</td>
        <td width="100"><input type="radio" name="MANAGEMENT"
value="COMPONENT">Component</td>
        <td width="40"></td>
        <td width="40"></td>
        <td width="40"></td>
        <td width="100"></td>
        <td width="100"></td>
        <td width="40"></td>
    </tr>
</table>
<p>
<input type="submit" name="Submit" id="Submit" value="Submit">

<input type="reset" name="Reset" id="Reset" value="Reset">

</p>
</form>
</body>
</html>

```

18. Press **Ctrl-S** to save the file and then close the CouplePhoneBookResultsForm.html file.

Part 3: Configuring the server and deploying the sample

To run a service in this sample, you must deploy the session bean to a server. In this case, the server runs in the WebSphere Unit Test Environment. This server must be configured and started. For this sample service, you need to create one server instance and server configuration. A server instance identifies the run-time environment that you want to use for testing your project resources. A server configuration contains information that is required to set up and publish to a server. After you configure the server, you will deploy the EAR project containing the service.

Note:

If you already created a server and server configuration, you still need to deploy the EAR file to the server.

Step 1: Configuring the server and deploying the EAR project

To create a server instance and configuration, complete the following steps:

1. In the Business Integration perspective, click the **Server Configuration** tab to open the Server Configuration view. Right-click anywhere in the Server Configuration view. Select **New > Server and Server Configuration**. The Create a New Server and Server Configuration wizard opens.
2. Type myIMSServicesServer for the server name. (The default folder name is Servers.)
3. Expand **WebSphere version 5.1** and select **Integration Test Environment**. Click **Next**.

4. If necessary, click **Yes** to create a server project named **Servers**.
5. The server port number defaults to 9080. The port identifies the location of the service. Click **Finish**. The new server instance appears in the Server Configuration view and in the Servers view.

You have just created an instance of the WebSphere Application Server that is emulated by the WebSphere Test Environment running on your local host on port 9080.

Next, you must add an instance of the JCA connection factory and configure its properties. The connection factory, as its name implies, provides connections to the EIS on demand. You specify all of the information needed by the resource adapter to connect to a particular instance of the EIS. For the IMS resource adapter, you must specify at least the `HostName`, `DataStoreName`, and `PortNumber` properties that determine which IMS to connect to. These values determine the IMS that will be accessed through all of the connections created by this instance of the connection factory. You also specify the JNDI lookup name under which the new connection factory instance will be available to components. The components can use this lookup name to quickly make a connection to the EIS. To add a connection factory, complete the following steps:

1. Click the **Server Configuration** tab to open the Server Configuration view. Expand **Servers**.
2. Double-click the server configuration, **myIMSServicesServer**. An editor opens.
3. Click the **J2C** tab. Click **Add** beside the J2C Resource Adapters table.
4. From the Resource Adapter Name list, select the resource adapter named **IMS**. Click **OK**.
5. In the J2C Resource Adapters table, select the **IMS resource adapter**, then click **Add** beside the J2C Connection Factories table. The application client will look up this connection factory instance using the JNDI interface. The application client will then use this connection factory instance to get a connection to the underlying IMS.
6. In the Create Connection Factory window, type the name `ims_cf`. Type the JNDI name `myIMSTarget`. Click **OK**.
7. In the Resource Properties table, type the property values appropriate for your environment. You might need to scroll down to see this table. See Connection properties for a description of these properties. For example:
 - In the **HostName** field, type `MYHOST.ABC.XYZ.COM`.
 - In the **DataStoreName** field, type `MYDSTOR`.
 - In the **PortNumber** field, type `9999`.
8. Press **Ctrl-S** to save the changes and then close the editor.

Next you need to add the EAR project (`myIMSPhoneBookServiceEAR`) to the server configuration that you created. To add the project, complete the following steps:

1. In the Server Configuration view under Servers, right-click **myIMSServicesServer**.
2. Select **Add and Remove Projects > myIMSPhoneBookServiceEAR**. `myIMSPhoneBookServiceEAR` is the name of the Enterprise Application project that you created earlier.
3. Click **Finish**.

You have now successfully generated an enterprise service from an IMS transaction and deployed that service to the WebSphere test environment.

Step 2: Running the sample

To run the sample, complete the following steps:

1. Click the **Servers** tab and check the status of the server instance. If the status for the server is stopped, then right-click the server instance and select **Start**. Wait until the server is started. The server is started when you see Started next to the server on the Servers tab.
2. In the Package Explorer view, expand **myIMSPhoneBookServiceWeb > WebContent**.
3. Right-click **CouplePhoneBookInputForm.html** and select **Run on Server**.
4. The URL is displayed in a web browser:
`http://localhost:9080/myIMSPhoneBookServiceWeb/
CouplePhoneBookInputForm.html`
5. Type data in the fields provided on the web page. For example, type the following information for the first individual:
 - Last Name: Test1
 - First Name: Test1
 - Extension: 1-1111
 - Zip code: 11111
6. Type the following information for the second individual:
 - Last Name: Test2
 - First Name: Test2
 - Extension: 2-2222
 - Zip code: 22222
7. Select the desired function to be performed. For example, select the **Add** radio button for a couple heading.
8. Select the type of transaction to be used. For example, select the **Container** radio button.
9. Click **Submit** to submit the data for processing. The transaction should run without exceptions. The results page appears with the message "Entry has been added." **Note:** Be careful when deleting an individual. Make sure that the individual is not part of a couple. If the individual is part of a couple, delete the entire couple.

Congratulations! You have now successfully built EJBs using both container-managed and component-managed EIS sign-on to run an IMS transaction.

Sample: Building input and output records using the CCI record helper class

Note: The class IMSCCIRecord is deprecated in IMS Connector for Java Version 9.1.0.1.1 and IMS Connector for Java Version 2.2.3. The functions provided by this class are now available in the development environments WebSphere Studio Application Developer Integration Edition and Rational Application Developer. For more information, see:

- “Building a Java application that uses the J2EE Connector Architecture Common Client Interface” on the IMS Examples Exchange at <http://www.ibm.com/software/data/ims/examples/exHome.html>
- “Using IMS data bindings in a CCI application” in the online help for the IMS resource adapter in Rational Application Developer

Objectives

The main objective of this sample is to describe how to build input and output records using the Common Client Interface (CCI) record helper class provided by the IMS resource adapter. This sample demonstrates how to:

- Extend the CCI record helper class, `IMSCCIRecord`
- Use the type-specific API and the field-specific API
- Create input and output records for your application

Time required

Allow 60 minutes. This will give you enough time to extend the CCI record helper class, create input and output records specific to your application, and run this sample.

Before you begin

This sample assumes that WebSphere Studio Application Developer Integrated Edition, Version 5.1 is installed on your machine and that you are familiar with using the product. It also assumes that the IMS resource adapter (also known as IMS Connector for Java) is correctly installed and configured on the WebSphere Application Server.

Description

To access IMS transactions through the IMS resource adapter, you can build applications with the tooling provided in WebSphere Studio Application Developer Integration Edition or by using the Common Client Interface (CCI). The CCI API provides access from J2EE clients, such as enterprise beans, JavaServer Pages (JSP) technology, and servlets to an enterprise information system (EIS), such as IMS. To use the Common Client Interface, an input byte array must be created with the values of the input fields; these fields are extracted from the output byte array, which is returned by IMS. To simplify this process, the IMS resource adapter provides a CCI record helper class that can be extended to handle the input and output byte arrays or records.

The `IMSCCIRecord` helper class provided by the IMS resource adapter contains two APIs, type-specific and field-specific. These APIs help create a byte array that is sent to IMS. This sample leads you through detailed steps that describe how to extend the CCI record helper class and build input and output records with both APIs to access IMS transactions. **Note:** Message Format Service (MFS) is not supported with the CCI Record Helper class.

This sample is based on the PhoneBook sample and creates simple input and output records. **Note:** For conversational transactions, you use the same procedures for creating the input and output records provided in this sample for creating the first iteration and the last iteration of a conversation; however, you must also create as many records necessary for your conversation for the middle iterations.

The steps in this sample are:

Part 1: Creating your records using field-specific API

1. Creating field-specific input and output records
2. Creating a Java application that uses the CCI API to access your records
3. Tailoring the field-specific input and output records to your transaction
4. Additional options for building input and output records

Part 2: Creating your records using the type-specific API

1. Creating type-specific input and output records
2. Creating a Java application that uses the CCI API to access your records
3. Tailoring the type-specific input and output records to your transaction
4. Additional options for building input and output records

Part 1: Creating your records using field-specific API

To create your records using the field-specific API, you must customize the setter and getter methods for each field. However, before you create the input and output records, you must create a Java project and Java package. The Java project stores all of the files needed by your application. To create a Java project, complete the following steps:

1. From the menu bar, select **File > New > Project** to create a Java project.
2. In the New Project page, select **Java** in the right-hand column, and then select **Java Project** in the left-hand column, and click **Next**.
3. In the New Java Project page, type `CCIPhoneBookField` in the Project Name field. Click **Next**.
4. Click the **Projects** tab and select the project that contains your `ims.rar` file to add it to your build path.
5. Click the **Libraries** tab and select **Add External JARs**.
6. Browse to locate the directory: `WS_install_dir\runtimes\ee_v51\lib`, where `WS_install_dir` is the directory where WebSphere Studio is installed, and select `j2ee.jar` file. Click **Finish**.

The new Java project is created. Now, you must create a Java package to hold the classes. To create a Java package, complete the following steps:

1. In the Package Explorer perspective, right-click the **CCIPhoneBookField** Java project that you just created and select **New > Package**.
2. In the Java Package page, type `CCI.field.sample` in the **Name** field. Ensure your source folder is **CCIPhoneBookField**. Click **Finish**. The Java package is created in the `CCIPhoneBookField` project.

After creating the Java project and Java package, you can create new input and output records.

Step 1: Creating field-specific input and output records

The `PhoneBookInputRecordField.java` and `PhoneBookOutputRecordField.java` classes extend the `IMSCCIRecord` helper class and build the input and output records. To create the field-specific input record class, complete the following steps:

1. Click the **Services** tab in the Business Integration perspective and expand **CCIPhoneBookField Java** project.

2. Right-click **CCI.field.sample** package and select **New > Class**.
3. In the New Java Class window, ensure the source folder is **CCIPhoneBookField** and the package name is **CCI.field.sample**.
4. In the Name field, type **PhoneBookInputRecordField.java**.
5. Uncheck all boxes under **Which method stubs would you like to create?** and click **Finish**.
6. An editor window opens. Replace all contents with the following Java code:

```
package test.CCIFieldSpecific;

import com.ibm.ims.base.*;
import test.CommonClient.*;
public class PhoneBookInputRecordField extends IMSCCIRecord{

    final static DLTypeInfo[] fieldInfo = {
        new DLTypeInfo("in__cmd",
            DLTypeInfo.CHAR, 1, 8),
        new DLTypeInfo("in__name1",
            DLTypeInfo.CHAR, 9, 10),
        new DLTypeInfo("in__name2",
            DLTypeInfo.CHAR, 19, 10),
        new DLTypeInfo("in__extn",
            DLTypeInfo.CHAR, 29, 10),
        new DLTypeInfo("in__zip",
            DLTypeInfo.CHAR, 39, 7)
    };

    public PhoneBookInputRecordField (String encoding)
    {

        super("phoneBookIn", fieldInfo, 45, encoding);
    }

    public short getIn__ll()
    {
        short result = getLL();
        return result;
    }

    public void setIn__ll(short in__ll)
    {
        setLL(in__ll);
    }

    public short getIn__zz()
    {
        short result = getZZ();
        return result;
    }

    public void setIn__zz(short in__zz)
    {
        setZZ(in__zz);
    }

    public String getIn__trcd()
    {
        String result = getTranCode();
        return result;
    }
}
```

```

public void setIn__trcd(String in__trcd)
{
    setTranCode(in__trcd);
}

public String getIn__command()
{
    return (String)basicGet("in__cmd");
}

public void setIn__command(String in__cmd)
{
    basicSet("in__cmd", in__cmd);
}

public String getIn__name1()
{
    return (String)basicGet("in__name1");
}

public void setIn__name1(String in__name1)
{
    basicSet("in__name1", in__name1);
}

public String getIn__name2()
{
    return (String)basicGet("in__name2");
}

public void setIn__name2(String in__name2)
{
    basicSet("in__name2", in__name2);
}

public String getIn__extn()
{
    return (String)basicGet("in__extn");
}

public void setIn__extn(String in__extn)
{
    this.basicSet("in__extn", in__extn);
}

public String getIn__zip()
{
    return (String)this.basicGet("in__zip");
}

public void setIn__zip(String in__zip)
{
    this.basicSet("in__zip", in__zip);
}
}

```

7. Press **Ctrl-S** to save the changes and then close the editor. The field-specific input class is created.

Similarly, you create the output record. To create the field-specific output record class, complete the following steps:

1. Click the **Services** tab in the Business Integration perspective and expand **CCIPhoneBookField Java** project.
2. Right-click **CCI.field.sample** package and select **New > Class**.

3. In the New Java Class window, ensure the source folder is **CCIPhoneBookField** and the package name is **CCI.field.sample**.
4. In the Name field, type **PhoneBookOutputRecordField.java**.
5. Uncheck all boxes under **Which method stubs would you like to create?** and click **Finish**.
6. An editor window opens. Replace all contents with the following Java code:

```
package test.CCISpecific;

import com.ibm.ims.base.*;
import test.CommonClient.*;

public class PhoneBookOutputRecordField extends IMSCCRecord {

    final static DLTypeInfo[] fieldInfo = {
        new DLTypeInfo("out_mesg",
            DLTypeInfo.CHAR, 1, 40),
        new DLTypeInfo("command",
            DLTypeInfo.CHAR, 41, 8),
        new DLTypeInfo("out_name1",
            DLTypeInfo.CHAR, 49, 10),
        new DLTypeInfo("out_name2",
            DLTypeInfo.CHAR, 59, 10),
        new DLTypeInfo("out_extn",
            DLTypeInfo.CHAR, 69, 10),
        new DLTypeInfo("out_zip",
            DLTypeInfo.CHAR, 79, 7),
        new DLTypeInfo("out_segno",
            DLTypeInfo.CHAR, 86, 4),
    };

    public PhoneBookOutputRecordField (String encoding) {
        super("phoneBookout", fieldInfo, 89, encoding);
    }

    public short getOut__ll()
    {
        short result = getLL();
        return result;
    }

    public void setOut__ll(short out__ll)
    {
        setLL(out__ll);
    }

    public short getOut__zz()
    {
        short result = getZZ();
        return result;
    }

    public void setOut__zz(short out__zz)
    {
        setZZ(out__zz);
    }

    public String getOut__name1()
    {
        return (String)this.basicGet("out__name1");
    }
}
```



```

public void setOut__name1(String out__name1)
{
    this.basicSet("out__name1", out__name1);
}

public String getOut__name2()
{
    return (String)this.basicGet("out__name2");
}

public void setOut__name2(String out__name2)
{
    this.basicSet("out__name2", out__name2);
}

public String getOut__extn()
{
    return (String)this.basicGet("out__extn");
}

public void setOut__extn(String out__extn)
{
    this.basicSet("out__extn", out__extn);
}

public String getOut__zip()
{
    return (String)this.basicGet("out__zip");
}

public void setOut__zip(String out__zip)
{
    this.basicSet("out__zip", out__zip);
}

public String getOut__segno()
{
    return (String)this.basicGet("out__zip");
}

public void setOut__segno(String out__zip)
{
    this.basicSet("out__zip", out__zip);
}

    public String getOut__mesg()
    {
        return (String)this.basicGet("out__mesg");
    }

public void setOut__mesg(String out__mesg)
{
    this.basicSet("out__mesg", out__mesg);
}
}

```

7. Press **Ctrl-S** to save the changes and then close the editor. The field-specific output class is created.

Step 2: Creating a Java application that uses the CCI API to access your records

In this step, you will create a Java application that uses the Java CCI API to invoke a transaction in IMS. The Java application calls the input and output classes that

you created in Step 1: Creating the field-specific input and output records. Note, instead of a Java application, you can also create an EJB or other J2EE clients to use the CCI API.

To create the Java application, complete the following steps:

1. Click the **Services** tab in the Business Integration perspective and expand **CCIPhoneBookField Java** project.
2. Right-click **CCI.field.sample** package and select **New > Class**.
3. In the New Java Class window, ensure the source folder is **CCIPhoneBookField** and the package name is **CCI.field.sample**.
4. In the Name field, type **PhoneBookCCIField.java**.
5. Uncheck all boxes under **Which method stubs would you like to create?** and click **Finish**.
6. An editor window opens. Replace all contents with the following Java code:

```
package test.CCISpecific;

//import test.CommonClient.*;
import javax.resource.cci.*;
import javax.naming.*;
import com.ibm.connector2.ims.ico.*;

public class PhoneBookCCIField {

    private boolean isManaged = false;

    public PhoneBookCCIField(booleanmanagedFlag) {
        this.isManaged = managedFlag;
    }

    public void execute() {
        try {

            ConnectionFactory cf = null;

            if (this.isManaged) {
                Context ic = newInitialContext();
                cf =(ConnectionFactory) ic.
                    lookup("MyIMS");
            } else {
                //Create and set values for a managed connection
                factory for ECI IMSManagedConnectionFactory
                mcf = new IMSManagedConnectionFactory();
                mcf.setDataStoreName("IMS1");
                mcf.setHostName("csdmec06.svl.ibm.com");
                mcf.setPortNumber(newInteger(9999));

                //Create a connectionfactory
                cf = (IMSConnectionFactory)mcf.createConnectionFactory();
            }

            Connection connection = cf.getConnection();

            //Create an interaction with IMS to start IMSPROG program
            IMSInteraction interaction = (IMSInteraction)
                connection.createInteraction();
            IMSInteractionSpec ixnSpec = new IMSInteractionSpec();
            ixnSpec.setInteractionVerb(IMSInteractionSpec.SYNC_SEND_RECEIVE);

            //Create a new record for IMS PhoneBookInputRecordField
            input = new PhoneBookInputRecordField("cp037");
            input.setIn__11((short)59);
```

```

        input.setIn__zz((short) 0);
        input.setIn__trcd("IVTNO");
        input.setTranCodeLength(10);
        input.setIn__command("DISPLAY");
        input.setIn__name1("LAST3");
        input.setIn__name2("");
        input.setAllFieldsGiven(false);

        PhoneBookOutputRecordField
output = new PhoneBookOutputRecordField("cp037");
        interaction.execute(ixnSpec, input, output);

        System.out.println ("Output is: ");
        System.out.println("\nMessage: "

                                + output.getOut__mesg()
                                + "\nName: "
                                + output.getOut__name1()
                                + " "
                                + output.getOut__name2()
                                + "\nExtension: "
                                + output.getOut__extn()
                                + "\nZipcode: "
                                + output.getOut__zip()
                                );

        //Close both the interaction and the connection
        interaction.close();
        connection.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public static void main(String[] args) {
    // When running in an unmanaged environment
    (i.e. standalone java app)
    PhoneBookCCIField pb =
    new PhoneBookCCIField(false);

    // When running in a managed environment
    (i.e. J2EE server)
    // PhoneBookCCI pb = new PhoneBookCCI(true);

    pb.execute();
}
}

```

7. Press **Ctrl-S** to save the changes and then close the editor. The field-specific java application is created.

Step 3: Tailoring the field-specific input and output records to your transaction

The input class, output class, and the Java application created in Step 1 and 2 are specific to the CCI PhoneBook sample. To run your transactions, complete the following steps to tailor the input and output records created in Step 2. After changing the records, create a new Java project and package. Ensure the names of the project and package are consistent with the application you want to run, and copy the modified records to the new package.

1. Change the input class

- a. Replace the DLTypeInfo array to match the fields of your transaction. Each array element has the following format: name, type, offset, and length. Ensure you provide information for each field. For example:

```

final static DLTypeInfo[] fieldInfo = {

    // new DLTypeInfo("in_cmd", DLTypeInfo.CHAR, 1, 8),
    new DLTypeInfo("in_yourField1",DLTypeInfo.FieldType,
        offset, length),
    ....
    ....
    new DLTypeInfo("in_yourField4",DLTypeInfo.FieldType,
        offset(0), length(L))

};

```

- b. Change the class name. For example:

```
public class yourApplication extends IMSCCRecord
```

Note: Ensure the Java file matches the name in your new package.

- c. Change the constructor. For example:

```

public yourApplication (String encoding) {

    super("yourApplication", fieldInfo, total_length_of_record, encoding);
}

```

Note: The `total_length_of_record` is equivalent to `O +L-1`.

- d. Replace the setter and getter methods to match the fields of your transaction. We recommend that you follow the `set_FieldName()` format. For example:

```

public void setIn__fieldName(String in__ fieldName)
{
    this.basicSet("in__ fieldName ", in__ fieldName);
}

    public String getIn__fieldName()
    {
        return (String)this.basicGet("in__ fieldName ");
    }

```

Use the **basicSet("fieldname", value)** method to set the value of all the fields. Use the **basicGet("fieldname")** method to get the value of any field. The value returned is of type Object and therefore must be typecasted to the appropriate type before being returned.

2. Change the output class

- Repeat the same steps for changing the input class and apply the same changes to the methods in the output record.
- It is recommended that you use "in_" and "out_" as a prefix to the field names to differentiate between input and output fields.

3. Change the CCI API class

- a. Set the hostname, portnum, and datastore name. For example:

```

//Create and set values for a managed connection factory for ECI
IMSMangedConnectionFactory mcf = new IMSManagedConnectionFactory();
mcf.setDataStoreName("yourDatastore");
mcf.setHostName("yourHostName");
mcf.setPortNumber(new Integer(yourPortNumber));

//Create a connection factory
cf = (IMSConnectionFactory) mcf.createConnectionFactory();

}

```

- b. Create an instance of your input and output record class and call the setter methods for the fields in your transaction. The following methods are common to most applications (input records generally contain LL, ZZ & trancode fields). If your application does not use these fields, set

setAllFieldsGiven() to true and do not call the methods below. Refer to *Step 4: Additional options for building input and output records* for a detailed discussion on the options supported. For example:

```
//Create a new record for IMS
yourApplication input = new yourApplication("cp037");
    input.setIn_ll((short) 59);
    input.setIn_zz((short) 0);
    input.setIn_trcd("IVTN0");
    input.setTranCodeLength(10);
    input.setAllFieldsGiven(false);
```

- c. Use the setter methods created in **Step 1: Changing the input class**, and set the transaction fields with appropriate values. For example:

```
input.setIn__fieldName1(value1);
input.setIn__fieldName2(value1);
.....
.....
```

- d. Create an instance of the output record and use the getter methods to get the output values.

```
yourApplication output = new yourApplication("cp037");

output.getOut__fieldName()
```

Step 4: Additional options for building input and output records

There are a couple of other options for building the input and output records. These options include:

- Modifying the setAllFieldsGiven() method to False.

If you modify the setAllFieldsGiven() to False, the IMSCCIRecord helper class assumes that the input record format is:

LL	ZZ	Trancode	Data
----	----	----------	------

The IMSCCIRecord helper class will use the values provided through the setter method to create a record in that format. When setAllFieldsGiven() is set to false, only specify the Data fields of the input message in the DLTypeInfo array. Do not add LL, ZZ, and trancode to the DLTypeInfo array because it will create duplicate LL, ZZ fields in the final message.

In addition, if the LL value is not provided, the IMSCCIRecord helper class will automatically compute and set it. The IMSCCIRecord helper class will also set ZZ to 0 if no value is provided for ZZ. The trancode is set only if the trancode and trancodeLength values are provided. **Note:** If setTranCode() is called, then setTranCodeLength(Int) method must also be called.

If you set the setAllFieldsGiven() to False, the IMSCCIRecord helper class assumes that the output record format is:

LL	ZZ	Data
----	----	------

Similarly, LL and ZZ should not be specified in the DLTypeInfo array.

- Modifying the setAllFieldsGiven() method to True.

This option enables you to create records that do not conform to the LL, ZZ, Trancode, Data format. If you set the setAllFieldsGiven() method to True, the record is built based on the values in the DLTypeInfo array of the input and output record. Therefore, the methods listed below cannot be used because LL, ZZ, and trancode are not elements in the DLTypeInfo array.

```
- input.setIn_ll((short) 59);
```

- `input.setIn_zz((short) 0);`
- `input.setIn_trcd("IVTNO");`
- `input.setTranCodeLength(10);`

The only method allowed with this option is: `input.set_fieldName(value);` where `fieldname` is an element of the `DLTypeInfo` array..

Part 2: Creating your records using the type-specific API

Use the type-specific API to keep your input and output records small and simple. You will use the setter and getter methods that are implemented by the `IMSCCIRecord`. The setter method format is `setTypeNames("fieldname", value);` for example, `setString("lastName", "LAST1")`. The getter method format is `getTypeNames("fieldname");` for example, `getString("lastName")`. Before you create the input and output records, you must create a Java project and Java package. The Java project stores all of the files needed by your application. To create a Java project, complete the following steps:

1. From menu bar, select **File > New > Project** to create a Java project.
2. In the New Project page, select **Java** in the right-hand column, and then select **Java Project** in the left-hand column, and click **Next**
3. In the New Java Project page, type `CCIPhoneBookType` in the Project name field. Click **Next**.
4. Click the **Projects** tab and select the project that contains your `ims.rar` file to add it to your build path.
5. Click the **Libraries** tab and select **Add External JARs**.
6. Browse to locate the directory: `WS_install\dir\runtimes\ee_v5\lib`, where `WS_install\dir` is the directory where WebSphere Studio is installed, and select `j2ee.jar` file. Click **Finish**.

The new Java project is created. Now, you must create a Java package to hold the classes. To create a Java package, complete the following steps:

1. In the Package Explorer perspective, right-click the **CCIPhoneBookType** Java project that you just created and select **New > Package**.
2. In the Java Package page, type `CCI.type.sample` in the **Name** field. Ensure your source folder is **CCIPhoneBookType**. Click **Finish**. The Java package is created in the `CCIPhoneBookType` project.

After creating the Java project and Java package, you can create a new input and output records.

Step 1: Creating type-specific input and output records

The `PhoneBookInputRecordType.java` and `PhoneBookOutputRecordType.java` classes extends the `IMSCCIRecord` helper class and builds the input and output records. To create the type-specific input record class, complete the following steps:

1. Click the **Services** tab in the Business Integration perspective and expand **CCIPhoneBookType Java** project.
2. Right-click **CCI.type.sample** package and select **New > Class**.
3. In the New Java Class window, ensure the source folder is **CCIPhoneBookType** and the package name is **CCI.type.sample**.
4. In the Name field, type `PhoneBookInputRecordType.java`.
5. Uncheck all boxes under **Which method stubs would you like to create?** and click **Finish**.

6. An editor window opens. Replace all contents with the following Java code:

```
package test.CCISpecific;

import com.ibm.ims.base.*;
import test.CommonClient.*;

public class PhoneBookInputRecordType extends IMSCCRecord {

    final static DLTypeInfo[] fieldInfo = {
        new DLTypeInfo("in_cmd",
            DLTypeInfo.CHAR, 1, 8),
        new DLTypeInfo("in_name1",
            DLTypeInfo.CHAR, 9, 10),
        new DLTypeInfo("in_name2",
            DLTypeInfo.CHAR, 19, 10),
        new DLTypeInfo("in_extn",
            DLTypeInfo.CHAR, 29, 10),
        new DLTypeInfo("in_zip",
            DLTypeInfo.CHAR, 39, 7)
    };

    public PhoneBookInputRecordType (String encoding)
    {

        super("phoneBookIn", fieldInfo, 45 ,encoding);

    }
}
```

7. Press **Ctrl-S** to save the changes and then close the editor. The type-specific input class is created.

Similarly, you can create the output record. To create the type-specific output record class, complete the following steps:

1. Click the **Services** tab in the Business Integration perspective and expand **CCIPhoneBookType Java** project.
2. Right-click **CCI.type.sample** package and select **New > Class**.
3. In the New Java Class window, ensure the source folder is **CCIPhoneBookType** and the package name is **CCI.type.sample**.
4. In the Name field, type **PhoneBookOutputRecordType.java**.
5. Uncheck all boxes under **Which method stubs would you like to create?** and click **Finish**.
6. An editor window opens. Replace all contents with the following Java code:

```
package test.CCISpecific;

import com.ibm.ims.base.*;
import test.CommonClient.*;

public class PhoneBookOutputRecordType extends IMSCCRecord {

    final static DLTypeInfo[] fieldInfo = {
        new DLTypeInfo("out_mesg",
            DLTypeInfo.CHAR, 1, 40),
        new DLTypeInfo("command",
            DLTypeInfo.CHAR, 41, 8),
        new DLTypeInfo("out_name1",
            DLTypeInfo.CHAR, 49, 10),
        new DLTypeInfo("out_name2",
            DLTypeInfo.CHAR, 59, 10),
    };
}
```

```

        new DLTypeInfo("out__extn",
            DLTypeInfo.CHAR, 69, 10),
        new DLTypeInfo("out__zip",
            DLTypeInfo.CHAR, 79, 7),
        new DLTypeInfo("out__segno",
            DLTypeInfo.CHAR, 86, 4),
    };

    public PhoneBookOutputRecordType (String encoding)
    {
        super("phoneBookout", fieldInfo, 89, encoding);
    }
}

```

7. Press **Ctrl-S** to save the changes and then close the editor. The type-specific output class is created.

Step 2: Creating a Java application that uses the CCI API to access your records

In this step, you will create a Java application to that uses the Java CCI API to invoke a transaction in IMS. The Java application calls the input and output classes that you created in Step 1. In addition to a Java application, you can also create an EJB or other J2EE clients to use the CCI API.

To create the Java application, complete the following steps:

1. Click the **Services** tab in the Business Integration perspective and expand **CCIPhoneBookType** Java project.
2. Right-click **CCI.type.sample** package and select **New > Class**.
3. In the New Java Class window, ensure the source folder is **CCIPhoneBookType** and the package name is **CCI.type.sample**.
4. In the Name field, type **PhoneBookCCIType.java**.
5. Uncheck all boxes under **Which method stubs would you like to create?** and click **Finish**.
6. An editor window opens. Replace all contents with the following Java code:

```

package test.CCITypeSpecific;

//import test.CommonClient.*;
import javax.resource.cci.*;
import javax.naming.*;
import com.ibm.connector2.ims.ico.*;

public class PhoneBookCCIType {

    private boolean isManaged = false;

    public PhoneBookCCIType(boolean managedFlag) {
        this.isManaged = managedFlag;
    }

    public void execute() {
        try {

            ConnectionFactory cf = null;

            if (this.isManaged) {
                Context ic = new InitialContext();
                cf = (ConnectionFactory)

```



```

        ic.lookup
        ("MyIMS");

    } else {
        //Create and set values for a
        managed connection factory
        for ECI
        IMSManagedConnectionFactory mcf =
        new IMSManagedConnectionFactory();

mcf.setDataStoreName("IMS1");

mcf.setHostName("csdmec06.svl.ibm.com");
        mcf.setPortNumber(new Integer(9999));

        //Create a connection factory
        cf = (IMSConnectionFactory)
        mcf.createConnectionFactory();
    }

    Connection connection = cf.getConnection();

    //Create an interaction with IMS to start
    IMSPROG program
    IMSInteraction interaction = (IMSInteraction)
    connection.createInteraction();
    IMSInteractionSpec ixnSpec = new
    IMSInteractionSpec();

    ixnSpec.setInteractionVerb(IMSInteractionSpec.SYNC_SEND_RECEIVE);

        //Create a new input record for IMS
    PhoneBookInputRecordType input = new
        PhoneBookInputRecordType("cp037");

        input.setLL((short) 59);
        input.setZZ((short) 0);

        input.setTranCode("IVTNO");
    input.setTranCodeLength(10);
    input.setString("in__cmd","DISPLAY");
    input.setString("in__name1","LAST3");
    input.setString("in__name2","FIRST3");
    input.setAllFieldsGiven(false);

        PhoneBookOutputRecordType output = new
        PhoneBookOutputRecordType("cp037");

    interaction.execute(ixnSpec, input, output);

        System.out.println ("Output is: ");
        System.out.println("\nMessage: "
            + output.getString("out__mesg")
            + "\nName:"
            + output.getString("out__name1")
            + " "
            + output.getString("out__name2")
            + "\nExtension: "
            + output.getString("out__extn")
            + "\nZipcode: "
            + output.getString("out__zip")
        );

        //Close both the interaction and the connection
    interaction.close();

```

```

        connection.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public static void main(String[] args) {
    // When running in an unmanaged
    // environment (i.e.
    // standalone java app)
    PhoneBookCCIType pb =
    new PhoneBookCCIType(false);

    // When running in a managed
    // environment (i.e.
    // J2EE server)
    // PhoneBookCCI pb = new PhoneBookCCI(true);

    pb.execute();
}
}

```

7. Press **Ctrl-S** to save the changes and then close the editor. The type-specific CCI API class is created.

Step 3: Tailoring the type-specific input and output records to your transaction

The input class, output class, and the Java application created in Step 1 and 2 are specific to the CCI PhoneBook sample. To run your transactions, complete the following steps to tailor the input and output records created in Step 2. After changing the records, create a new Java project and package. Ensure the names of the project and package are consistent with the application you want to run, and copy the modified records to the new package.

1. Change the input class

- a. Replace the `DLTypeInfo` array to match the fields of your transaction. The array element has the following format: name, type, offset, and length. Ensure you provide information for each field. For example:

```

final static DLTypeInfo[] fieldInfo = {
    // new DLTypeInfo("in_cmd", DLTypeInfo.CHAR, 1, 8),
    new DLTypeInfo("in_yourField1",DLTypeInfo.FieldType,
    offset, length),
    ....
    ....
    new DLTypeInfo("in_yourField4",DLTypeInfo.FieldType,
    offset(0), length(L))
};

```

- b. Change the class name. For example:

```
public class yourApplication extends IMSCCIRecord
```

Note:Ensure the Java file matches the name in your new package.

- c. Change the constructor. For example:

```

public yourApplication (String encoding) {

    super("yourAplication", fieldInfo, total_length_of_record, encoding);
}

```

Note: The `total_length_of_record` is equivalent to `O + L - 1`.

2. Change the Java application calling the CCI API

- a. Set the hostname, portnum, and datastore name. For example:

```
//Create and set values for a managed connection factory for ECI
IMSMangedConnectionFactory mcf = new IMSMangedConnectionFactory();
mcf.setDataStoreName("yourDatastore");
mcf.setHostName("yourHostName");
mcf.setPortNumber(new Integer(yourPortNumber));

//Create a connection factory
cf = (IMSConnectionFactory) mcf.createConnectionFactory();
}
```

- b. Create an instance of your input and output record class and call the setter methods for the fields in your transaction. The following methods are common to most applications (input records generally contain LL, ZZ & trancode fields). If your application does not use these fields, set `setAllFieldsGiven()` to true and do not call the methods below. Refer to *Step 4: Additional options for building input and output records* for a detailed discussion on the options supported. For example:

```
//Create a new record for IMS
yourApplication input = new yourApplication("cp037");

input.setLL((short) value);
input.setZZ((short) value);
input.setTranCode(" XXXXX ");
input.setTranCodeLength(value);
```

- c. Use the setter and getter methods provided by `IMSCCIRecord` class and set your transaction fields with appropriate values. For example:

```
input.setfieldtype("field_Name", value);
yourApplication output = new yourApplication("cp037");
output.getFieldType("field_Name")
```

Step 4: Additional options for building input and output records

There are a couple of other options for building the input and output records. These options include:

- Modifying the `setAllFieldsGiven()` method to False.

If you modify the `setAllFieldsGiven()` to False, the `IMSCCIRecord` helper class assumes that the input record format is:

LL	ZZ	Trancode	Data
----	----	----------	------

The `IMSCCIRecord` helper class will use the values provided through the setter method to create a record in that format. When `setAllFieldsGiven()` is set to false, only specify the Data fields of the input message in the `DLTypeInfo` array. Do not add LL, ZZ, and trancode to the `DLTypeInfo` array because it will create duplicate LL, ZZ fields in the final message.

In addition, if the LL value is not provided, the `IMSCCIRecord` helper class will automatically compute and set it. The `IMSCCIRecord` helper class will also set ZZ to 0 if no value is provided for ZZ. The trancode is set only if the trancode and trancodeLength values are provided. **Note:** If `setTranCode()` is called, then `setTranCodeLength(Int)` method must also be called.

If you set the `setAllFieldsGiven()` to False, the `IMSCCIRecord` helper class assumes that the output record format is:

LL	ZZ	Data
----	----	------

Similarly, LL and ZZ should not be specified in the `DLTypeInfo` array.

- Modifying the `setAllFieldsGiven()` method to True.

This option enables you to create records that do not conform to the LL, ZZ, Trancode, Data format. If you set the `setAllFieldsGiven()` method to `True`, the record is built based on the values in the `DLTypeInfo` array of the input and output record. Therefore, the methods listed below cannot be used because LL, ZZ, and trancode are not elements in the `DLTypeInfo` array.

- `input.setLL((short) value);`
- `input.setZZ((short) value);`
- `input.setTranCode("XXXXX");`
- `input.setTranCodeLength(value);`

The only method allowed with this option is:

`input.setTypeNames("name",value);` where `fieldname` is an element of the `DLTypeInfo` array.

Sample: Creating an Enterprise Java Bean to communicate with a conversational IMS application

Objectives

Note: The class `IMSConversationalHelper` is deprecated in IMS Connector for Java Version 9.1.0.1.1 and IMS Connector for Java Version 2.2.3. In the future this will only be available as a sample class. It is recommended that you obtain “Using an EJB and generated helper classes to run a conversational IMS transaction” from the IMS Examples Exchange at <http://www.ibm.com/software/data/ims/examples/exHome.html>. Additional versions, using other types of input and output messages, will be made available over time.

The main objective of this sample is to show how to create an EJB to communicate with a conversational IMS application. In addition, this sample will show how to use the EJB in a Java application that communicates with a conversational IMS application. This sample, which is based on the conversational PhoneBook IMS Installation Verification Program, has five main parts.

- *Part 1: Building input and output records using the CCI record helper class* describes how to extend the `IMSCCIRecord` helper class, which is provided by the IMS resource adapter, and shows how to create classes for the input and output records of the messages of the iterations of the conversational IMS application.
- *Part 2: Importing the conversational record classes* shows how to import the input and output record classes created in this sample. These classes were created using the process shown in Part One.
- *Part 3: Creating a conversational EJB* describes the step-by-step tasks for extending the `IMSConversationalHelper` provided by the IMS resource adapter. The helper class is used to help create an EJB that communicates with the conversational PhoneBook IVP application.
- *Part 4: Building a conversational application* describes how the EJB created in Part Three can be used in a simple Web application.
- *Part 5: Configuring the server and running the sample application* shows how to configure your server so that you can deploy and run your sample.

Time required

Allow 90 minutes. This will give you enough time to import the input and output record classes, build the stateful session bean (the conversational EJB), and run the PhoneBook sample.

Before you begin

This sample assumes that WebSphere Studio Application Developer Integrated Edition, Version 5.1 is installed on your machine and that you are familiar with using the product. It also assumes that:

- the conversational IMS verification procedure (IVTCB) is installed on your IMS system.
- your environment meets the prerequisites for using the IMS resource adapter.
- the IMS resource adapter, **ims.rar**, has been imported into the WebSphere Studio workspace you use to develop your conversational application.
- the name of the IMS resource adapter is IMS.

Description

This sample leads you through detailed steps that describe how to build a stateful session bean to enable a conversation between a client application and an IMS conversational application program. Through a stateful session bean, the client application and the IMS application can exchange messages, representing iterations of the conversation, back and forth for an extended period of time. The conversation does not end after one message as it typically does for a non-conversational application program. Instead, the stateful session bean stores the state of the conversation for the client and allows the client application to continue communicating with the IMS application.

The client application uses the IMS resource adapter to interact with IMS through the host product, IMS Connect. To establish a conversation between the client application and the IMS application, you can build a stateful session bean. The EJB used by this sample uses the Common Client Interface (CCI) as well as conversational and CCI record helper classes to communicate with the IMS resource adapter. The CCI record helper classes are used to assist the client application to prepare the input and output messages from the iterations of a conversation. Because the stateful session bean uses IMS Connect to interact with the IMS application, you must use the same connection (for example, a persistent socket) for the duration of the conversation. IMS Connect uses the persistent socket to map to the IMS application.

Within WebSphere Studio, you will use wizards to help you generate the sample, and then deploy the sample to the WebSphere test environment that is part of WebSphere Studio. You will also generate a session bean to access the conversational IMS application program, and you will add EJB methods to remote interfaces to be used by client applications. You will also customize CCI and conversational helper classes to help your application prepare the input and output messages from the iterations of a conversation. For this sample, you run all of the server and client applications on the same machine.

The steps in this sample are:

Part 1: Building input and output records using the CCI record helper class

Part 2: Importing the conversational record classes

1. Creating a Java project
2. Creating a Java package
3. Importing the record classes

Part 3: Creating a conversational EJB

1. Creating an Enterprise Application project
2. Creating the EJB session bean
3. Creating the EJB business methods
4. Adding the business methods to the remote interface of the EJB
5. Setting the transaction attributes for the EJB session bean
6. Adding the record classes as a project utility JAR and adding it to the JAR dependency list
7. Generating the deployed code

Part 4: Building a conversational application

1. Generating access beans
2. Importing the conversational web pages

Part 5: Configuring the server and running the sample application

1. Configuring the server and deploying the EAR project
2. Running the sample

Part 1: Building input and output records using the CCI record helper class

The CCI record helper class, `IMSCCIRecord`, is a class that incorporates routines for data conversion to and from a byte array. `IMSCCIRecord` is extended to create classes that represent the input and output messages of an IMS transaction. The classes that extend `IMSCCIRecord` define the structure of the message, the data types associated with each of the fields, as well as the length and offset of each field. In addition, these record classes also provide setter and getter methods for the fields in the records, as well as the creation of a byte array that can be sent to the IMS resource adapter.

To create an EJB for a conversational IMS application, you need to extend the CCI record helper class, `IMSCConversationalHelper`, provided by the IMS resource adapter, to create the input and output records of your application. The customized CCI record helper class must contain a `DLTypeInfo` array to map `DLTypeInfo` fields to the COBOL copybook input record data fields.

To build the input and output records for a conversational sample, see the sample, *Building input and output records using the CCI record helper class* for further information.

Part 2: Importing the conversational record classes

To create an EJB for a conversational IMS application, you must have classes to represent the input and output messages (records) of each iteration of the conversation. Part 1 describes how to build the record classes. This sample also provides already-built record classes. To import the conversational record classes provided by this conversational PhoneBook sample, you must first create a Java project and then import the record classes into your Java package.

Step 1: Creating a Java project

The Java project stores all of the files for your Java project, including imported source files and files generated by wizards. You will use the New Project wizard to create your Java project. To create a Java project, complete the following steps:

1. From menu bar, select **File > New > Project** to create a Java project.
2. In the New Project page, select **Java** in the right-hand column, and then select **Java Project** in the left-hand column, and click **Next**
3. In the New Java Project page, type `MyConversationalPhoneBookCCIRRecordClasses` in the Project Name field. Click **Next**.
4. Click the **Projects** tab and select the project into which you imported the `ims.rar` file and add it to the build path of the `MyConversationalPhoneBookCCIRRecordClasses` project.
5. Click the **Libraries** tab and select **Add External JARs**.
6. Browse to locate the directory: `WS_install_dir\runtimes\ee_v51\lib`, where `WS_install_dir` is the directory where WebSphere Studio is installed, and select the `j2ee.jar` file and click **Open**.
7. Click **Finish**.

The new Java project is created.

Step 2: Creating a Java package

Before importing the record helper classes, create a Java package to hold the classes. To create a Java package, complete the following steps:

1. In the Package Explorer perspective, right-click the **MyConversationalPhoneBookCCIRRecordClasses** Java project that you created and select **New > Package**.
2. In the Java Package page, type `sample.ims.conv.record` in the **Name** field. Ensure your source folder is **MyConversationalPhoneBookCCIRRecordClasses**. Click **Finish**. The Java package is created in the `MyConversationalPhoneBookCCIRRecordClasses` project.

Step 3: Importing the conversational record classes

In this step you import the conversational record classes, *CPBInputRecord.java*, *CPBMiddleInputRecord.java*, and *CPBOutputRecord.java*, which are needed to create your conversational EJB. The conversational record classes define the structure of the input and output messages of the PhoneBook sample. To import the record classes, complete the following steps:

1. Expand **MyConversationalPhoneBookCCIRRecordClasses** project, right-click **sample.ims.conv.record** and select **Import**.
2. Select **File System** to import the java classes from the file system and click **Next**.
3. Click **Browse** beside the directory field to locate the following directory:
`WS_install_dir\wstools\eclipse\plugins\com.ibm.etools.ctc.samples.ims_5.1.0\sampleparts`

where `WS_install_dir` is the directory where WebSphere Studio is installed.

4. Select the following files:
 - `CPBInputRecord.java`

- CPBMiddleInputRecord.java
- CPBOutputRecord.java

5. Click **Finish**.

The **sample.ims.conv.record** package contains the CCI record classes for the conversational PhoneBook IVP. These classes are used by the conversational EJB.

Part 3: Creating a conversational EJB

Using the IMSConversationalHelper helper class which is provided by the IMS resource adapter, you can create a simple conversational EJB for this PhoneBook sample. Before you generate the EJB, you must first create an Enterprise Application project to contain parts of your conversational application. The Enterprise Application project contains an EJB project for the conversational EJB and a Web project for JSP pages. After you create an Enterprise Application project, you need to place the projects containing the IMS resource adapter and the conversational record classes on the build path of the EJB and Web project. Once the resource adapter and record classes are in your build path, create a stateful session bean that contains the business methods which access the conversational IMS application in the EJB project.

After the new EJB has been created, you modify the EJB deployment descriptor and generate deployment code for the EJB.

The steps in this part are:

Step 1: Creating an Enterprise Application project

Step 2: Creating the EJB session bean

Step 3: Creating the EJB business methods

Step 4: Adding the business methods to the remote interface

Step 5: Setting the transaction attributes for the EJB session bean

Step 6: Adding the record classes to the project utility JARS and the JAR dependency list

Step 7: Generating the deployed code

Step 8: Generating the access bean

Step 1: Creating an Enterprise Application project

The Enterprise Application project is used to package together EJBs, Web projects, or client projects into an EAR file that can be deployed on an application server. In this sample, the Enterprise Application project consists of an EJB project containing an EJB module and a Web project containing a web module.

1. From the menu bar, select **File > New > Project**.
2. In the New Project window, select **J2EE** in the left-hand column and **Enterprise Application Project** in the right-hand column. Click **Next** and the Enterprise Application Project Creation wizard opens.
3. In the J2EE Specification version page, select **Create J2EE 1.3 Enterprise Application project** and click **Next**.

4. In the Enterprise Application Project page, type in the **Enterprise application project name** field, `MyConversationalPhoneBook` and click **Next**. The **New Enterprise Application Project** window opens.
5. In the **New Enterprise Application Project** window, click **New Module** and ensure **Create default module projects** is selected. Also, ensure that the **EJB Project** and **Web Project** are selected and the other options are deselected. Click **Finish** to close and save the settings in the **New Enterprise Application Project** window.
6. Click **Finish**.

The EJB and Web modules are created and stored in the Enterprise Application project.

Step 2: Creating the EJB session bean

In this step, you will create an EJB session bean. The session bean will have five business methods. The business methods provide the business logic of the conversational IMS application.

Before you create the conversational EJB session bean, you must first add the project into which you imported the `ims.rar` file and the conversational record classes to the build path of the EJB project. To add the IMS project, complete the following steps:

1. In the Business Integration Perspective, click the **J2EE Hierarchy** tab.
2. Expand **EJB Modules** and right-click `MyConversationalPhoneBookEJB` and select **Properties**.
3. The Properties for `MyConversationalPhoneBookEJB` window opens. In the left-hand column, select **Java Build Path** and then select the **Projects** tab.
4. In **Required projects on the build path** box, select the **IMS** project and `MyConversationalPhoneBookCCIRRecordClasses` Java project and click **OK**.

After you have updated your build path, you can create the conversational session bean. To create a conversational EJB session bean, complete the following steps:

1. In the Business Integration Perspective, click the **J2EE Hierarchy** tab.
2. Expand **EJB Modules** and right-click on the `MyConversationalPhoneBookEJB` project. Select **New > Enterprise Bean**.
3. The Enterprise Bean Creation wizard opens. Ensure that the EJB project name is `MyConversationalPhoneBookEJB` and click **Next**.
4. In the Create a 2.0 Enterprise Bean window:
 - Ensure that **Session bean** is selected.
 - In the **Bean name** field, type `MyConversationalPhoneBook`
 - In the **Default package** field, type `sample.ims.conv`. Click **Next**.
5. In the Enterprise Bean Details window:
 - For the **Session type**, select **stateful**.
 - For the **Transaction type**, select **Container**.
 - Accept all other defaults, click **Next**.
6. In the EJB Java Class Details window, specify the `IMSConversationalHelper` as the superclass to the bean. To specify the `IMSConversationalHelper` as the superclass, click **Browse** next to the **Bean superclass** field.

7. The Type Selection window opens. In the **Select a class using** field, type `IMSConversationalHelper` to bring up the matching type. Select the **IMSConversationalHelper** class and click **OK**. Click **Finish**.

The conversational EJB session bean is created and generated in the `sample.ims.conv` package under the **MyConversationalPhoneBookEJB** project.

Step 3: Creating the EJB business methods

In this section, you will create a total of five EJB business methods. The first business method runs the initial iteration, the second business method continues the conversation. The third business method checks the conversational status, the fourth business method ends the conversation, and the fifth business method sets the `IMSConnectionSpec` property values. The business methods that you will create for the conversational EJB bean are:

runFirstIteration()

This method is called to begin a conversation with the conversational PhoneBook IVP application and is used only once per conversation.

runMiddleIteration()

This method can be called more than once to continue the conversation with the conversational PhoneBook IVP application.

isConvEnded()

This method is called after each iteration of the conversation to verify if the EJB is still in conversation with the IMS application. For some conversational applications, for example this conversational PhoneBook IVP application, the conversation can be terminated by the IMS application. The conversation is terminated with the END command input message.

endConversation()

This method is called to force the end of the conversation with the IMS application from the client side. This method causes the IMS resource adapter to send a message to IMS through IMS Connect. The method is similar to entering an /EXIT command from an IMS terminal. When the IMS resource adapter sends this type of message to IMS Connect it is a SEND ONLY interaction and the outcome of the interaction cannot be determined by the application client.

setComponentAuthorization()

This method is used to provide username, password, and groupname values if component-managed sign-on is used by an application component.

Note: In this sample, if you begin another conversation before you end the previous conversation, the conversational EJB will throw an exception indicating that the previous conversation has ended and the new conversation will not start. To create the EJB business methods for the conversational EJB bean, complete the following steps:

1. In the J2EE Hierarchy tab, expand **EJB Modules > MyConversationalPhoneBookEJB > MyConversationalPhoneBook**.
2. Double-click **MyConversationalPhoneBookBean** to open the source editor.
3. In the editor view, add business methods: `runFirstIteration()`, `runMiddleIteration()`, `endConversation()`, `isConvEnded()`, and `setComponentAuthorization()` and ensure the code matches the following Java code:

```

package sample.ims.conv;

import javax.ejb.EJBException;
import javax.resource.ResourceException;
import javax.resource.cci.ConnectionFactory;
import sample.ims.conv.record.*;
import com.ibm.connector2.ims.ico.IMSConnectionSpec;
import com.ibm.connector2.ims.ico.IMSManagedConnectionFactory;

/*****
/*
/* (c) Copyright IBM Corp. 2001, 2002, 2003
/* All Rights Reserved
/* Licensed Materials - Property of IBM
/*
/* DISCLAIMER OF WARRANTIES.
/*
/* The following code is provided to you solely for the purpose of
/* assisting you in the development of your applications.
/* The code is provided "AS IS." IBM MAKES NO WARRANTIES, EXPRESS OR
/* IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF
/* MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING
/* THE FUNCTION OR PERFORMANCE OF THIS CODE.
/* IBM shall not be liable for any damages arising out of your use
/* of the provided code, even if it has been advised of the
/* possibility of such damages.
/*
/* DISTRIBUTION.
/*
/* This code can be freely distributed, copied, altered,
/* and incorporated into other software, provided that:
/* - It bears the above Copyright notice and DISCLAIMER intact
/* - The software is not offered for resale
/*
*****/

/**
 * Bean implementation class for Enterprise Bean: ConversationalPhoneBook
 */
public class MyConversationalPhoneBookBean
    extends com.ibm.connector2.ims.ico.IMSConversationalHelper
    implements javax.ejb.SessionBean {

    private javax.ejb.SessionContext mySessionCtx;

    /* The middleIteration variable is a flag for whether to
     * expect a runMiddleIteration() method call
     */
    private boolean middleIteration = false;

    private int myExecTimeout=0;
    private String myLtermName="";
    private String myMapName="";
    private int mySocketTimeout=0;

    public javax.ejb.SessionContext getSessionContext() {
        return mySessionCtx;
    }

    public void setSessionContext(javax.ejb.SessionContext ctx) {
        mySessionCtx = ctx;
    }

    public void ejbCreate() throws javax.ejb.CreateException {
    }

    public void ejbActivate() {

```

```

    }

    public void ejbPassivate() {
    }

    public void ejbRemove() {
        try {
            /* When the EJB is removed from the EJB container due to a session
             * timeout, ejbRemove() will be called to clean it up. This code will
             * automatically end the conversation when the bean is removed, terminating
             * the stranded conversation
             */
            this.endConversation();
        } catch (Exception e) {
            throw new EJBException(e);
        }
    }

    /**
     * Ends the current conversation with the conversational IVP phone book
     IMS application
     */
    public void endConversation() {
        try {
            super.endConversation();
            this.middleIteration = false;
        } catch (Exception e) {
            throw new EJBException(e);
        }
    }

    /**
     * Begins a conversation with the conversational IVP phone book
     * IMS application.
     *
     * The provided input record will be used to specify the input
     to the IMS
     * application. Make sure that the tranocode parameter of the input
     record has been
     * set to "IVTCB" to use the conversational IVP application.
     *
     * Do not call this method more than once while in a conversation.
     * If you call it more
     * than once, your conversation will be ended and a RuntimeException
     * will be thrown.
     *
     * @param input the CCI input record to send to IMS
     *
     * @return sample.ims.conv.CPBOutputRecord the output returned by the
     * IMS application.
     *
     * @throws RuntimeException when the runFirstIteration() is called more
     * than once in a conversation
     */
    public CPBOutputRecord runFirstIteration(CPBInputRecord input) {
        try {
            /* Check to see if runFirstIteration() is being called when we're
             * already in a conversation. If it is, we end the conversation and
             * throw an exception
             */
            if (this.middleIteration) {
                this.endConversation();
                throw new RuntimeException("Invalid iteration.Must end conversation before
                starting new one.Conversation is force-ended");
            }

            // Instantiate an output record

```

```

CPBOutputRecord output = new CPBOutputRecord();

super.setExecutionTimeout(myExecTimeout);
super.setLtermName(myLtermName);
super.setMapName(myMapName);
super.setSocketTimeout(mySocketTimeout);
// Send the input to IMS Connect
super.execute(input, output);

/* If the conversation succeeded, we set the middleIteration variable
 * to true. It will be used as a flag to catch invalid calls to
 * runFirstIteration()
 */
if (!this.isConvEnded()) {
    this.middleIteration = true;
}

return output;
} catch (Exception e) {
    throw new EJBException(e);
}
}

/**
 * Continues a conversation previously started with the {@link
 * runFirstIteration} method.
 *
 * The provided input record will be used to specify the input to the IMS
 * application.
 *
 * Do not call this method without first calling {@link runFirstIteration}.
 * If you do, a RuntimeException will be thrown to notify you of the error.
 *
 * @param input the CCI input record to send to IMS
 *
 * @return sample.ims.conv.CPBOutputRecord the output returned by the IMS
 * application.
 *
 * @throws RuntimeException when runMiddleIteration() is called without first
 * calling runFirstIteration()
 */
public CPBOutputRecord runMiddleIteration(CPBMiddleInputRecord input) {
    try {
        /* Here we check to see what iteration we are currently in. If we haven't
         * yet run runFirstIteration() we throw a RuntimeException rather than
         * allowing the iteration to fail
         */
        if (!this.middleIteration) {
            throw new RuntimeException("Invalid iteration. Start a conversation
first.");
        }

        // Instantiate the output record
        CPBOutputRecord output = new CPBOutputRecord();

        super.setExecutionTimeout(myExecTimeout);
        super.setLtermName(myLtermName);
        super.setMapName(myMapName);
        super.setSocketTimeout(mySocketTimeout);

        // Send it to IMS
        super.execute(input, output);

        /* Check the status of the conversation. If the conversation has ended
         * for some
         * reason, we can perform a runFirstIteration() call again

```

```

        */
        if (this.isConvEnded()) {
            this.middleIteration = false;
        }

        return output;
    } catch (Exception e) {
        throw new EJBException(e);
    }
}

/**
 * Returns the current status of the conversation
 *
 * @return boolean true if the conversation has ended, false if it has not
 */
public boolean isConvEnded() {
    return super.isConvEnded();
}

/**
 * Allows the user to specify username, password, and groupname values to
 * override
 * the defaults provided by the ConnectionFactory.
 *
 * Most useful for ComponentAuthorization. A ConnectionSpec will be created
 * and provided to the helper class.
 *
 * @param username the username value to use for the ConnectionSpec
 * @param password the password value to use for the ConnectionSpec
 * @param groupname the groupname value to use for the ConnectionSpec
 */
public void setConnectionSpec(
    String username,
    String password,
    String groupname) {
    try {
        IMSConnectionSpec connSpec = new IMSConnectionSpec();
        connSpec.setUserNames(username);
        connSpec.setPassword(password);
        connSpec.setGroupName(groupname);
        super.setConnectionSpec(connSpec);
    } catch (Exception e) {
        throw new EJBException(e);
    }
}

/**
 * Allows the user to specify LTermname, Mapname, Execution Timeout and
 * Socket Timeout values for the conversation.
 *
 * @param anLtermname the LTermname value to use for the InteractionSpec
 * @param aMapname the Mapname value to use for the InteractionSpec
 * @param anExecTimeout the Execution Timeout value (in milliseconds) to
 * use for the InteractionSpec
 * @param aSocketTimeout the Socket Timeout value (in milliseconds) to
 * use for the InteractionSpec
 */
public void setInteractionSpec(
    String anLtermname,
    String aMapname,
    int anExecTimeout,
    int aSocketTimeout) {
    this.myLtermName=anLtermname;
    this.myMapName=aMapname;
    this.myExecTimeout=anExecTimeout;
    this.mySocketTimeout=aSocketTimeout;
}

```

```

    }

}

```

4. Press **Ctrl-S** to save your changes and then close the editor.

Step 4: Adding the business methods to the remote interfaces

The remote interfaces specify which business methods of an EJB can be used by client applications. Generating the EJB automatically builds an empty Remote Interface and a Home Interface with a default create method. To make the EJB business methods you created in Step 3, accessible to client applications, add the methods to the Remote Interface by completing the following steps:

1. In the J2EE Hierarchy tab, expand **EJB Modules** > **MyConversationalPhoneBookEJB** > **MyConversationalPhoneBook**.
2. In the Outline pane, expand the **MyConversationalPhoneBookBean.java** class and select all of the following methods:
 - **runFirstIteration()**
 - **runMiddleIteration()**
 - **isConvEnded()**
 - **endConversation()**
 - **setConnectionSpec()**
 - **setInteractionSpec()**
3. Right-click any of the selected methods and select **Enterprise Bean** > **Promote to Remote Interface**. This adds a declaration for each of the selected methods to the Remote Interface. The method declarations are added to the Remote Interface file, **MyConversationalPhoneBook.java**.

Step 5: Setting the EJB transaction attributes for the EJB session bean

In Step 2 where you created a conversational EJB session bean, you specified a **TransactionType** of **Container**. Setting the transaction attributes specifies how a container should manage a specific method or all of the methods within the remote interface of an EJB. Because the IMS resource adapter currently only supports **Commit Mode 1** with **OTMA SyncLevel None** for each iteration of a conversation, the only transactional attribute of a conversational EJB is **Not Supported**.

To set the EJB transaction attributes, complete the following steps:

1. Click the **J2EE Hierarchy** tab in the Business Integration perspective and expand **EJB Modules**. Right-click **MyConversationalPhoneBookEJB** and select **Open With** > **Deployment Descriptor Editor**.
2. Click the **Assembly Descriptor** tab.
3. Under **Container Transactions**, click **Add**. The **Add Container Transaction** wizard opens. In the **Enterprise Bean Selection** page, select the **MyConversationalPhoneBook** bean and click **Next**.
4. Click the **Container transaction type** field and select **Not Supported** from the pull-down list.
5. Under **Methods found**, click the checkbox for the **MyConversationalPhoneBook** bean to select all of its methods. Click **Finish**.
6. Press **Ctrl-S** to save your changes and then close the editor.

Step 6: Adding the record classes as a project utility JAR and adding the JAR to the JAR dependency list

In this sample, the record classes for the input and out messages are in a separate project, **MyConversationalPhoneBookCCIRRecordClasses**. To ensure that the record classes are available to the deployed EAR file, they must be added to the Enterprise Application project as a utility JAR.

To add the record classes to the project as a utility JAR, complete the following steps:

1. Click the **J2EE Hierarchy** tab in the Business Integration perspective and expand **Enterprise Applications**.
2. Right-click **MyConversationalPhoneBook** and select **Open With > Deployment Descriptor Editor**.
3. In the Deployment Descriptor editor, select the **Module** tab. In the Module page under Project Utility Jars, click **Add**.
4. The Add Utility Jar window opens. Select **MyConversationalPhoneBookCCIRRecordClasses** project as a utility JAR. Click **Finish**.
5. Press **Ctrl-S** to save the changes and then close the EJB Deployment Descriptor editor.

In addition, the utility JAR created above must be added to the JAR dependency list of the EJB project. To add the utility JAR to the JAR dependency list, complete the following steps:

1. In the J2EE Hierarchy tab, expand **EJB Modules** and right-click **MyConversationalPhoneBookEJB**. Select **Open With > JAR Dependency Editor**.
2. The JAR Dependency editor opens. In the Dependencies section, check the box next to **MyConversationalPhoneBookCCIRRecordClasses.jar** to add it to the dependency list.
3. Press **Ctrl-S** to save the changes and then close the JAR Dependency editor.

Step 7: Generating the deployed code

To use the conversational EJB in a deployed application, you must first generate the deployment code for the EJB session bean. The deployment classes allow your bean to run on an EJB server. To generate the deployment code, follow these steps:

1. Click on the **J2EE Hierarchy** tab in the Business Integration perspective and expand **EJB Modules**.
2. Right-click **MyConversationalPhoneBookEJB** and select **Generate > Deploy and RMIC Code**.
3. The Generate Deploy and RMIC Code wizard opens. Select the enterprise java bean by checking the box next to **MyConversationalPhoneBook** and click **Finish**.

Part 4: Building a conversational application

An EJB session bean is a general purpose component that can be used to create many different types of applications. For example, it can be used to build a SOAP service, a Java application, or a web application. In this sample, the EJB session bean is used in a simple web application. To build a simple web application you need to create JSP web pages in your web project. These JSP web pages are used to

execute the iterations of the conversation and provide a mechanism for providing the input and displaying the output of the iterations. In a complicated conversation, you may have many web pages. This section shows how to generate an access bean that can be easily invoked from a web page.

Step 1: Generating the access beans

This step provides instructions on how to generate an access bean. The access bean simplifies access to the Home and Remote Interfaces of your enterprise bean and allows a standard Java bean approach to using your EJB. To generate an access bean, complete the following steps:

1. Click on the **J2EE Hierarchy** tab in the Business Integration perspective and expand **EJB Modules**.
2. Right-click **MyConversationalPhoneBookEJB** and select **New > Access Beans**. The Add an Access Bean wizard opens.
3. Select **Java bean wrapper** for the access bean type and click **Next**.
4. Ensure that the EJB project name is **MyConversationalPhoneBookEJB** and select the **MyConversationalPhoneBook** access bean. Click **Finish**.
5. Click on the **Package Explorer** tab and right-click **MyConversationalPhoneBookWeb**. Select **Properties**.
6. In the Properties for MyConversationalPhoneBookWeb window, select **Java Build Path** and click the **Projects** tab. Accept all the defaults and select both projects, **MyConversationalPhoneBookCCIRecordClasses** and **MyConversationalPhoneBookEJB**.
7. Click **OK**.

Step 2: Importing the conversational web pages

To import the conversational web pages with this sample, complete the following steps:

1. Click on the **Package Explorer** tab and expand **MyConversationalPhoneBookWeb > WebContent**.
2. Right-click **WebContent** and the Import Select page opens. Select **Import > File System**. Click **Next**.
3. The Import File System page opens. Click **Browse** next to the **Directory** field and go to

```
WS_installdir\wstools\eclipse\plugins\com.ibm.etools.ctc.samples.  
ims_5.1.0\sampleparts
```

where *WS_installdir* is the directory where WebSphere Studio is installed.

4. Select the following three files:
 - **index.html**
 - **results.jsp**
 - **error.jsp**
5. Click **Finish**.

The index.html page is a simple web form that is used to provide input for the first iteration of the conversation and invoke the results.jsp web page. The results.jsp web page is the key web page of the application. It is used to invoke the **runFirstIteration()** and **runMiddleIteration()** methods of the access bean, **ConversationalPhoneBookAccessBean**, as well as to present the results from the first iteration and middle iterations of the conversation. The results.jsp page can be

used repeatedly for the middle iterations of the conversation. The logic contained in the results.jsp file determines which business methods to call for given inputs, catches and displays exceptions such as a user's EJB session time out, and handles the termination of the conversation. The error.jsp file is used to display exceptions.

Optional: If you want to create your own web pages, you can create them directly with the tooling provided by WebSphere Studio. To create a JSP page, complete the following steps:

1. Click on the **Package Explorer** tab and expand **MyConversationalPhoneBookWeb > WebContent**.
2. Right-click **WebContent** and select **New > Other**.
3. The Select page of the New wizard opens. In the left-pane, select **Web** and in the right-pane select **JSP file**. Click **Next**.
4. The New JSP File window opens. In the page:
 - a. Ensure that the Destination folder is:
/MyConversationalPhoneBookWeb/WebContent.
 - b. Type the name of your JSP file in the **Name** field.
 - c. Ensure the **Markup language** is **HTML**.
 - d. Click **Finish**.
5. The new JSP file that you created opens in the source editor. Add your code to the file.
6. Press **Ctrl-S** to save your changes and then close the editor.

Part 5: Configuring the server and running the sample

To run this sample, you must deploy the Enterprise Application project, in the form of an EAR file, to a server. In this case, the server runs in the WebSphere Unit Test Environment. This server must be configured and started. For this sample, you need to create one server instance and server configuration. A server instance identifies the run-time environment that you want to use for testing your project resources. A server configuration contains information that is required to set up and publish to a server. After you configure the server, you will deploy the EAR project. **Note:** If you already created a server and server configuration, you still need to deploy the EAR file to the server.

Step 1: Configuring the server and deploying the Enterprise Application (EAR)

To create a server instance and configuration, complete the following steps:

1. In the Business Integration perspective, click the **Server Configuration** tab to open the Server Configuration view. Right-click anywhere in the Server Configuration view. Select **New > Server and Server Configuration**. The create a New Server and Server Configuration wizard opens. (**Note:** If the Server Configuration tab is not available, you need to open the Server window. From the file menu select **Window > Show View > Server**.)
2. Type **myIMSServicesServer** for the server name. (The default folder name is Servers.)
3. Expand **WebSphere version 5.1** and select **Integration Test Environment**. Leave the template set to **None** and click **Next**.
4. Click **Yes** to create a server project named **Servers**.

5. The server port number defaults to **9080**. The port identifies the location of the service. Click **Finish**. The new server instance appears in the Server Configuration view and in the Servers view.

You have just created an instance of the WebSphere Application Server that is emulated by the WebSphere Test Environment running on your local host on port 9080.

Next you must add an instance of a JCA connection factory and configure its properties. The connection factory provides connections to the EIS on demand. In the case of the IMS resource adapter, an instance of an `IMSCConnectionFactory` provides an application with connections to IMS OTMA through IMS Connect.

You specify all of the information needed by the resource adapter to connect to a particular instance of the EIS. For the IMS resource adapter, with TCP/IP connections, you must specify at least the `HostName`, `DataStoreName`, and `PortNumber` properties. For Local Option connections, you must specify values for at least the `IMSCConnectName` and `DataStoreName` properties. These values determine the IMS that will be accessed through all of the connections created by this instance of the connection factory.

You also specify a JNDI lookup name under which the new connection factory instance will be available to an application component. The resource reference of an application component is mapped to the JNDI lookup name so at runtime the component can use the connection factory to make connections to the EIS. To create and configure a connection factory, complete the following steps:

1. Click the **Server Configuration** tab to open the Server Configuration view. Expand **Servers**.
2. Double-click the server configuration, **myIMSServicesServer**. An editor opens.
3. Click the **J2C** tab. Click **Add** beside the **J2C Resource Adapters** table.
4. From the Resource Adapters name list, select the resource adapter named **IMS**. This is the name you chose when you imported the IMS resource adapter into WebSphere Studio in the section "Before you begin." Click **OK**.
5. In the J2C Resource Adapters table, select the **IMS resource adapter**, then click **Add** beside the **J2C Connection Factories** table. The application client will look up this connection factory instance using the JNDI interface. The application component (EJB) will then use the connection factory instance to get a connection to the underlying IMS.
6. In the Create Connection Factory window, type a name for the new connection factory. For example, `ims_cf`. Type a JNDI name for the new connection factory. For example, `MyIMSTarget`. Click **OK**.
7. In the Resource Properties table, type the property values appropriate for your environment. You might need to scroll down to see this table. See Connection properties for a description of these properties. For example,
 - In the **HostName** field, type `MYHOST.ABC.XYZ.COM`.
 - In the **PortNumber** field, type `9999`.
 - In the **DataStoreName** field, type `MYDSTOR`.
8. Press **Ctrl-S** to save the changes and then close the editor.

Next you need to provide an EJB resource reference to the Connection Factory you just created. Because the JNDI lookup for your conversational EJB is done by the `IMSConversationalHelper` class, the EJB resource reference for your conversational

EJB must match the resource reference used by the `IMSConversationalHelper` class. To provide a resource reference and map it to your new connection factory, complete the following steps:

1. Click on the **J2EE Hierarchy** tab in the Business Integration perspective and expand **EJB Modules**.
2. Right-click **MyConversationalPhoneBookEJB** and select **Open with > Deployment Descriptor Editor**.
3. Click the **References** tab and select the **MyConversationalPhoneBook** bean. Click **Add** to add a new resource reference. The Add Reference wizard opens.
4. In the Reference window, select **Resource Reference** and click **Next**.
5. In the EJB Resource Reference window, type `ibm/ims/IMSTarget` in the **Name** field. The `IMSConversationalHelper` class uses `ibm/ims/IMSTarget` to locate the connection factory. If you wish to use a value other than `ibm/ims/IMSTarget`, you can invoke the method `setJNDILookupName()` before the first invocation of the `execute()` method. A possible location for this call is the `ejbCreate()` method of your conversational EJB.
6. In the **Type** field, select `javax.resource.cci.ConnectionFactory` from the drop down box.
7. In the **Authentication** field, select **Container** from the drop down box.
8. Ensure that the **Sharing scope** field is set to **Unshareable**. This guarantees that the connection to IMS is unique and will not be closed at the end of an iteration, allowing IMS Connect to maintain the same connection for the duration of the conversation. If a connection is Shareable it is returned to the connection pool when the transaction ends. Because the conversational EJB has a **TransactionType** of **Container**, the application server will consider an iteration to be in the scope of a transaction, even though the container transaction type for all the methods of the EJB have been designated as **Not Supported**.
9. Click **Finish**.
10. In the **References** tab, expand the **MyConversationalPhoneBook** bean and select the new ResourceRef you just created, `ibm/ims/IMSTarget`.
11. Under the WebSphere Bindings section, type the JNDI name you chose for your connection factory. For example, `MyIMSTarget`.
12. Press **Ctrl-S** to save the changes, and then close the Deployment Descriptor editor.

Because the conversational EJB is a stateful session bean, you can configure the duration in seconds before the session bean times out. When the timeout value is reached, the EJB is removed. Removing the EJB will end the conversation. Pick a timeout value that is appropriate for your conversation.

1. Click the **J2EE Hierarchy** tab in the Business Integration perspective and expand, **EJB Modules**.
2. Right-click **MyConversationalPhoneBookEJB** and select **Open with > Deployment Descriptor Editor**.
3. Click the **Beans** tab and select the **MyConversationalPhoneBook** bean. Scroll down to the **Session Timeout** section and enter a value for the Timeout integer. For example, a value of 600 seconds (10 minutes).
4. Press **Ctrl-S** to save the changes, and then close the Deployment Descriptor editor.

Finally, you need to add the EAR project (MyConversationalPhoneBook) to the server configuration that you created. To add the project, complete the following steps:

1. In the Server Configuration view under Servers, right-click **myIMSServicesServer**.
2. Select **Add > MyConversationalPhoneBook**. MyConversationalPhoneBook is the name of the Enterprise Application Project that you created earlier.

You have now successfully generated an enterprise service from an IMS transaction and deployed that service to the WebSphere test environment.

Step 2: Running the sample

To run the sample, complete the following steps:

1. In the Package Explorer view, expand **MyConversationalPhoneBookWeb > WebContent**.
2. Right-click **index.html** and select **Run on Server**.
3. The URL is displayed in a web browser:
`http://localhost:9080/MyConversationalPhoneBookWeb/index.html`
4. Type data in the fields provided on the web page. For example, type the following information for **Last Name: Last1**.
5. Select the command to be performed. For example, select **Display** and then click **Submit** to submit the data for processing. The First Iteration Results page appears with the status **Entry Was Displayed**.
6. To continue the conversation, select another command and then click **Submit** to submit data for processing. To end the conversation, select either the **End** or **Force End** command option. The End command option sends an input message containing the END command to the IMS application program and the IMS application program ends the conversation in response to the input message. The Force End command option sends a special OTMA message to IMS Connect, resulting in IMS ending the conversation.

Congratulations! You have now successfully built an Enterprise Java bean to communicate with a conversational IMS application.

Sample: Building an Application to Process Variable Length and Multiple Segment IMS Transaction Output Messages

Objective

This sample illustrates how to use WebSphere Studio Application Developer Integration Edition 5.0 to build a simple Java application that processes an IMS transaction that returns a multi-segment output message. The methodology used by this sample can also be used to build an application that processes a variable length IMS transaction output message.

The steps for building this application are very similar to the steps described in WebSphere Studio's online help, **Help > Help Contents > WebSphere Studio > Developing > Enterprise Services > IMS Services > Samples > Creating an enterprise service for an IMS transaction**. For this reason, the multi-segment sample will provide high level descriptions for some steps. For other steps you can refer to the online help for more details.

Before you begin

The IMS transaction that is used in this sample is *not* one of the IMS Installation Verification Programs. This sample uses DFSDDLTO, an IMS application program that issues calls to IMS based on control statement information. The DFSDDLTO control statements for this sample are provided below. However, to run this sample you must configure your environment for DFSDDLTO and provide the necessary JCL.

DFSDDLTO control statements

```
S11 1 1 1 1 TP 1
L GU
E OK
E Z0017 DATA SKS2 M2 SI1M3 SI1
WTO SEGMENT S11 RECEIVED
L GN
E QD
WTO END OF INPUT SEGMENTS
L ISRT IW06OUT
L Z0012 DATA *****M1S01
E OK
WTO SEGMENT S01 INSERTED
L ISRT
L Z0027 DATA *****M1S02*****M2S02
E OK
WTO SEGMENT S02 INSERTED
L ISRT
L Z0048 DATA *****M1S03*****M2S03*****M3S03
E OK
WTO SEGMENT S03 INSERTED
WTO CURRENT PROGRAM STLDDLTO TERMINATED
L GU
```

Note: This sample uses SKS2 as the transaction code for the DFSDDLTO application.

In addition to setting up DFSDDLTO, this sample assumes:

- Your runtime environment meets the prerequisites for using the IMS resource adapter.
- You have imported the IMS resource adapter into your WebSphere Studio environment.

Description

This sample uses the COBOL code below to describe the IMS transaction input and output messages. Note that the output message returned by IMS consists of three fixed length segments:

- OUTPUT-SEG1 (16 bytes)
- OUTPUT-SEG2 (31 bytes)
- OUTPUT-SEG3 (52 bytes)

The output message returned by this particular IMS application is a fixed size of 99 bytes and is represented by the COBOL 01 structure OUTPUT-MSG.

One way of developing this multi-segment application is to use the COBOL definition OUTPUT-MSG to define the output of the transaction. A second way is to create an output message for the output of the transaction. The code provided with this sample uses the second method, since it can also be used to build an

application that processes a variable length output message. The COBOL definitions for the individual message segments will continue to be used to simplify access to the data of the individual segments. The COBOL code for this sample is shown below and is also in file **MSOut.ccp**:

```

01 INPUT-MSG.
   02 IN-LL          PICTURE S9(3) COMP.
   02 IN-ZZ          PICTURE S9(3) COMP.
   02 IN-TRCD        PICTURE X(5).
   02 IN-DATA1        PICTURE X(6).
   02 IN-DATA2        PICTURE X(6).

01 OUTPUT-MSG.
   02 OUT-ALLSEGS    PICTURE X(99) VALUE SPACES.

01 OUTPUT-SEG1.
   02 OUT-LL          PICTURE S9(3) COMP VALUE +0.
   02 OUT-ZZ          PICTURE S9(3) COMP VALUE +0.
   02 OUT-DATA1        PICTURE X(12) VALUE SPACES.

01 OUTPUT-SEG2.
   02 OUT-LL          PICTURE S9(3) COMP VALUE +0.
   02 OUT-ZZ          PICTURE S9(3) COMP VALUE +0.
   02 OUT-DATA1        PICTURE X(13) VALUE SPACES.
   02 OUT-DATA2        PICTURE X(14) VALUE SPACES.

01 OUTPUT-SEG3.
   02 OUT-LL          PICTURE S9(3) COMP VALUE +0.
   02 OUT-ZZ          PICTURE S9(3) COMP VALUE +0.
   02 OUT-DATA1        PICTURE X(15) VALUE SPACES.
   02 OUT-DATA2        PICTURE X(16) VALUE SPACES.
   02 OUT-DATA3        PICTURE X(17) VALUE SPACES.

```

Step 1: Creating the service project

1. Open the Business Integration perspective.
2. From the toolbar, click the **Create a service project** icon.
3. The New Project wizard opens. In the **Project name** field, type `MultiSegmentOutput`. Click **Finish**.

Step 2: Importing the COBOL file

1. Select the **MultiSegmentOutput** service project and click the **New Java package** icon.
2. Create a new package named `sample.ims`. Click **Finish**.
3. In the Services view, expand the **MultiSegmentOutput** service project and right-click the **sample.ims** package and select **Import**.
4. In the Import wizard, select **File system** and click **Next**.
5. On the File System page, click **Browse** and import **MSOut.ccp**, which is the COBOL source for the IMS transaction input and output message. The `MSOut.ccp` file is located in:

```

ws_installdir\wstools\eclipse\plugins\com.ibm.etools.ctc.samples.
ims_5.1.0\sampleparts

```

where the `ws_installdir` is the directory where WebSphere Studio is installed.

Step 3: Generating the enterprise service

1. Expand **Service Projects > MultiSegmentOutput > sample.ims**.
2. Right-click **MSOut.ccp** and select **New > Service built from**
3. In the Create Service page, select **IMS** and click **Next**.

4. In the Connection Properties page, enter the property values appropriate for your environment, then click **Next**. **Note:** Because the connection properties are not encrypted, you should remove at minimum the User name and password from the port definition after you have completed testing.
5. In the Service Binding page, ensure that the following values are correct:
 - The Source folder field contains /MultiSegmentOutput
 - The Package field contains sample.ims
 - The Target namespace is http://ims.sample/
6. In the **Interface file name** field, type MSO, then click **Finish** to accept all other default names. Click **OK**.

Define an operation for the first segment of the output message. This step is primarily for obtaining helper classes for the message segment. The helper classes for the segment consist of a Java bean with get and set methods for each of the COBOL fields in the OUTPUT-SEG1 01 data structure and a formatHandler that can be used to deserialize the COBOL buffer to the Java bean.

1. Ensure the MSOIMSBinding.wsdl file is open in the WSDL editor. If it is not open, double-click the file. Optionally, right-click the **MSOIMSBinding.wsdl** file and select **Open With > WSDL Editor**.
2. In the Bindings container of the Graph view, right-click MSOIMSBinding and select **Generate Binding Content**.
3. In the Specify Binding Details page, ensure **IMS** is selected for the **Protocol field** and click **Add** to add binding operations.
4. In the **Operation name** field of the Operation Binding page type outSegment1 for the name of the operation and select ONE_WAY as the type of operation. Click **Next**.
5. In the imsConnector Operation Binding Properties page, accept the default values and click **Next**.
6. In the Operation Binding page, click **Import** next to the Input message field. The File Selection page opens. Import the **MSOut.ccp** file to specify the XML schema definition for the input part.
 - Expand **MultiSegmentOutput > sample > ims** and select **MSOut.ccp**. Click **Next**.
 - In the COBOL Import Properties page, choose the z/OS platform and click **Next** to accept the default values.
 - In the COBOL Importer page, the data structures from the **MSOut.ccp** file are displayed. Select **OUTPUT-SEG1**. You can accept the default to overwrite the XSD types. Click **Finish**.
7. On the Operation Binding page, click **Finish** to complete the operation and return to the Binding wizard.
8. Click **Finish** to close the Binding wizard.
9. Repeat the steps used for the first segment of the message for the second and third segments of the message.
10. If you have a COBOL definition for the output message, such as the OUTPUT-MSG declaration in **MSOut.ccp**, you can proceed to the next step. If you do not have a COBOL definition for the output message, or if the message returned by the IMS transaction is variable length, you will need to create a message to contain the output of the IMS transaction. To create the output message, perform the following steps:

- a. Right-click **MSO.wsdl**, select **Open With > Source editor** to view the source. Locate the `<schema></schema>` section and add the following `complexType` following the last `complexType` in the section:

```
<complexType name="OutMsg">
  <annotation>
    <appinfo source="http://www.wsadie.com/appinfo">
      <messageBuffer>true</messageBuffer>
    </appinfo>
  </annotation>
</complexType>
```

Press **Ctrl-S** to save the file and then close the source editor.

- b. Right-click **MSO.wsdl** and select **Open With > WSDL editor**.
- c. In the Messages Container of the Graph view, right-click on the **Messages** title bar and select **Add Child > message**.
- d. The New Message window opens. Type **OutMsg** for the name of the message and click **OK**. The new message appears in the Message container.
- e. Right-click **OutMsg** and select **Add Child > part**.
- f. In the New Part window, type **buffer** for the name of the new message part and click **OK**. The new part, **buffer**, appears in the Message container.
- g. Right-click the **buffer** part and select **Set Type**.
- h. In the Specify Type wizard, select the **Import type from a file** radio button.
- i. In the Workbench files list, select the **Browse** button and scroll to **MultiSegmentOutput**.
- j. Expand **MultiSegmentOutput > sample > ims** and select **MSO.wsdl**.
- k. Select **OutMsg** from the drop down list and click **Finish**.
- l. Save the changes to **MSO.wsdl** by pressing **Ctrl-S**.
- m. Open the binding WSDL file, **MSOIMSBinding.wsdl**, in the source view. Scroll down and locate the `<format:typeMapping></format:typeMapping>` section and add the following `typeMap` to the section:


```
<format:typeMap formatType="MSOIMSBinding" typeName="tns:OutMsg"/>
```

Adding the above `typeMap` entry enables a format handler to be created for the output message.

11. Define an operation to run the IMS transaction:
 - a. Open the binding WSDL file, **MSOIMSBinding.wsdl**, in the WSDL editor.
 - b. In the Bindings container of the Graph view, right-click **MSOIMSBinding** and select **Generate Binding Content**.
 - c. In the Specify Binding Details page, ensure **IMS** is in the Protocol field and click **Add** to add binding operations.
 - d. In the **Operation name** field of the Operation Binding page type **runMultiSegOutput** for the name of the operation. Leave the type of operation as **REQUEST_RESPONSE** and click **Next**.
 - e. In the **imsConnector Operation Binding Properties** page accept the default values and click **Next**.
 - f. In the Operation Binding page, click **Import** next to the Input message field. The File Selection page opens. Import the **MSOut.ccp** file to specify the XML schema definition for the input part.

- Expand **MultiSegmentOutput** > **sample** > **ims** and select **MSOut.ccp**. Click **Next**.
 - In the COBOL Import Properties page, choose the z/OS platform and click **Next** to accept the default values.
 - In the COBOL Importer page, the data structures from the **MSOut.ccp** file are displayed. Select **INPUT-MSG**. You can accept the default to overwrite the XSD types. Click **Finish**.
- g. In the Operation Binding page, ensure that the **Import or use an existing message** radio button is selected.
- If you have a COBOL definition for the output message, complete the following steps:
- Click **Import** next to the Output message field. The New File Selection page opens. Import the **MSOut.ccp** file to specify the XML schema definition for the output part.
 - Expand **MultiSegmentOutput** > **sample** > **ims** and select **MSOut.ccp**. Click **Next**.
 - In the COBOL Import Properties page, choose the z/OS platform and click **Next** to accept the default values.
 - In the COBOL Importer page, the data structures from the **MSOut.ccp** file are displayed. Select **OUTPUT-MSG** and change the name in the XSD name field to **OutMsg**. You can accept the default to overwrite the XSD types. Click **Finish**.
- If you created an output message, complete the following steps:
- Click **Browse** next to the Output message field. The Select a message Window opens.
 - Expand **MultiSegmentOutput** > **sample.ims** and select **MSO.wsdl**.
 - In the Choose a message from a WSDL file drop down list, select **OutMsg** from the drop down list and click **OK**.
- h. Click **Finish** on the Operation Binding page to complete the operation and return to the Bindings page.
- i. Click **Finish** to close the Binding Wizard.

Step 4: Creating a Java service proxy and helper classes

1. Expand the **MultiSegmentOutput** project and the **sample.ims** package. Select the service file **MSOIMSService.wsdl**.
2. Right-click the file and select **Enterprise Services** > **Generate Service Proxy**. The Generate Service Proxy wizard opens.
3. The Proxy Selection page opens, ensure the type of proxy you want to generate is selected and click **Next**.
4. In the Service Proxy page, ensure that the service you want to create the proxy for is shown.
 - a. Change the class name for the proxy to **MSOProxy.java**, if necessary.
 - b. Ensure that the package name is **sample.ims**.
 - c. Ensure that Generate helper classes is selected. Click **Next**.
5. In the Service Proxy page, specify the style of the proxy and the operations to expose in the proxy:
 - a. Select the **Client stub** proxy style.
 - b. Select the **MSO** check box so all the operations are included in the proxy.

Note: By selecting all the operations, helper classes will be generated for all the operations, even though only the **runMultiSegmentOutput** operation

will be used in the proxy. This can also be accomplished by only including the **runMultiSegmentOutput** operation in the proxy and generating helper classes for the other operations separately.

6. Click **Finish**. The Java service proxy, **MSOProxy**, is generated in the **MultiSegmentOutput** project.

Step 5: Using the Java service proxy to test the enterprise service

1. Expand the **MultiSegmentOutput** project and then select the **sample.ims** package. From the toolbar click the **New Java class** icon
2. In the Java class page, ensure **MultiSegmentOutput** is the source folder and that **sample.ims** is the package name.
3. Type **TestMSOProxy** for the name of the new class.
4. Accept all other defaults and click **Finish**.
5. Replace the code in the editor with the following Java code:

```
package sample.ims;

import sample.ims.ims.ibmcbol.*;
import sample.ims.ims.ibmcbol.*;
import com.ibm.connector2.ims.ico.*;
import com.ibm.etools.marshall.util.MarshallIntegerUtils;

public class TestMSOProxy
{
    public static void main(String[] args)
    {
        byte[] segBytes = null;
        int srcPos = 0;
        int dstPos = 0;
        int totalLen = 0;
        int remainLen = 0;
        byte[] buff;
        short LL = 0;
        short ZZ = 0;

        try
        {
            // -----
            // Populate the IMS transaction input message with
            // data. Use the input message format handler method
            // getSize() to set the LL field of the input message.
            // -----
            INPUTMSGFormatHandler inFmtHndlr =
                new INPUTMSGFormatHandler();
            INPUTMSG input =
                (INPUTMSG) inFmtHndlr.getObjectPart();
            input.setIn__ll((short) inFmtHndlr.getSize());
            input.setIn__zz((short) 0);
            input.setIn__trcd("SKS2 ");
            input.setIn__data1("M2 SI1");
            input.setIn__data2("M3 SI1");

            // -----
            // Run the IMS transaction. The multi-segment output
            // message is returned.
            // -----
            MSOProxy proxy = new MSOProxy();
            OutMsg output = proxy.runMultiSegOutput(input);

            // -----
            // Retrieve the multi-segment output message as a
            // byte array using the output message format
```

```

// handler method getBytes().
// -----
OutMsgFormatHandler outFmtHndlr =
    (OutMsgFormatHandler) output._getFormatHandler();
segBytes = outFmtHndlr.getBytes();

// -----
// Note: At this point, if the IMS application program
// returned a variable length output message, the
// application would process the byte array segBytes.
// -----

srcPos = 0;
dstPos = 0;
totalLen = segBytes.length;
remainLen = totalLen;

// -----
// Populate first segment object from the byte array.
// -----
buff = null;
// Get length of segment.
LL =
    MarshallIntegerUtils
        .unmarshallTwoByteIntegerFromBuffer(
            segBytes,
            srcPos,
            true,
            MarshallIntegerUtils.SIGN_CODING_TWOS_COMPLEMENT);

// Put segment in byte array.
buff = new byte[LL];
System.arraycopy(segBytes, srcPos, buff, dstPos, LL);
remainLen -= LL;

// Create and populate segment object from byte array.
OUTPUTSEG1FormatHandler outSeg1FH =
    new OUTPUTSEG1FormatHandler();
outSeg1FH.setBytes(buff);
OUTPUTSEG1 S1 =
    (OUTPUTSEG1) outSeg1FH.getObjectPart();
System.out.println(
    "\nOutSeg1 LL is:    "
    + S1.getOut__ll()
    + "\nOutSeg1 ZZ is:    "
    + S1.getOut__zz()
    + "\nOutSeg1_DATA1 is: "
    + S1.getOut__data1());

// -----
// Populate second segment object the byte array..
// -----
srcPos += LL;
buff = null;
// Get length of segment.
LL =
    MarshallIntegerUtils
        .unmarshallTwoByteIntegerFromBuffer(
            segBytes,
            srcPos,
            true,
            MarshallIntegerUtils.SIGN_CODING_TWOS_COMPLEMENT);

// Put segment in byte array.
buff = new byte[LL];
System.arraycopy(segBytes, srcPos, buff, dstPos, LL);
remainLen -= LL;

```

```

// Create and populate segment object from byte array.
OUTPUTSEG2FormatHandler outSeg2FH =
    new OUTPUTSEG2FormatHandler();
outSeg2FH.setBytes(buff);
OUTPUTSEG2 S2 =
    (OUTPUTSEG2) outSeg2FH.getObjectPart();
System.out.println(
    "\nOutSeg2 LL is:  "
    + S2.getOut__ll()
    + "\nOutSeg2 ZZ is:  "
    + S2.getOut__zz()
    + "\nOutSeg2_DATA1 is: "
    + S2.getOut__data1()
    + "\nOutSeg2_DATA2 is: "
    + S2.getOut__data2());
// -----
// Populate third segment object the byte array.
// -----
srcPos += LL;
buff = null;
// Get length of segment.
LL =
    MarshallIntegerUtils
        .unmarshallTwoByteIntegerFromBuffer(
            segBytes,
            srcPos,
            true,
            MarshallIntegerUtils.SIGN_CODING_TWOS_COMPLEMENT);

// Put segment in byte array.
buff = new byte[LL];
System.arraycopy(segBytes, srcPos, buff, dstPos, LL);
remainLen -= LL;

// Create and populate segment object from byte array.
OUTPUTSEG3FormatHandler outSeg3FH =
    new OUTPUTSEG3FormatHandler();
outSeg3FH.setBytes(buff);
OUTPUTSEG3 S3 =
    (OUTPUTSEG3) outSeg3FH.getObjectPart();
System.out.println(
    "\nOutSeg3 LL is:  "
    + S3.getOut__ll()
    + "\nOutSeg3 ZZ is:  "
    + S3.getOut__zz()
    + "\nOutSeg3_DATA1 is: "
    + S3.getOut__data1()
    + "\nOutSeg3_DATA2 is: "
    + S3.getOut__data2()
    + "\nOutSeg3_DATA3 is: "
    + S3.getOut__data3());
}
catch (Exception e)
{
    System.out.println("\nCaught exception is: " + e);
}
}
}

```

6. Press **Ctrl-S** to save the changes and then close the editor.
7. Select **TestMSOProxy.java** and expand the **Run** icon on the toolbar by selecting the arrow beside it. From the pop-up menu, select **Run As > Java Application**.
8. If you are able to run the DFSDDL0 script provided with this sample, the Java application runs and you see the following output on the console:

```

OutSeg1_LL is: 16
OutSeg1_ZZ is: 768
OutSeg1_DATA1 is: *****M1S01

OutSeg2_LL is: 31
OutSeg2_ZZ is: 768
OutSeg2_DATA1 is: *****M1S02
OutSeg2_DATA2 is: *****M2S02

OutSeg3_LL is: 52
OutSeg3_ZZ is: 768
OutSeg3_DATA1 is: *****M1S03
OutSeg3_DATA2 is: *****M2S03
OutSeg3_DATA3 is: *****M3S03

```

A Note about Processing Variable Length IMS Transaction Output Messages

For this sample, the IMS application program returns a fixed length multisegment message in the byte array **segBytes**. If the IMS application program returns a variable length message it is also available to the application in the byte array **segBytes**. Because the message in **segBytes** is in the format used by the IMS application program, it must be converted to a representation appropriate to the Java application. The application can obtain the length (LL) of the variable message using the method illustrated above. If the message contains text data, it will be in the single byte EBCDIC encoding scheme, and should be converted to UNICODE for use by the Java application.

Sample: Building an Application to Process IMS Transaction Input and Output Messages Containing Arrays

Objectives

This sample illustrates how to use WebSphere Studio Application Developer Integration Edition Version 5.0 to build a simple Java application that processes an IMS transaction input message and output message which contain arrays. For this sample, the input message and output message of the transaction are identical.

Many applications use arrays with a variable number of elements. For example, an IMS transaction input message can contain an array that has a maximum of 1000 elements, but for a particular execution of the IMS transaction, the array may only have three elements. In this case, for best performance, it is appropriate to send a three element array from the IMS resource adapter to the IMS application program. This sample illustrates how to ensure that only the necessary number of elements are included in the input message.

The steps for building this application are very similar to the steps described in the sample, **Creating an enterprise service for an IMS transaction**. The sample, **Creating an enterprise service for an IMS transaction**, can be found in WebSphere Studio's online help, **Help > Help Contents > Business integration > Resource adapters > IMS resource adapter > Samples**. Because both samples are similar to one another, this sample will only provide the high level steps, and refer you to the online help for more details. The discussion focuses on the how to program the Java application to ensure only the necessary number of elements are included in the array sent to the IMS application program.

Before you begin

The IMS transaction that is used by this sample is **not** one of the IMS Installation Verification Programs. It uses DFSDDLTO, an IMS application program that issues calls to IMS based on control statement information. The DFSDDLTO control statements for this sample are provided below. However, you must configure your environment for DFSDDLTO and provide the necessary JCL if you wish to run the sample.

DFSDDLTO control statements

```
S11 1 1 1 1    TP    1

L          GU

E          OK

E  Z0088 DATA  SKS2 03CN001Cathy Tang          CN002Haley Fung
  X
                CN003Steve Kuo          123456

WT0 IC4JINOU: Single segment received from JITOC

L          GN

E          QD

WT0 IC4JINOU: End of input segments from JITOC

L          ISRT  JITOC53

L  Z0113 DATA  TRNCD04CN001Cathy T.          CN002Haley F.
  X
                CN003Steve K.          CN004Kevin F.
65432X
                1

E          OK

WT0 IC4JINOU: Single segment inserted - 3 elements !!!!!!!!!!!!!!!

L          GU
```

Note:

This sample uses SKS2 as the transaction code for the DFSDDLTO application. In addition to setting up DFSDDLTO, this sample assumes:

- That your runtime environment meets the prerequisites for using the IMS resource adapter.
- That you have imported the IMS resource adapter into your WebSphere Studio environment.

Description

This sample uses the COBOL code below to describe the IMS transaction input and output messages. The input and the output messages are identical and contain an array of "customer" elements, followed by a single field containing a function code. The array can have a maximum of eight elements, but for this sample only three elements are input to the IMS application program and only four elements are returned by the IMS application program. The COBOL code for this sample is shown below and is also in the file, **InEqualsOut.cbl**:

```

01  IN-OUT-MSG.
05  WS-LL                      PIC S9(3) COMP VALUE +0.
05  WS-ZZ                      PIC S9(3) COMP VALUE +0.
02  WS-TRCD                    PIC X(5).
05  INDX                      PIC 99.
05  WS-CUSTOMER OCCURS 1 TO 8 TIMES
    DEPENDING ON INDX.
    15 WS-CUST-NUMBER          PIC X(5).
    15 WS-CUST-NAME           PIC X(20).
05  WS-FUNC-CODE               PIC X(6).

```

Step 1: Creating the service project

1. Open the Business Integration perspective.
2. From the toolbar, click the **Create a service project** icon.
3. Use the New Service Project wizard to create a service project and name it InOutArray. Click **Finish**.

Step 2: Importing the COBOL file

1. Select the InOutArray service project and click the **New Java package** icon.
2. Create a new package named sample.ims. Click **Finish**.
3. In the Services view, expand the InOutArray service project and right-click the sample.ims package and select **Import**.
4. In the Import wizard, select **File system** and click **Next**.
5. On the File System page, click **Browse** and import **InEqualsOut.ccp**, which is the COBOL source for the IMS transaction input and output message. The MSOut.ccp file is located in:

```

ws_installdir\wstools\eclipse\plugins\com.ibm.etools.ctc.samples.
ims_5.1.0\sampleparts

```

where the *ws_installdir* is the directory where WebSphere Studio is installed.

Step 3: Generating the enterprise service

1. Expand **Service Projects > InOutArray > sample.ims**.
2. Right-click **InEqualsOut.cbl** and select **New > Service built from**.
3. In the Create Service page, select **IMS** and click **Next**.
4. In the Connection Properties page, enter the property values appropriate for your environment, then click **Next**. **Note:** Because the connection properties are not encrypted, you should remove at minimum the User name and password from the port definition after you have completed testing.
5. In the Service Binding page, ensure that the following values are correct:
 - The Source folder field contains /InOutArray
 - The Package field contains sample.ims
 - The Target namespace is http://ims.sample/
6. In the **Interface file name** field, type InOut, click **Finish** to accept all other default names, and then click **OK**.
7. Define an operation to run the IMS transaction.
 - a. The InOutIMSBinding.wsdl file opens in the WSDL editor.
 - b. In the Bindings container of the Graph view, right-click **InOutIMSBinding** and select **Generate Binding Content**.
 - c. The Specify Binding Details page opens. Ensure **IMS** is selected for the **Protocol** field and then click **Add** to add binding operations.

- d. In the **Operation name** field of the Operation Binding page type `runInOut` for the name of the operation and select **REQUEST_RESPONSE** as the type of operation. Click **Next**.
- e. In the `imsConnector` Operation Binding Properties page, accept the default values and click **Next**.
- f. In the Operation Binding page, click **Import** next to the **Input message** field. The File Selection page opens. Import the **InEqualsOut.cbl** file to specify the XML schema definition for the input part.
 - Expand **InOutArray** > **sample** > **ims** and select **InEqualsOut.cbl**. Click **Next**.
 - In the COBOL Import Properties page, choose the **z/OS** platform and click **Next** to accept the default values.
 - In the COBOL Importer page, the single data structure from the **InEqualsOut.cbl** file is displayed. Select **IN-OUT-MSG** for the input message. You can accept the default to overwrite the XSD types. Click **Finish**.
- g. In the Operation Binding page, select the check box **Use input message for output** and click **Finish**.
- h. In the Specify Binding Details page, click **Finish** to close the Binding wizard.

Step 4: Creating a Java service proxy and helper classes

1. Expand the **InOutArray** project and the **sample.ims** package. Select the service file **InOutIMSService.wsdl**.
2. Right-click the file and select **Enterprise Services** > **Generate Service Proxy**. The Generate Service Proxy wizard opens.
3. In the Proxy selection page, ensure that the service you want to create the proxy for is shown and click next.
 - a. Change the class name for the proxy to `InOutProxy.java`, if necessary.
 - b. Ensure that the package name is **sample.ims**.
 - c. Ensure that Generate helper classes is selected. Click **Next**.
4. In the Service Proxy page, specify the style of the proxy and the operations to expose in the proxy:
 - a. Select the **Client stub** proxy style.
 - b. Select the **InOut** check box. The single operation **runInOut** will be included in the proxy.
5. Click **Finish**. The Java service proxy, **InOutProxy**, is generated in the **InOutArray** project.

Step 5: Using the Java service proxy to test the enterprise service

1. Expand the **InOutArray** project and then select the **sample.ims** package. From the toolbar click the **New Java class** icon
2. Ensure that **InOutArray** is the source folder and that **sample.ims** is the package name.
3. Type `TestInOutProxy` for the name of the new class.
4. Accept all other defaults and click **Finish**.
5. Replace the code in the editor with the following Java code:
6. `package sample.ims;`

```
import com.ibm.connector2.ims.ico.*;
```

```

import sample.ims.ims.ibmcbol.*;

public class TestInOutProxy
{
    public static void main(String[] args)
    {
        try
        {
            // -----
            // Create the formatHandler, then create the input
            // message bean from the formatHandler.
            // -----
            INOUTMSGFormatHandler inoutFmtHndlr = new INOUTMSGFormatHandler();
            INOUTMSG input = (INOUTMSG) inoutFmtHndlr.getObjectPart();

            int sz = inoutFmtHndlr.getSize();
            System.out.println("\nInitial size of input message is: " + sz);

            // -----
            // Don't set the length (LL) field yet... wait until
            // input message has been adjusted to reflect only
            // the number of array elements actually sent.
            // -----
            input.setWs__zz((short) 0);
            input.setWs__trcd("SKS2 ");

            // -----
            // Construct an array and populate it with the elements
            // to be sent to the IMS application program. In this
            // case three elements are sent.
            // -----
            Inoutmsg_ws__customer[] customers = new Inoutmsg_ws__customer[3];

            Inoutmsg_ws__customer aCustomer1 = new Inoutmsg_ws__customer();
            aCustomer1.setWs__cust__name("Cathy Tang");
            aCustomer1.setWs__cust__number("CN001");
            customers[0] = aCustomer1;

            Inoutmsg_ws__customer aCustomer2 = new Inoutmsg_ws__customer();
            aCustomer2.setWs__cust__name("Haley Fung");
            aCustomer2.setWs__cust__number("CN002");
            customers[1] = aCustomer2;

            Inoutmsg_ws__customer aCustomer3 = new Inoutmsg_ws__customer();
            aCustomer3.setWs__cust__name("Steve Kuo");
            aCustomer3.setWs__cust__number("CN003");
            customers[2] = aCustomer3;

            // -----
            // Set the array on the input message.
            // -----
            input.setWs__customer(customers);
            input.setIndx((short) 3);

            System.out.println("\nInitial value of INDX is: " + input.getIndx());

            // -----
            // Flush contents of the input message bean into the
            // formatHandler.
            // -----
            input.fireElementEvents();

            // -----
            // Reallocate the buffer to the actual size
            // -----
            byte[] bytes = inoutFmtHndlr.getBytes();
            int size = inoutFmtHndlr.getSize();

```

```

byte[] newBytes = new byte[size];
System.arraycopy(bytes, 0, newBytes, 0, size);

// -----
// Set the bytes back into the format handler and set
// the length field of the input message, now that
// we know the actual size.
// -----
inoutFmtHndlr.setBytes(newBytes);
input.setWs__ll((short) size);
System.out.println("\nAdjusted size of input message is: " + size);
System.out.println("\nAdjusted size of INDX is: " + input.getIndx());

// -----
// Set fields that follow the array after the input
// message has been adjusted.
// -----
input.setWs__func__code("123456");

InOutProxy proxy = new InOutProxy();

INOUTMSG output = new sample.ims.INOUTMSG();
output = proxy.runInOut(input);

short outndx = output.getIndx();
System.out.println("\nOutput value of INDX is: " + outndx);

Inoutmsg_ws__customer outArray[] = output.getWs__customer();

for (int i = 0; i < outndx; i++)
{
    System.out.println(
        "\n"
        + outArray[i].getWs__cust__name()
        + outArray[i].getWs__cust__number());
}
catch (Exception e)
{
    if (e instanceof org.apache.wsif.WSIFException)
    {
        Throwable ic4jEx =
            ((org.apache.wsif.WSIFException) e).getTargetException();
        if (ic4jEx instanceof IMSDFSMessageException)
        {
            System.out.println(
                "\nIMS returned message: "
                + ((IMSDFSMessageException) ic4jEx).getDFSMessage());
        }
        else
        {
            System.out.println(
                "\nIMS Connector exception is: " + ic4jEx);
        }
    }
    else
    {
        System.out.println("\nCaught exception is: " + e);
    }
}
}
}

```

7. Press Ctrl-S to save the changes and then close the editor.
8. Select **TestInOutProxy.java** and expand the **Run** icon on the toolbar by selecting the arrow beside it. From the pop-up menu, select **Run As > Java Application**.

9. If you are able to run the DFSDDLTO script provided with this sample, the Java application runs and you see the following output on the console:
10. Initial size of input message is: 217

```
Initial value of INDX is: 8

Adjusted size of input message is: 92

Adjusted size of INDX is: 3

Output value of INDX is: 4

Cathy T.          CN001
Haley F.          CN002
Steve K.          CN003
Kevin F.          CN004
```

In addition, a trace of the OTMA message sent to IMS Connect/IMS by the IMS resource adapter shows that only three elements of the array are transmitted:

```
Buffer sent:
[
    0000024e 001c0000 5cc8e6e2 d1c1e55c |...+...*HWSJAV*| :
16      00000000 c0000000 c8e6e2f5 e2f5f2e8 |....{...HWS5S52Y| :
32      ...
    00000000 00000000 00000000 00000000 |.....| :
496     0000005c 0000e2d2 e2f240f0 f3c3d5f0 |...*..SKS2 03CN0| :
512     f0f1c381 a388a840 e3819587 40404040 |01Cathy Tang   | :
528     40404040 4040c3d5 f0f0f2c8 819385a8 |      CN002Haley| :
544     40c6a495 87404040 40404040 404040c3 | Fung          C| :
560     d5f0f0f3 e2a385a5 8540d2a4 96404040 |N003Steve Kuo  | :
576     40404040 40404040 f1f2f3f4 f5f6      |      123456   |
]
```

A trace of the OTMA message received by the IMS resource adapter from IMS Connect/IMS shows that only four elements of the array are transmitted:

```
Buffer received:
[
    00000253 5cc8e6e2 d1c1e55c 01800000
|....*HWSJAV*....| : 16
    0000f9f9 f9f94040 4040a0f0 0000003e |..9999
.0....| : 32
    ...
    00000000 00000000 00000000 00000075
|.....| : 480
    0300e3d9 d5c3c4f0 f4c3d5f0 f0f1c381 |..TRNCD04CN001Ca| :
496     a388a840 e34b4040 40404040 40404040 |thy T.         | :
512     4040c3d5 f0f0f2c8 819385a8 40c64b40 | CN002Haley F. | :
528     40404040 40404040 404040c3 d5f0f0f3 |      CN003    | :
544
```

```
560      e2a385a5 8540d24b 40404040 40404040 |Steve K.      | :
      40404040 c3d5f0f0 f4d285a5 899540c6 |      CN004Kevin F| :
576
      4b404040 40404040 40404040 40f6f5f4 |.      654| :
592
      f3f2f1      |321      |
      ]
```

Chapter 10. Adding operations, messages, and bindings from COBOL source

Use the WSDL editor to add operations including messages and the bindings for the service. The operation and message definitions can come from existing XMI files in the workbench, or you can import them from COBOL source files.

Follow these steps after you have generated the skeleton WSDL files to add operations, messages, and bindings for the service:

1. **Open to the Bindings page:** If you have just completed the step to generate the code, the binding file is now open for editing. If the binding file is not open, double-click the binding WSDL file and ensure the **Graph** tab is selected.

Before you can specify the bindings for operations, you must first add the operations to the interface WSDL file. **Note:** The WSDL editor allows you to add new operations from the Bindings page of the *binding WSDL file*, and the new operations will be added (and saved) to the *interface WSDL file*. That is, when you have the binding file open in the WSDL editor, you do not have to open the interface WSDL file to add the new operations.

2. **Add the new operations:** To add operations from the binding WSDL file, complete the following steps:
 - a. Under the Binding container in the Graph view of the WSDL editor, right-click the binding file and select **Generate Binding Content**. The New Operation Binding Wizard opens.
 - b. In the Specify Binding Details page, ensure that **IMS** is selected in the **Protocol** field and then click **Add** to add binding operations.
 - c. In the Operation Binding page, type a name for the new operation and then use the table below to help you select the correct type of operation. For example, if you have a SEND only transaction, select ONE_WAY for the type of operation.

InteractionVerb	Operation type
SYNC_SEND	ONE_WAY or REQUEST_RESPONSE
SYNC_SEND_RECEIVE	REQUEST_RESPONSE
SYNC_RECEIVE_ASYNCOUTPUT_*	REQUEST_RESPONSE
SYNC_END_CONVERSATION	ONE_WAY

Click **Next**.

3. **Specify the input and output messages for the operation:** To specify the input and output messages, complete the following steps:
 - a. In the Operation Binding page, click **Import** next to the **Input message** field. The File Selection page opens. Expand the folders and import the COBOL source file to specify the XML schema definition for the input part. Click **Next**.

Note: If you have not imported the source file into the workbench workspace, you will not see the COBOL source file in the Source files container. In this case, click **Import** to import the source file into the workbench. The Import wizard is opened. Follow the instructions in Importing a COBOL, MFS, or C file to import the source file. When you see the source file in the Source file container, select it and click **Next**.

- b. In the `imsConnector` Operation Binding Properties page, select the default values and click **Next**.
 - c. In the COBOL Import Properties page, choose the **z/OS** platform and click **Next** to accept the default values.
 - d. In the COBOL Importer page, the single data structure from the COBOL source file is displayed. Select **INPUT-MSG** for the input message. You can accept the default to overwrite the XSD types. Click **Finish**
 - e. In the Operation Binding page, click **Import** next to the output message. The File Selection page opens. Import the COBOL source file to specify the XML schema definition for the output part. Click **Next**.
 - f. In the `imsConnector` Operation Binding Properties page, select the default values and click **Next**.
 - g. In the COBOL Import Properties page, specify the values for the output. Click **Next**.
 - h. In the COBOL Import window, select **OUTPUT-MSG** in the data structures list, which will populate the XSD type name with **OUTPUTMSG**. You can accept the default to overwrite the XSD types. Click **Finish** to return to the Operation Binding page.
4. **Add the binding for the operation:**
 - a. In the Specify Binding Details page, the **Overwrite existing binding information check box** is selected. Click **Finish** to complete the operation and return to the Binding wizard.
 - b. Click **Finish** to close the Binding wizard. The operation and messages are added to the interface WSDL file. In the WSDL editor, under the Bindings container, you should see the new operation you have just added.
 5. **Save the binding definition to the binding file:** Press **Ctrl-S** to save the changes to the binding file.

When you have completed these steps, you can generate the deploy code for the service.

Notices

The XDoclet Documentation included in this IBM product is used with permission and is covered under the following copyright attribution statement: Copyright (c) 2000-2004, XDoclet Team. All rights reserved.

Portions based on *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Copyright (c) 1995 by Addison-Wesley Publishing Company, Inc. All rights reserved.

U.S. Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this documentation in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this documentation. The furnishing of this documentation does not give you any license to these patents. You can send license inquiries, in writing, to:

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

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

*Intellectual Property Dept. for Rational Software
IBM Corporation
20 Maguire Road
Lexington, Massachusetts 02421-3112
U.S.A.*

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this documentation and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

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 and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application

programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

(C) (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. (C) Copyright IBM Corp. 2000, 2005. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

Programming interface information is intended to help you create application software using this program.

General-use programming interfaces allow you to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and service marks

See <http://www.ibm.com/legal/copytrade.shtml>.



Printed in USA

SC09-7869-04

