MQSeries® Adapter Builder for Windows NT®

Using the Control Center

Version 1 Release 0 Modification 1

GC34-5882-02

MQSeries® Adapter Builder for Windows NT®

Using the Control Center

Version 1 Release 0 Modification 1

GC34-5882-02

Note: Before using this information and the product it supports, read the information in "Notices" on page 51.

Third Edition (April 2001)

This edition applies to version 1, release 0, modification 1 of MQSeries Adapter Builder for Windows NT (product number 5639-L40) and to all subsequent releases and modifications until otherwise indicated in new editions.

IBM welcomes your comments. You can make comments on this information via e-mail at idrcf@hursley.ibm.com.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Figures

Tables

Welcome to the MQSeries Adapter Builder

This information describes the MQSeries Adapter Builder and explains how to plan for, install and use it.

Who should use this information

This information is for those who need to plan for, install or use the MQSeries Adapter Builder.

In order to use this information to help you build adapters, it is assumed that you already know:

- Programming terms, concepts, and methods that are appropriate to the messages with which you are working: C language, Java language, XML or XML OAG.
- About the MQSeries Adapter Kernel. Refer to the MQSeries Adapter Kernel documentation for more information.
- About the source applications and target applications that you are integrating.

Related information

See:

- The `readme.txt` file first. This file contains information that became available after this book was completed. Before installation, the `readme.txt` file is located in the same directory as `setup.exe` on the CD-ROM or local area network. After installation, the `readme.txt` file is located in the `root` directory of the MQSeries Adapter Builder installation.
- MQSeries Adapter Builder's extensive help system.
- The MQSeries Adapter Offering's website at http://www.ibm.com/software/ts/mqseries/adapter/information/.

  By following links from this website you can:
  - Obtain the latest information about MQSeries Adapter Offering.
  - Access tutorials, lab exercises and similar resources.
- The MQSeries product family website at http://www.ibm.com/software/ts/mqseries/.

  By following links from this website you can:
  - Obtain the latest information about the MQSeries product family, including the MQSeries Adapter Offering.

- Access the MQSeries books in HTML and PDF formats, potentially including a more recent edition of this book.
- Download MQSeries SupportPacs.

Summary of changes

The third edition (the current edition) includes the following additions and changes from the second edition:

- Changed the term connector flow to microflow.
- Updates to the build time and run time information. See “Build time and run time” on page 3.
- Added “Using adapters during run time” on page 6.
- Added information about Java importers. See “Importing and creating messages” on page 10.
- Added information about generating Java adapters. See “Generating an adapter” on page 19.
- Expanded description of building an adapter by adding content previously in Glossary. See “Procedures for building an adapter” on page 28.
- Updates to the Glossary.

Chapter 1. About MQSeries Adapter Offering

IBM MQSeries Adapter Builder is part of a set of application integration products that together are called IBM MQSeries Adapter Offering.

IBM MQSeries Adapter Offering works with MQSeries messaging to enable you to reduce the risk, complexity and cost of managing the point-to-point integration of your business processes.

In *point-to-point* integration, each application interfaces with all the other applications, individually. Each interface is different and there are many different interfaces. One change in one application typically requires changes to many interfaces. As the number of applications increases, point-to-point integration becomes rapidly less cost-effective. Adding each new application typically requires more work than adding the most recent-added application.

With MQSeries Adapter Offering, you can evolve from using point-to-point integration to *one-to-any* integration.

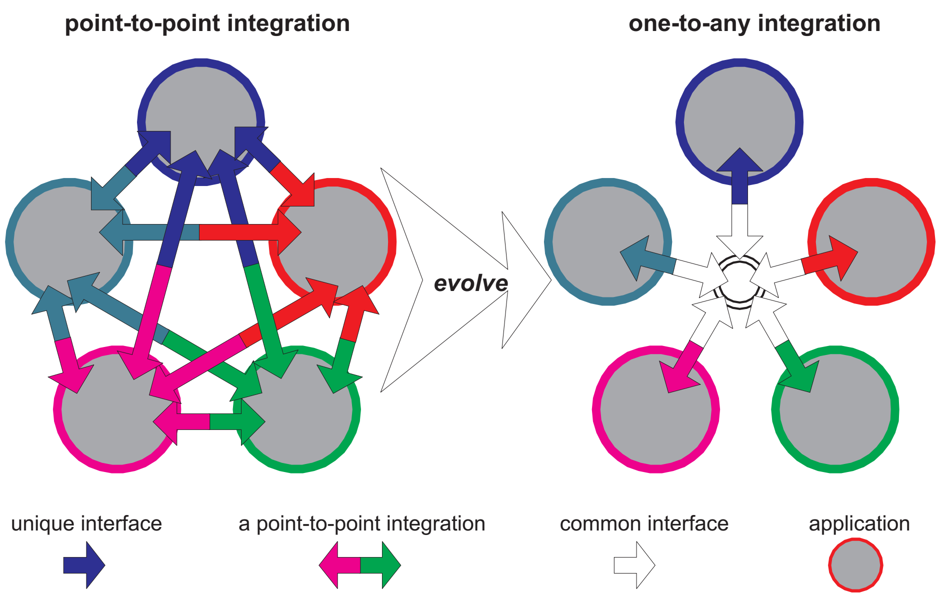


Figure 1. Evolution from point-to-point integration to one-to-any integration

On the left side of Figure 1, in point-to-point integration, each of five applications each integrate individually and separately with each of the other applications. There are 10 distinct integrations. On the right side, in one-to-any integration, each of the five applications integrate to one common interface in the center. In this case, there are five integrations.

MQSeries Adapter Offering supports both kinds of integration and facilitates evolution toward one-to-any integration.

MQSeries Adapter Offering supports integration of:
- C language based applications.
- Java language based applications
- Other applications, if used in conjunction with user-written code that wraps an applications non-C interface, for example, RPG for the AS/400 and file import/export, with a C or Java language interface.

Benefits of MQSeries Adapter Offering

MQSeries Adapter Offering has several benefits:
- All applications can use one common interface.
- Data from a *source application*, in the form of a *message*, is *routed* to one or more *target applications*.
- A change in one application typically affects only that one interface.
- Using a common interface that is application-neutral, such as an industry standard, can be even more cost-effective. More applications can be supported with less effort.
- As the number of applications increase, one-to-any integration becomes even more cost-effective. Adding each new application typically does not require significant changes to the interfaces to the other applications.
- The MQSeries Adapter Offering can be deployed without changing applications or business processes at all. Typically, all the integration work is performed in MQSeries Adapter Offering.
- Integration work can be facilitated through the use of templates.
- Business processes and each application can remain isolated from the specifics of middleware, message details and other applications. A common interface for messaging enables adding new applications without changing existing applications or business processes.
- MQSeries Adapter Offering can reduce the need to write custom code.

MQSeries Adapter kernel can be deployed in two tiers. One tier is the source side of the run time; the other tier is the target side of the run time. Two tier deployment can provide more efficient operation and less administrative overhead. A third tier for routing and delivery is not required to reside between the two sides of the run time.

As an option, MQSeries Integrator can be added to perform brokering, such as complex routing, data transformation and data mediation, as well as message transformation. It would add a third tier.

Adapters

In MQSeries Adapter Offering, the interface to or from one application is provided by an *adapter.* All applications need an adapter to provide the interface between the application environment and the MQSeries environment. Each adapter is specific to an application.

Several examples of adapters are:
- Add a sales order.
- Synchronize a customer record.
- Synchronize an inventory record.
- Synchronize an item.
- Synchronize a sales order.

Build time and run time

The MQSeries Adapter Offering consists of two environments: build time and run time.

build time

Via an intuitive visual interface, the MQSeries Adapter Builder enables you to build an adapter for virtually any application. See Figure 2 on page 4.

MQSeries Adapter Builder consists of several major components:
- Several *importers* that enable you to import an application’s interface into the MQSeries Adapter Builder in the form of messages, where then you can work on it. The importers import:
  - C header files containing function prototypes and structure definitions. Only ANSI C is supported.
  - XML Data Type Definitions (DTDs).
  - XML Data Type Definitions (DTDs) that contain standard data formats known as Business Object Documents (BODs) that have been defined by the Open Applications Group (OAG).
  - JavaBeans that contain either simple Java classes or access beans that encapsulate EJB session and entity beans.
- The *Control Center*, the user interface. It is similar to MQSeries Integrator’s user interface. Via the Control Center, you can:
  - Control the import of messages by the importers.
  - Modify imported messages or manually create new messages.

  - Compose the adapter model. The adapter model can contain a wide variety of functionalities, such as control flow, data flow, sequential navigation, conditional branching including decision and iteration, data typing, storing data context, transformation of data elements (known as mapping), logical operations and custom code.
  - Store your work in the repository.
  - Control the generation of adapter code by the generator.
- The *repository*, a directory in the builder's file system that contains the messages and adapter models.
- The *generator* that generates adapter code. The following platforms are supported:
  - Microsoft Windows NT 4/2000
  - IBM AIX v4.3.x
  - IBM OS/400 v4r4/5
  - HP-UX v11
  - Sun Solaris 7/8.

An ANSI C generator produces C code for source adapters or target adapters. The generated C code for source adapters or target adapters includes C header, C source code, make files for each platform, export file and Java wrapper.

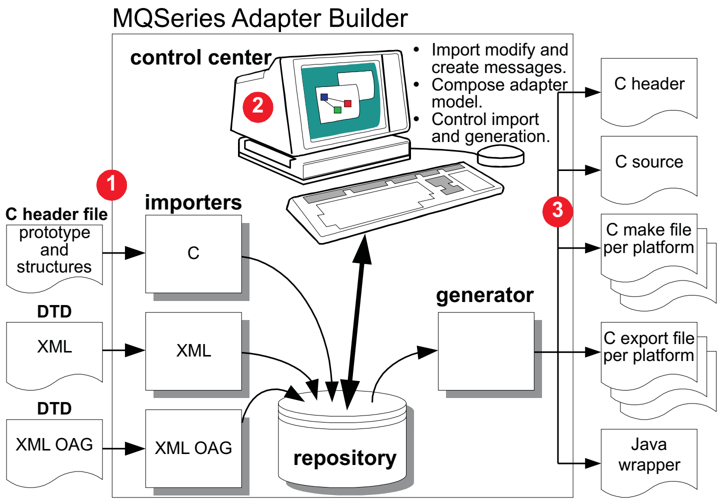


Figure 2. Components of the MQSeries Adapter Builder (ANSI C generator)

A Java generator produces code for a Java service adapter. You can move the adapter code to the desired platform to compile it and then test the adapter. The compiled Java service adapter may include Java classes (microflow, data structure, terminal data container, and mapper classes), a service bean or both.

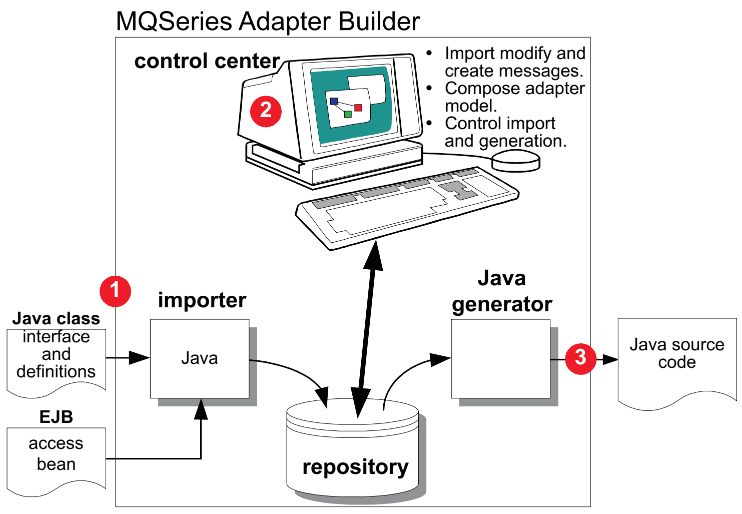


Figure 3. Components of the MQSeries Adapter Builder (Java generator)

run time

adapter

The output of the MQSeries Adapter Builder. An adapter provides the interface to an application or from an application. Typically, you build each adapter to be specific to *one message type* that is sent from or to an application. Thus, the adapters themselves are not part of MQSeries Adapter Offering.

An adapter consists of C source code that compiles to a shared library or Java source code that compiles to a class file. When the adapters and the MQSeries Adapter Kernel execute together at run time, they perform the run time functionality of the MQSeries Adapter Offering.

Depending on how you modeled it in the MQSeries Adapter Builder, the adapter can contain a wide variety of functionalities.

You may reuse adapters that you have created.

The ANSI C code generator can produce two types of adapters:

- Source adapters, for the application that sends the data.

- Target adapters, for the application that receives the data.

The Java code generator produces a Java service adapter.

- Java service adapters, provides services for the implementation of a business process flow.

An adapter is required for each message type. See "adapter" on page 55 for additional information, including a discussion of how many adapters are required in an example site.

MQSeries Adapter Kernel
A set of application programming interfaces (APIs), several executable programs in C and Java, and several configuration files. The kernel enables the deployment and execution of the adapters. In addition to directly supporting adapters, the kernel performs related functions, including simple routing of messages and infrastructure services such as message construction, transactional control, tracing, and interfacing with MQSeries or other messaging software.

The kernel is installed on each computer on which a source adapter or a target adapter run.

MQSeries Adapter Offering can be complemented by service offerings from IBM and others.

Except where specified, the rest of this information is about the MQSeries Adapter Builder only. For detailed information about the MQSeries Adapter Kernel, see the dedicated publications for that product.

Using adapters during run time

This section provides an overview of the adapter run time processing. The first example shows how source and target adapters relate when deployed in a run time environment. The second example shows the run time processing for a Java service adapter. See the MQSeries Adapter Kernel documentation for the details.

Source and target adapters

The MQSeries Adapter Offering run time includes the kernel and the source and target adapters you build. Its purpose is to (see Figure 4 on page 7):

- Transfer data from one application, the source, to another application, the target.
- Convert the source application's data to a message, typically in an application neutral format, that is trafficked through the kernel, via MQSeries.

- Route the message to the target application.
- Determine how to get the data to the target application.
- Convert the data from the format of the message that is trafficked through the kernel through an adapter to the target application's format.

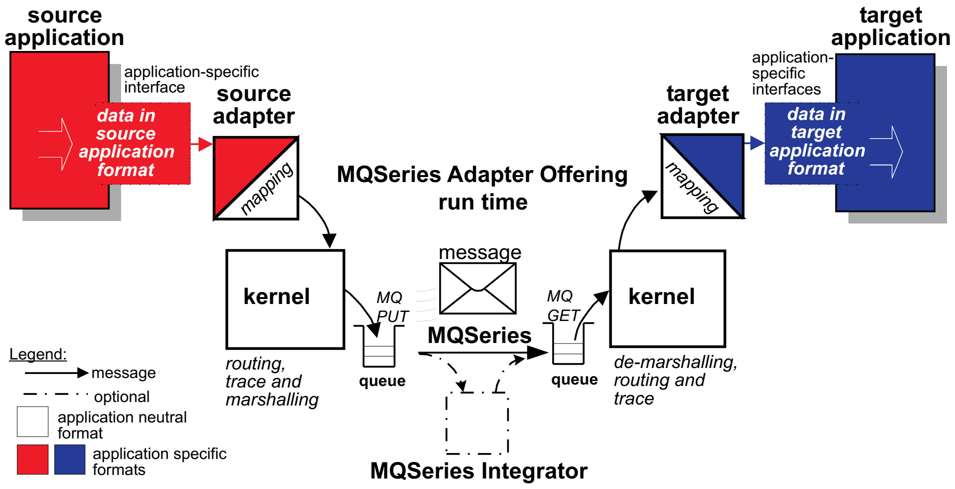

source application
application-specific interface
data in source application format
source adapter
mapping
MQSeries Adapter Offering run time
target adapter
mapping
application-specific interfaces
data in target application format
target application
kernel
routing, trace and marshalling
MQ PUT
queue
message
MQSeries
MQ GET
queue
kernel
de-marshalling, routing and trace
MQSeries Integrator
Legend:
message
optional
application neutral format
application specific formats

Figure 4. Run time deployment of source and target adapters

Java service adapters

The MQSeries Adapter Offering run time includes the kernel and the Java service adapters (service beans) you build. A component called the Java Message Service (JMS) listener works with the MQSeries Adapter Kernel and WebSphere Application Server Advanced Edition to provide an alternative way of delivering messages to target applications. An overview of the run time processing is (see Figure 5 on page 8):

- A Java Message Service (JMS) listener within WebSphere will receive a JMS message and pass the message to a Worker Message bean (session bean).
- The Worker Message bean will convert the JMS message into a kernel holder object.
- Based on the kernel header values the Worker Message bean will:
  - Convert the application data into an appropriate input data structure.
  - Determine the Service Session bean to invoke and will pass the data object to a specific method on the Service Session bean.
- The Worker Message bean converts any replies into a JMS Message object and sends the reply.
- In the case of errors, the Worker Message bean will put the JMS Message object into an error queue.

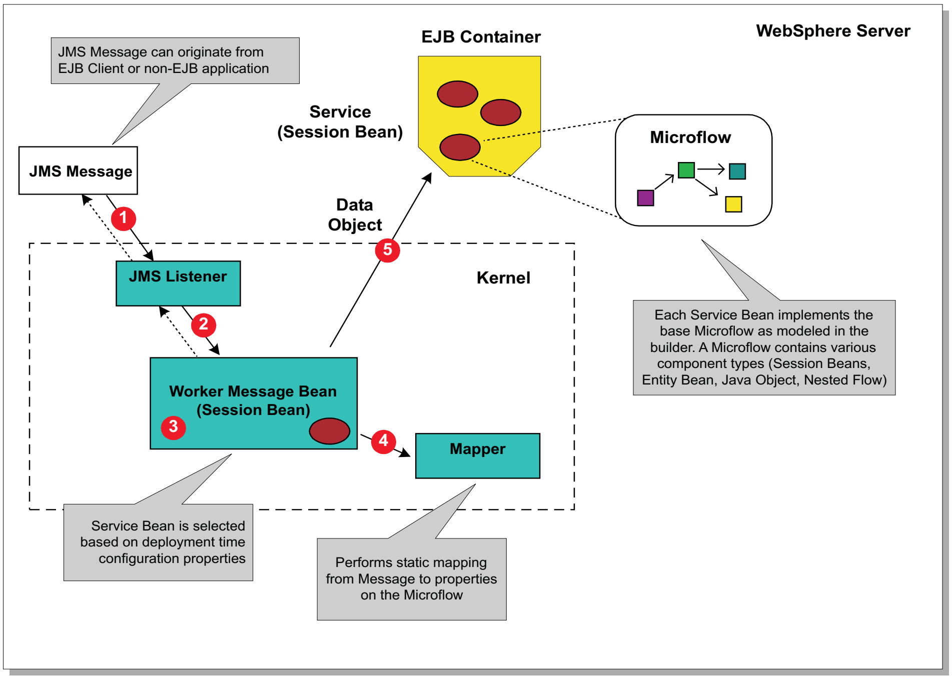

WebSphere Server
EJB Container
JMS Message can originate from EJB Client or non-EJB application
Service (Session Bean)
Microflow
JMS Message
1
Data Object
5
JMS Listener
Kernel
2
Each Service Bean implements the base Microflow as modeled in the builder. A Microflow contains various component types (Session Beans, Entity Bean, Java Object, Nested Flow)
Worker Message Bean (Session Bean)
3
4
Mapper
Service Bean is selected based on deployment time configuration properties
Performs static mapping from Message to properties on the Microflow

Figure 5. Run time deployment of Java service adapter

Chapter 2. About the builder

The builder has several benefits and capabilities.

Benefits of the builder

The MQSeries Adapter Builder provides several major benefits:

- It allows you to concentrate on the application domain rather than on low level programming.
- It provides a graphical display of the adapter during build time to show the processing flow within the adapter.
- Because of the builder's structured approach, you can use the same methodology to build a variety of different adapter types.
- You can build an adapter once and deploy it on multiple, different platforms.
- The Adapter Builder's user interface is based on MQSeries Integrator, version 2.0. Therefore, if you know how to use MQSeries Integrator's Control Center, learning to use MQSeries Adapter Builder is relatively easy.
- Adapters can be easily created, modified and regenerated.
- Adapter maintenance is simplified.
- The generated adapter includes tracing that can aid the test and debug of the adapter during run time.

The Control Center

The Control Center is the user interface of the MQSeries Adapter Builder. You use it to:

- Import messages using the DTD, OAG DTD, ANSI C, and JavaBean importers. See "Importing and creating messages" on page 10 for details.
- Modify imported messages or manually create new messages. See "Importing and creating messages" on page 10 for details.
- Compose the adapter model. The adapter model can contain a wide variety of functionalities, such as control flow, data flow, sequential navigation, conditional branching including decision and iteration, data typing, storing data context, transformation of data elements (known as mapping), logical operations and custom code.
- Store your work in the repository.
- Generate adapter code.

The Control Center has two different views:

Message Sets view
Where you define the different components of a message set. You can import messages, modify imported messages or manually create new messages.

Adapters view
Where you define and model the adapter's functionality. You create microflows by:

- Connecting various microflow nodes with connections.
- Setting various properties of microflow nodes and of connections.

Importing and creating messages

The MQSeries Adapter Builder contains several *importers* that enable you to import an application's interface into the MQSeries Adapter Builder in the form of messages, where then you can work on it. The importers are:

- *ANSI C importer* imports ANSI standard compliant C header files containing function prototypes and structure definitions. It supports:
  - Preprocessing of the header files, such as #includes, #defines, and so forth.
  - User definitions of macros.
  - User specification of include path.
  - User selection of the APIs and structures to be stored in the repository.

  Note: The ANSI C importer can only import APIs. To import structures, you must create an API that refers to the structures, and then import that API to get the structure definitions into the builder.
- *XML importer* imports XML Data Type Definitions (DTDs).
- *OAG XML importer* imports XML Data Type Definitions (DTDs) that contain standard data formats known as Business Object Documents (BODs) that have been defined by the Open Applications Group (OAG).

  OAG has encoded valid values for Segment qualifiers and types by creating parameter entities that consist of the segment name, the qualifier name, and type name. The OAG XML importer recognizes these special entities and adds the qualifier and type values as element qualifiers for the imported messages.
- *JavaBean importer* imports Java class definitions by using bean introspection. It supports:
  - Java classes, including the interface and parameter definitions.
  - Access beans that encapsulate session and entity beans.

After importing a message, you can modify and store it.

Note: You can re-import as a way to add messages (XML/OAG elements, C APIs, or Java methods) to a message set that were missed or forgotten on initial import. However, if you have made any changes or deletions to messages in the builder after the initial import, then the re-import will overwrite these modifications.

You use the builder's Message Sets view to import and modify messages.

In addition to importing a message, you can choose to create your own message via the builder's Message Sets view. See the online help system for details of the builder's capabilities in creating messages.

The following message sets are supplied with the builder:
- Base Java — required if you are doing iteration in Java.
- MQAO Kernel — required for building target adapters in C.
- MQAO Kernel Java — required to add class types to the workspace if you are building Java service adapters.

To access the system-supplied message sets, you must add them to your workspace.

Composing microflows

You can define and model the adapter's functionality in the builder's Adapters view.

The Adapters view is split into two panes:

Adapter Tree View
Lists adapters and components, such as Microflow types and Command types.

Microflow Definition
Composition pane where you define microflows graphically. This is called modeling.

The Adapter Tree View displays *types*. A type represents a template that can be used as a building block in modeling the microflow. When you *drag* a type onto the Microflow Definition pane, it is *instantiated* and referred to as a *microflow node*. A single type can be used to create one or more microflow nodes (instances) as part of the same microflow. In the Microflow Definition pane, you connect the microflow nodes manually by means of *connections* and set their *properties.* The connections that you see in the Microflow Definition pane show you how *control* and *data* will flow through the microflow.

See “Examples of composing microflows” on page 14 for example figures that illustrate how microflows are composed.

A microflow and associated terms are defined:

microflow

A directed graph that models the processing of a message as it passes *within* one adapter, from the input of the adapter to the output of the adapter:

- Within the source adapter, the input terminal of the adapter is where the message is received from the source application. The output terminal of the adapter is where information may be returned to the source adapter. Internally, within the adapter, messages may be created, transformed and handed to the kernel to be put on the MQSeries queue.
- Within the target adapter, the output terminal of the adapter is where a reply message may be sent back into the kernel. Internally within the adapter, messages may be created and transformed and sent to a target application.

Microflows can be nested. In this case, the most outermost nesting microflow adheres to the standard definition for the microflow, but the nested microflows do not. Instead of interacting directly with the source or target adapters, a nested microflow gets a message from a nesting microflow, processes it, and then passes it back to the nesting microflow.

The microflow is the model describing the logic which is implemented in the code that gets generated for the adapter. A microflow consists of a set of microflow nodes and the connections that connect them.

microflow node

The generic term for any node in the microflow. Each type of microflow node represents a well defined processing stage. A set of microflow nodes is provided with the builder.

One of the microflow components that is provided with the builder is called “Microflow”. It models the adapter. All of the other microflow nodes and connectors that can potentially be used in a microflow are modeled within the Microflow node.

connection

A wire that connects an output terminal of one microflow node to the input terminal of another. There are two types of connections: control connection and data connection.

control connection

A connection that provides the logical relationship between two nodes in a microflow. Control and data are passed from one node to the next via the control connection.

data connection
A connection that establishes a relationship between the data of two nodes in a microflow. It is used when the data is coming from anywhere except the node from which control is being passed. It can be used to support mapping.

Examples of composing microflows

Let’s look at some examples of composing microflows.

Figure 6 shows the builder’s Adapters view. You drag a type from the collections of types in the Adapter Tree View on the left, to the Microflow Definition Pane, where it becomes instantiated as a node.

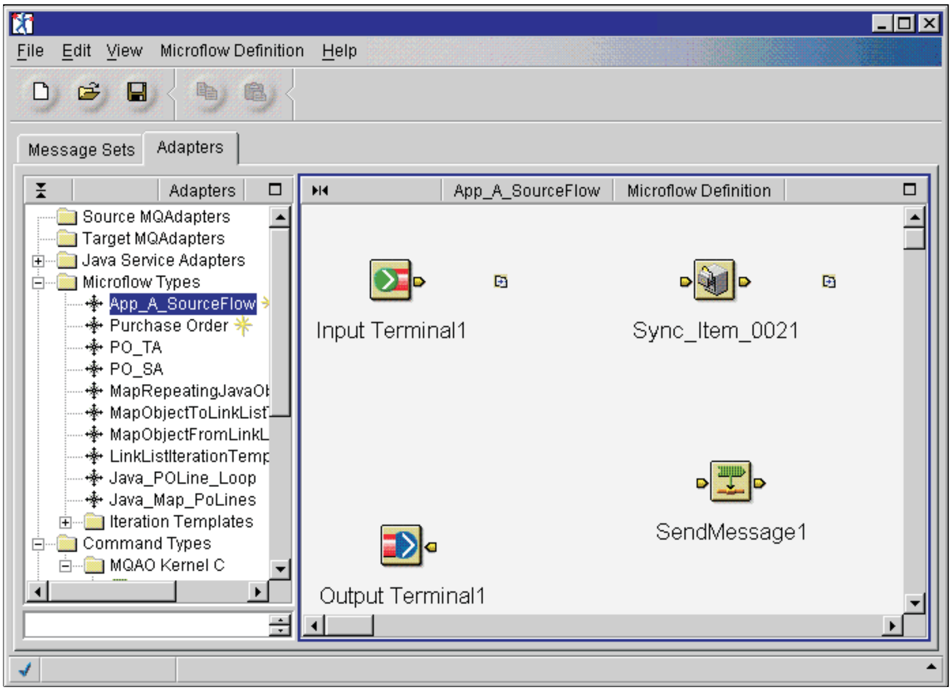


Figure 6. Example of dragging types to instantiate nodes

See the Adapter Tree View on the left:

- Source MQAdapters, Target MQAdapters and Java Service Adapters are folders that contain the adapters that you have already built.
- Microflow Types is a folder that contains the microflows that you have modeled. The microflow that is being modeled in the figure is called “App_A_SourceFlow”.
- The other items are collections of components that are provided with the builder. See “Microflow components” on page 16 for details.

In Figure 6, several nodes have been instantiated but their properties are not yet set and they are not yet connected:

- You right click on the node, select **Properties** and set the properties. Each kind of node has different properties. See the online help system for details. To perform transformation of data elements, known as mapping, you set properties on a Map node.
- You connect nodes by holding down the left mouse button while the cursor is over an outTerminal of a node and dragging a connecting line to the inTerminal of a second node.

Figure 7 shows a Send Message Command node and an Output Terminal node connected by a control connection.

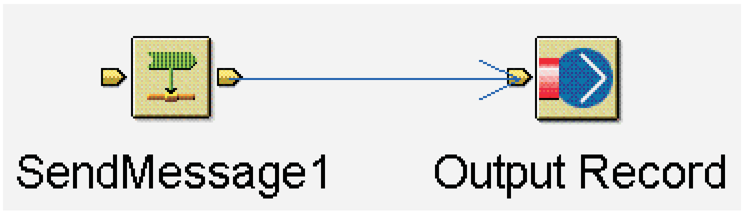


Figure 7. Example of control connection, Send Message Command node and Output Terminal node

Figure 8 shows a Decision node and a Data Context node connected by a data connection.

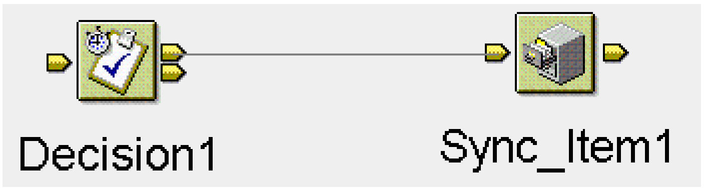


Figure 8. Example of data connection, Decision node and Data Context node

Note that the control connection and the data connection look different.

See Figure 9 for an example of a completed, simple microflow in the builder's Adapters view, Microflow Definition Pane.

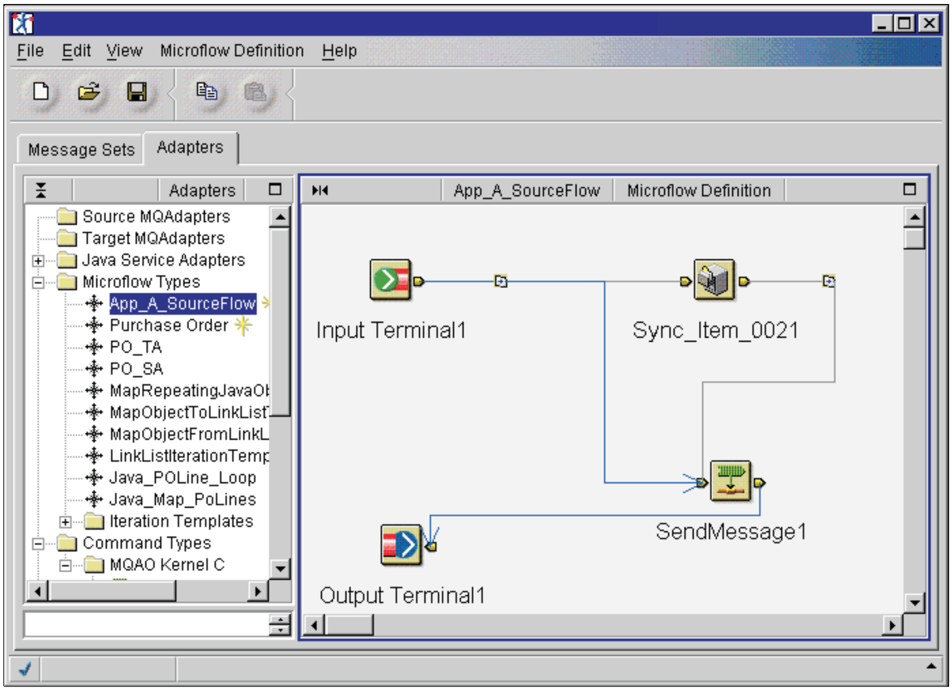


Figure 9. Example of completed, simple microflow in the builder's Adapters view

Once created and saved, a microflow becomes a type on the Adapter Tree View. Thus you can re-use microflows. You can nest a microflow node (that you have created previously) within other microflows.

At this point, you can *create* the adapter by giving the adapter a name and then associating the microflow with the adapter.

Once the microflow has been defined, you *generate* the adapter's code. See "Generating an adapter" on page 19.

Microflow components

The types of microflow components that are supplied with the builder come in several categories:

- Simple: Components that do not consist of other components.
- Primitive: Simple components that are opaque. Their inputs and outputs are visible to the user but their internals are not visible.
- Composed: Components that consist of other components that are connected by control connections.

Several types of microflow components are supplied with the builder.

Table 1. Microflow components that are supplied with the builder

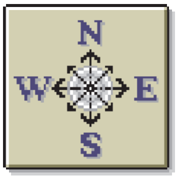
Icon	**Name**	**Description**
	Microflow node	A composed component that models the adapter. All of the other microflow nodes and connectors that can potentially be used in a microflow are modeled within the Microflow node.
	Class node	A simple component that represents a class and its associated methods.
	Command node	A simple component that is used to represent application APIs that you import into the builder.
	Command node	A simple component that is used to represent the supplied MQAO Kernel commands. The builder provides commands that are used to send messages from a source adapter to the kernel which sends it to the target adapter, and as an option, wait for a response. The builder also supplies begin, commit, and rollback commands that support transaction processing within the kernel.

Table 1. Microflow components that are supplied with the builder (continued)

Icon	Name	Description
	Data Context node	A simple component that is used to store data for later access via a data control connector.
	Decision node	A composed component that is used to test a condition for true or false, to resolve the control flow path.
	Iteration node	A simple component that is used to access repeatable patterns of logic or mapping or both. It allows part of the microflow to repeat at run time. The repetition can be based on user variables or data values that occur during run time. It consists of a loop condition that can be tested for true or false, while or until.
	Map node	A composed component that models data transformation. Data transformation can include: • Associating a field in one message with a field in another message. • String mapping such as specifying pad characters. • Date mapping, such as converting a date in one format to a date in another format. • Putting literal data into messages. • Adding custom code to perform other data transformation functions.

Table 1. Microflow components that are supplied with the builder *(continued)*

Icon	Name	Description
	Input Terminal node	A primitive component that is used to represent data types that are input to a microflow.
	Output Terminal node	A primitive component that is used to represent data types that are output from a microflow.
	Error Terminal node	A primitive component that represents an error output terminal for a microflow. If a run time exception is thrown during the invocation of a method and that exception is not defined in the method's signature, then control is passed to the Error Terminal. In addition, if a method has an unconnected error terminal (red-colored output terminal) and an exception is thrown, then control is passed to the Error Terminal node.

There are two types of Command nodes; each has a different icon.

The control connection and the data connection are primitive components that are also supplied with the builder. They are defined in "Composing microflows" on page 11.

Generating an adapter

After the messages have been imported or created, the microflow has been modeled and the adapter has been associated with the microflow, then the adapter can be generated.

The ANSI C generator generates the following adapter code: C header (ANSI standard compliant), C source code (ANSI standard compliant), C make file for each platform, C export file for each platform and, for target adapters, a Java wrapper.

The Java generator generates a number of Java source files representing the microflow and its types. During generation the following types may be produced:

- Microflow classes
- Datastructure classes
- TerminalDataContainer classes
- Mapper classes
- Service Bean
  - Bean implementation
  - Remote interface
  - Home interface
  - JetAce.xml input file (input to the WebSphere JetAce tool used to create a deployable jar file)
  - Sample Config file (.xml file used as a sample for configuring the kernel)

The final step of the generation produces the Java source files. The Java source files must be compiled into class files and can be packaged into a jar file for run time deployment.

Five platforms are supported:

- Microsoft Windows NT 4/2000
- IBM AIX v4.3.x
- IBM OS/400 v4r5
- HP-UX v11
- Sun Solaris 7/8.

Before generation, the builder checks for:

- Forward references
- Dead branches

Before you can use the generated adapter, you must:

- Move adapter source files to a test system and compile and test.
- Move the compiled adapter to a production system, configure the kernel appropriately if required and, if you are satisfied with the adapter that you built, put it into production.

See “Deploy an adapter” on page 30.

Chapter 3. Planning to install the builder

After you read this chapter, you should follow the installation checklist in “Chapter 4. Installing the builder” on page 25.

This chapter lists the prerequisites for and components of the MQSeries Adapter Builder.

For latest details, see the MQSeries product family Web site at:

http://www.ibm.com/software/ts/mqseries/

IBM reserves the right to update the information shown here. For the latest information regarding levels of supported software, refer to:

http://www.ibm.com/software/ts/mqseries/platforms/supported.html

Hardware

The MQSeries Adapter Builder is a specialized graphical program that has demanding requirements on the operating environment. The following setup is recommended for systems on which it is installed:

- Any Year 2000 compliant Intel Pentium III (or above) processor-based IBM PC machine or compatible, with 500+MHz processor speed.
- A minimum of 256 Megabytes (MB) of RAM.

MQSeries Adapter Builder requires a minimum of approximately 40 megabytes (MB) of disk space for product code and data, using NTFS. Online documentation requires 10 MB.

You should also allow disk space for the adapters that you build. Their size is highly dependent on the size of the data structures, on the complexity of mappings and on the custom code used.

In addition, allow a minimum of 20 MB for working space for the builder. Working space requirements can vary based on a number of factors, such as the size of message sets, the amount and complexity of data mapping, and the size and complexity of the microflow that you compose. In addition, you should allow working space for your adapters.

Software

The prerequisites for the builder and for the adapters that you build with it are different.

Planning to install the builder

Builder

The supported level for the builder is shown.

- Microsoft Windows NT Version 4, Service Pack 5.

You can determine the version and service pack of Windows NT. On the Windows NT **Task Manager**, select **Help > About Task Manager**.

The builder's installation program includes all other software needed to run the builder.

Note: MQSeries Adapter Offering uses Java run time environment (JRE) 1.2.2 which is included with the installation program.

Adapters

There are different requirements for compiling, running and testing the adapters that you build.

- The builder generates adapter code that, after compilation, can run on:
  - Microsoft Windows NT 4/2000
  - IBM AIX v4.3.x
  - IBM OS/400 v4r4/5
  - HP-UX v11
  - Sun Solaris 7/8.
- Each platform on which you will compile the adapters that you built requires a compiler. The following compilers are supported:
  - For Windows systems:
    - Microsoft Visual C++ 6.0 Compiler.
    - Java Development Kit version 1.2.2.
  - For AIX:
    - IBM C Set++ for AIX version 3.1.3.
    - Java Development Kit version 1.2.2.
  - For HP-UX:
    - HP-UX C/ANSI C Compiler. See the `readme.txt` for details.
    - Java Development Kit version 1.2.2.
  - For Solaris:
    - Sun Workshop Compilers C/C++. See the `readme.txt` for details.
    - Java Development Kit version 1.2.2.
  - For OS/400:
    - Integrated Language Environment C for AS/400.
    - Java Toolkit and Development Kit version 1.2.2.

- Testing and running your adapters also require the MQSeries Adapter Kernel to be installed and properly configured on each computer on which an adapter runs. Refer to MQSeries Adapter Kernel documentation for details.

Year 2000 statement

MQSeries Adapter Builder, when used in accordance with its associated documentation, is capable of correctly processing, providing, and/or receiving date information within and between the twentieth and twenty-first centuries, provided that all products (for example, hardware, software, and firmware) used with this IBM program properly exchange accurate date information with it.

Customers should contact third party owners or vendors regarding the readiness status of their products.

For the latest IBM statement regarding Year 2000 readiness, refer to:

http://www.ibm.com/ibm/year2000

Components of the builder

After installation, MQSeries Adapter Builder resides in its root directory, default name `IBM MQSeries Adapter Builder v1.0.1`. It contains several directories which in turn can contain other directories. The root and its directories are listed, along with a summary of the files that are most relevant to installation.

root

- `mqab.jar` file that contains:
  - The Control Center executables.
  - The generator.
- `install.log` file and `uninstall.log` file. These are created only if a problem occurs during installation or uninstalling.
- All other MQSeries Adapter Builder directories.
- `readme.txt` file.

bin

- Importer DLLs.

com

- Properties files for national language support.

documentation

- HTML and PDF versions of manuals available in the Information Center.

Help

- Help system.

images

- Images used by the Control Center.

jre

- Java Runtime Environment.

license

- International program license agreements.

Register

- Contains program registration information.

repository

The builder repository that contains:

- export directory — contains exported workspace files.
- working directory — contains the workspace files.
- message sets, microflows that you have modeled, adapters that you have built, component types that are provided with the builder and everything that they refer to, such as primitives.

Chapter 4. Installing the builder

Prepare for installation

You must have administrator authority in order for you to install MQSeries Adapter Builder.

You must have permission to create and access files in the location where you install MQSeries Adapter Builder. Ensure that all user IDs that will run the builder have read, write and execute permission.

You do not have to install the kernel in order to install the MQSeries Adapter Builder or to use it to build your adapters. You do need the kernel to compile, test and deploy your adapters.

Install

Use these procedures:

__ Step 1. Read the `readme.txt` file on the CD-ROM or local area network first. It contains important information that became available after this book was completed. It is located in the same directory as `setup.exe`.

__ Step 2. You can choose to visit the MQSeries website at http://www.ibm.com/software/ts/mqseries/. It might contain important information that became available after this book was published, possibly including a new edition of this book.

__ Step 3. Check that the prerequisites have been installed. See “Chapter 3. Planning to install the builder” on page 21.

__ Step 4. Check that you have prepared for installation, for example, that you are authorized to install the builder. See “Prepare for installation”.

__ Step 5. You do not have to copy `setup.exe` to another location before you double click it. During the installation process, you are asked to choose where to install MQSeries Adapter Builder:

- If you are installing from a local area network, double click on `setup.exe` on the local area network to start the process.
- If you are installing from a CD-ROM, insert the MQSeries Adapter Builder for Windows NT CD-ROM into the CD-ROM drive.

Installing the builder

- If autorun is enabled, the installation process starts. If it is not, double click on `setup.exe` in the root directory on the CD-ROM to start the installation process.

__ Step 6. Follow the actions described in the InstallShield Wizard screens that are presented to you.

Note: If the installation process is interrupted, you should remove the builder (see "Remove the builder"), and then run the installation process again.

__ Step 7. The builder is installed and ready to use.

__ Step 8. Start the builder by selecting its menu item under the MQSeries Adapter Builder root menu from the **Start** menu.

Remove the builder

There are two ways that you can remove the builder:

- You can the run the MQSeries Adapter Builder install.exe program. The installation program detects that you have MQSeries Adapter Builder installed on your system and then gives you the option to **Remove** the Builder.
- Alternately, use the Add/Remove Programs utility in the Control Panel.

Chapter 5. Build an adapter

Adapters are modeled and built in the Control Center. Before you model and build an adapter, you must understand:

- The MQSeries Adapter Kernel. Adapters work with the kernel. See the MQSeries Adapter Kernel documentation. Several items in the kernel are of particular importance when you build an adapter. You must coordinate these items in the kernel with the associated items in the adapter that you build:
  - MQSeries Adapter Kernel commands
  - *Message control values* in messages that:
    1. The kernel uses to control the marshalling and routing of messages during run time.
    2. Each adapter uses to control, in part, how it performs its functionality.

    Message control values include:
    - Source logical identifier
    - Destination logical identifier
    - Respond to logical identifier
    - Body category
    - Body type
    - Acknowledgment requested

    You set these values in the Command node. See the online help system for details.
  - In the kernel, the *application identifier* of the source application or target application with which the adapter is associated.

    You set this value in the **Application Name** field when you generate the adapter. See the online help system for details.
- The source application or target application for which you are building the adapter. Each adapter is specific to an application.

The general procedure for building an adapter is as follows:

1. Import or create messages.
2. Compose a microflow.
3. Perform data mapping, in the microflow.
4. Create an adapter and associate a microflow with it.

5. Generate adapter code.
6. Move adapter source files to a test system and compile and test.
7. Move the compiled adapter to a production system, configure the kernel appropriately if required and, if you are satisfied with the adapter that you built, put it into production.

See the online help system for details.

Start the builder

To start the builder, you click its menu item under the MQSeries Adapter Builder root menu on the **Start** menu.

Procedures for building an adapter

The adapter is built in two complementary stages:

- The structures of messages are imported or created and maintained in the form of message sets, in the Message Set view. See “message set” on page 60.
- The processing of messages is modeled in the form of a microflow in the Adapters view. See “microflow” on page 60.

To send one type of message from one application to a second application typically requires one source adapter and one target adapter. If the second application must send one type of message to the first application, another source adapter and another target adapter are required. Thus, in this case, in order to

Send one type of message from the first application to the second application and

Then to send another type of message from the second application back to the first application,

four adapters are typically deployed.

Note: There is also the possibility of a *hybrid* adapter. This is a target adapter that receives a message, but also creates a new message to send to an application.

For example, assume that a site consists of applications A, B and C and that they all are different, that is, no two applications, in the absence of the kernel, support the same message types:

1. A sends 4 message types to B and 2 messages types to C. A is acting as the source application and B and C are acting as the target applications. You would build 6 source adapters for A and 6 target adapters, 4 for B and 2 for C.

2. B sends 3 message types to A and no messages to C. B is acting as the source application and only A is acting as the target application. You would build 3 source adapters for B and 3 target adapters for A.
3. C sends 2 message types to A and 1 message type to B. C is acting as the source application and A and B are acting as the target applications. You would build 3 source adapters for C and 2 target adapters for A and 1 target adapter for B. Because the applications all are different, that is, no two applications, in the absence of the kernel, support the same message types, you could not reuse adapters.

This site would require a total of 12 source adapters and 12 target adapters. A would be supported by 6 source adapters and 5 target adapters. B would supported by 3 target adapters and 5 target adapters. C would be supported by 3 source adapters and 2 target adapters.

Assume that, in the absence of the kernel, two of the applications were identical in terms of the message types that they support. In this case, you might not need to build as many adapters. You might be able to reuse some adapters for the two applications.

The detailed procedures for building an adapter are provided in the builder's online help system. See the online help system.

Generate an adapter

If you have modeled a C adapter (either a source adapter or target adapter), then the ANSI C generator produces C code for the source or target adapter. A C adapter is an MQAK Command Interface adapter.

If you have modeled a Java adapter, then the Java generator produces Java code. You have a choice of generating:

- an EJB Interface adapter, for deployment in an application server
- an MQAK Command Interface adapter, for running with the kernel daemon in command line mode
- a Simple Interface adapter, an adapter that will not need to interface to the MQAO Kernel

These three types of adapters require different procedures in order to deploy, test and put into production. An EJB Interface adapter is an adapter that will be deployed in an application server, like IBM's WebSphere Application Server. An EJB Interface adapter can be deployed on a different machine from where the kernel is running. An MQAK Command Interface adapter is an adapter that must reside on the same machine as the kernel.

Build an adapter

To compile an EJB Interface adapter ensure you have the following in your `CLASSPATH` environment variable:

- MQAOJFramework.jar
- <INSTALL_DIRECTORY>\WebSphere\Appserver\lib\ujc\lib
- <INSTALL_DIRECTORY>\mqak\bin
- Destination directory for generated code and adapter

In addition to the normal Java files used for an adapter, the builder generates a test class for an EJB Interface adapter. The test class can be used to test the EJB Interface adapter without the need for deploying it to an application server. The test class is located in the same directory as your bean classes (generate path + Adapter Name + package name) and has the name of your bean + ″Test″.

Once your classes have been compiled, run the test class with the following command from a command prompt:

```
java packageName.BeanNameTest
```

where packageName is the package name you specified for your adapter and BeanName is the Bean Name you specified for your adapter in the Generator window. If the test class is invoked with no parameters, a list of possible parameters will be displayed.

Deploy an adapter

After you generate and then compile an adapter, you must deploy it in order to test it and to put it into production. This section summarizes the procedures for deploying an MQAK Command Interface adapter and an EJB Interface adapter. There is also a description for a Simple Interface adapter which is not deployed with the MQAO Kernel or as a EJB.

MQAK Command Interface adapter

Follow this procedure to deploy an MQAK Command Interface adapter for either test or production:

__ Step 1. Keep a backup copy of your compiled adapter in a safe location in case you need to recover it.

__ Step 2. Move the compiled adapter from the working directory on your computer to the computer on which the kernel is installed, verified and capable of functioning.

__ Step 3. Create a directory for the adapter. To simplify your organization, it is recommended that you keep all your source adapters in one directory and all your target adapters in a different directory.

__ Step 4. You must set environment variables to point to the compiled adapter and to its Java class files:

On Windows NT:

- Add the directory containing the adapter's DLL to the `PATH` system environment variable.
- Add the directory containing the Java class files to the `CLASSPATH` system environment variable.

On AIX:

- Add the directory containing the adapter's library (`libxxx.so`) file to the `LIBPATH` environment variable.
- Add the directory containing the Java class files to the `CLASSPATH` environment variable.

On HP-UX:

- Add the directory containing the adapter's library (`libxxx.sl`) file to the `SHLIB_PATH` environment variable.
- Add the directory containing the Java class files to the `CLASSPATH` environment variable.

On Solaris:

- Add the directory containing the adapter's library (`libxxx.so`) file to the `LD_LIBRARY_PATH` environment variable.
- Add the directory containing the Java class files to the `CLASSPATH` environment variable.

On OS/400:

- Add the library containing the adapter's service program to the library list.
- Add the directory containing the Java class files to the `CLASSPATH` environment variable.

__ Step 5. Configure the kernel to support the adapter. See MQSeries Adapter Kernel documentation.

__ Step 6. Make the changes to environment variables active:

On Windows NT:

- If you have updated the environment variables in an earlier step, restart the computer to make the changes active. (Alternatively, you can just exit any existing MS/DOS command prompts and start a new one.)

On AIX:

- If you have updated the environment variables in an earlier step, log off and then log on to make the changes active (This would only be necessary if you updated the environment variables by making changes to your .profile or .kshrc files.).

On HP-UX:

- If you have updated the environment variables in an earlier step, log off and then log on to make the changes active (This would only be necessary if you updated the environment variables by making changes to your .profile or .kshrc files.).

On Solaris:

- If you have updated the environment variables in an earlier step, log off and then log on to make the changes active (This would only be necessary if you updated the environment variables by making changes to your .profile or .kshrc files.).

On OS/400:

- Exit any existing QSH Command entry windows and start a new one.

__ Step 7. Take appropriate steps on each platform to start the kernel:

- Re-start the process in which the source adapter is run. Note that the source adapter is run in the source application's process. Any daemon or server that contains the source adapter needs to be started. You do not start the source adapter.
- Re-start the adapter daemon.

Follow the instructions for startup in the MQSeries Adapter Kernel documentation.

To modify a compiled adapter that has already been run, you follow the same steps, except that setting environment variables might not be necessary.

EJB Interface adapter

Follow this procedure to deploy an EJB Interface adapter to the WebSphere Application Server for either test or production:

__ Step 1. Ensure all java source files have been compiled.

__ Step 2. Ensure the path where the jetace tool resides is in your path environment variable (`<INSTALL_DIRECTORY>\WebSphere\Appserver\bin`).

__ Step 3. Change to the directory where the xxx_JetAce.xml file was generated (generate path + Adapter Name) where *xxx* is the name of your Adapter bean.

__ Step 4. Invoke the JetAce tool to create a deployable jar file passing it the name of the generated .xml file as input (JetAce `xxx_JetAce.xml`)

__ Step 5. In the WebSphere Application Server, ensure the MQAOJFramework.jar file is in both the Server Node's Dependent classpath and the classpath specified for the Application Server where the EJB Interface adapter will be deployed.

__ Step 6. The classpath specified for the Application Server should also include the jar file created by the jetace tool in step 4.

__ Step 7. Pick a container to deploy the adapter into and right click on that container, select *Create...*and then *Enterprise Bean* from the pop-up menu.

__ Step 8. Use the Browse button to browse to the location where the jar file created by the jetace tool resides and then follow the normal WebSphere operations for deploying a jar file.

__ Step 9. Configure the kernel to support the adapter. A sample configuration file was generated that can be used as an example of what needs to be put into the kernel's configuration.

Simple Interface adapter

A Simple Interface adapter is not deployed in the MQAO Kernel or as an EJB, but rather it is called directly from another Java program. For a Simple Interface adapter, the builder generates a Java class which has an execute*XXX*() method for every input terminal on every microflow included in the adapter, where *XXX* is the name of the microflow.

If an input terminal is typed using a DTD or XML document, then the builder generates two execute*XXX*() methods: one that takes a String as a parameter (the XML document) and another that takes a org.w3c.dom.Node as a parameter. The org.w3c.dom.Node interface is the primary datatype for the entire Document Object Model.

If an input terminal is typed as a Java class, the builder generates one execute method that takes as a parameter an object of the type the input terminal is typed as. If the input terminal is typed as a method of a Java class, then the builder generates an execute*XXX*() method that takes the same number and types of parameters as the method that input terminal is typed with.

You can call into a microflow by simply instantiating the Simple Interface adapter class and invoking the appropriate execute*XXX*() method from within a Java program.

Chapter 6. Manage your workspaces and the repository

You can use the builder to manage your workspaces and the repository. For the most important information on this topic, read:

- "Techniques for managing your workspaces and the repository" on page 44.
- "Tasks: Manage your workspaces and the repository" on page 46.

Workspaces and the repository

A *workspace* is a view of what you can work with at one time. A workspace is displayed as the graphical space in the builder where you build adapters. The workspace's content is:

- Certain message sets, which are displayed in the Message Sets view.
- The microflows that you have modeled (that are associated with the message sets), the adapters that you have built and the component types provided with the builder, which are displayed in the Adapters view.
- Everything that the message sets, microflows and adapters refer to, such as primitives.

See Figure 10 on page 37.

The options available when working with workspaces are:

- Create a new workspace.
- Open a workspace to display and modify its content.
- Save the open workspace, including its content.
- Import and export a workspace, including its content.

The characteristics of each workspace (but not its content) are contained in a workspace file. Workspace files are stored in the *working directory.*

The *contents* of all workspaces are stored in the repository.

The *repository* is a directory in the builder's file system that contains:

- The working directory in which all the workspace files reside.
- The contents of all the workspaces.
- The export directory to which a workspace and its content can be exported.

The repository also contains things that are no longer part of any workspace. For example, a message could appear in the repository and in a single workspace. Even if the message is removed from the workspace, so that it no longer appears in any workspace, it will remain in the repository.

Two versions of each workspace’s content can be stored:

development version
: The workspace content that you can actively work on. You develop on the development version. You can create and open workspaces, import or create messages and model new microflows, modify messages and microflow, and create adapters. You save workspaces, including their content, to protect the work that you have done but before you have completed building the adapter.

committed version
: The workspaces and their content that you have completed work on. You have committed the workspace’s content: the message sets, microflows and the adapters. You can *export* the committed version to an *export file* in the *export directory.* You can make the export file available to enable someone else to use the workspace and its content on their copy of the builder. To here, you can *import* a workspace and its content that had previously been exported to an export file.

  export file
  : A compressed “zip” format file that contains one workspace file and all of the workspace’s content. An export file is self-sufficient as long as all message sets had been in the workspace when it was exported. It contains everything that is required to enable someone to import the workspace and then work on its content “as is”.

    Note: Do not confuse the two types of importing that you can perform:
    - Importing an application’s interface in the form of messages where you can work on it. You select and right-click to import. After the messages have been imported, they are saved as a development version of a message set.
    - Importing a workspace. You perform this import from the **File** menu.

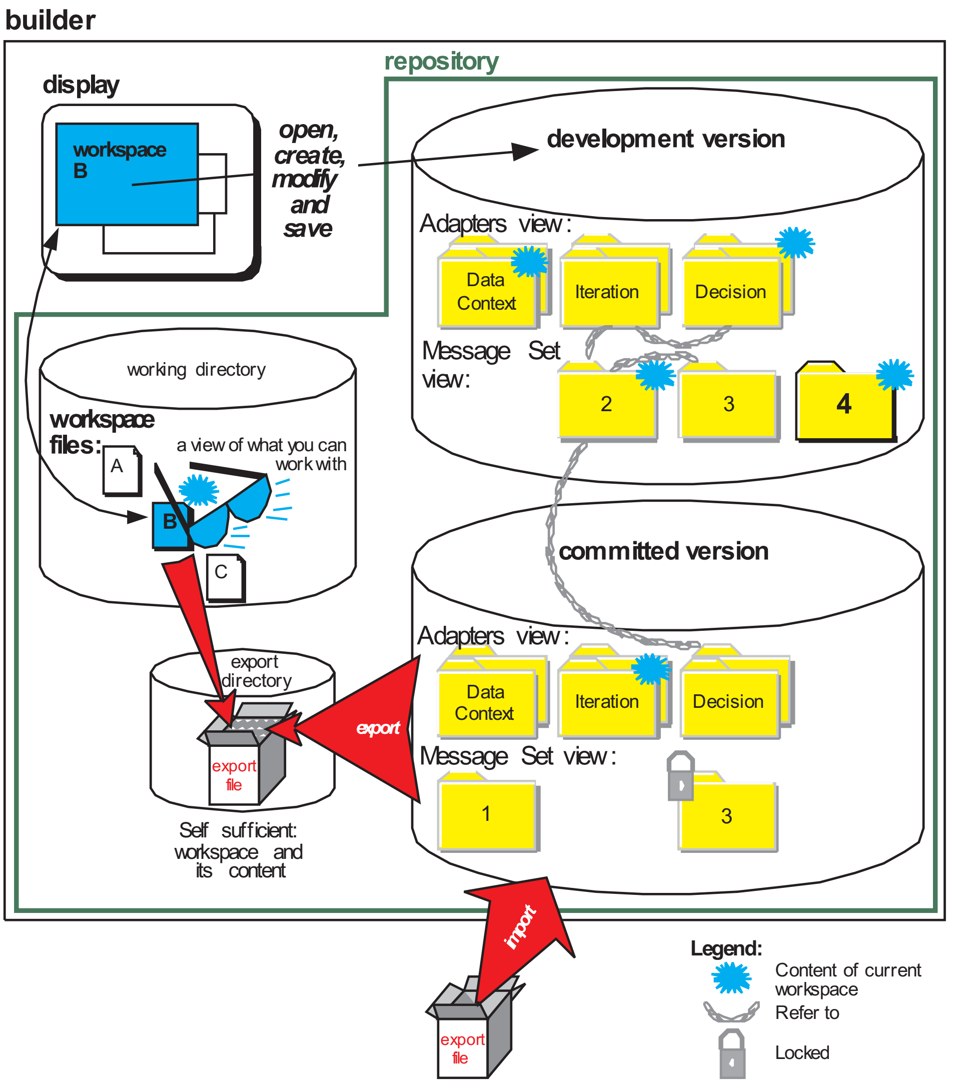
Within the repository:

- Each message set is contained within a directory named by a universal unique identifier.
- The development versions of all workspaces’ content are stored in one directory.

- The committed versions of all workspaces' content are stored in another directory.
- Each file contains its own status, that is, whether it is a development version or a committed version.

See Figure 10.

Figure 10. Workspaces and their content in the repository.



In this example, workspace B has been opened. It shows a view of the development version of message set 2 and certain microflows that you have modeled, adapters that you have created and everything that they refer to. They can refer to the development version or to the committed version. In this workspace, you can create a new message or message set or you can model a new microflow or you can create an adapter.

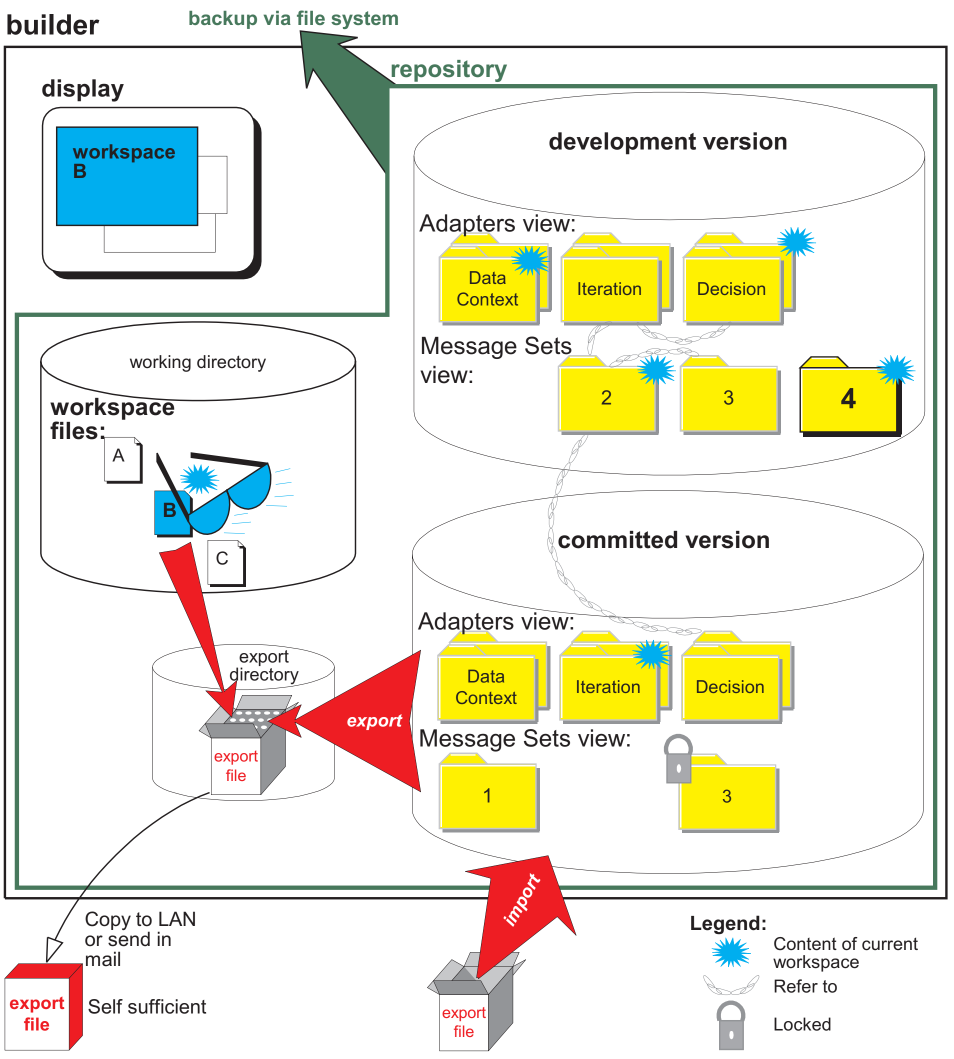
After creating anything (except message sets) a development version does not exist until the workspace has been saved. In this example, message 4 has been created and saved. When a workspace is saved, a development version of any newly created components is created. Prior to the workspace save, these newly created components only exist in the builder's memory. Thus, there is a development version but no committed version yet.

When you create a message set, its committed version is made but it is immediately and automatically checked out. The result is that a newly created message set exists in both versions.

Manage the repository

You can manage the repository itself. See Figure 11.

Figure 11. Manage the repository itself.



The repository resides in the builder that runs on your computer. It cannot reside on a network drive or other remote drive. The builder uses the Windows file system to manage the repository. The repository is not managed by a database.

You can and should backup and, if needed, restore the repository as a whole, by direct action through the file system. You could backup the repository directory by zipping and storing it on a network drive or removable media or offsite. You can copy export files from the export directory to make them available to others, by direct action through the file system.

The repository itself is not shared. However you can:

- Share the committed version of your workspace and its content, in the form of an export file, with others for use with their copy of the builder.
- Share others' export files by importing them into your copy of the builder.

You share with others by making your export files available to others by direct action through the file system, such as by copying it to a network drive or attaching it to an e-mail.

Manage workspaces and versions of workspace content

You can manage workspaces and versions of workspace content. You manage them within the repository. See Figure 12 on page 43.

- You *save the workspace,* that is, what you have built in the graphical space in the builder. This stores your workspace and the development version of its content.
- You can *check in* any item in the content of a workspace, or all of the content of one workspace or all development versions. This creates the committed version.

  There are several ways to check in:

  check in → list
  : You can select any number of items in the workspace and check them in.

  check in → all in workspace
  : You check in:
    - Everything in the displayed workspace that had already been saved as a development version, including everything that it refers to.
    - Anything that is visible in the displayed workspace but that had not yet been saved as a development version.

  check in → all
  : You check in:
    - All development versions that have been saved. This includes anything that had been checked out and anything that has been saved but not yet checked in.
    - All development versions that have not been saved, that it, that reside in memory. This includes anything that has never been saved, in other words, new, and anything that had been checked out and edited but not saved yet.

    Check in → all works on all development versions, even those that do not belong to any workspace. For example, you could have something in your workspace that is checked in. If you check it out and then remove it from your workspace, it might not be in any workspace anymore but it still gets checked in.

When you check in the workspace content or any item in the content, the builder creates the committed version and then deletes the development version. The builder also releases the lock that had been put on it when it was checked out. In Figure 12 on page 43, message 4 is being checked in.

Typically, you would check in before exporting, to release all locks and thus enable someone else to work on the exported workspace.

- You can *check out* the committed version of an item. When you check out an item, the builder creates the development version based on a copy of the committed version. The committed version remains, but the builder locks it. In Figure 12 on page 43, message 3 is being checked out. Because the committed version of message 3 is checked out, it is locked.

  Note: Locked Committed Version. A locked committed version can be exported. The locked condition is exported also. After the workspace content has been imported to another copy of the builder, its committed version is locked on that copy of the builder. When the other user attempts to check the locked workspace content out, the other user is warned that the workspace content is locked. This prompts the other user to check with the original user to determine if the export file is actually ready to be worked on:

  – If ready, the other user can unlock the committed version of the item in the workspace via the builder. To unlock:

    1. Determine under which user id the item was locked. When you attempt to check out the item, the message will tell you the user id. Disregard the user workstation.
    2. Create that user id on your computer and sign on under it.
    3. Open the workspace with the locked item, select it and invoke **Unlock**.
    4. Sign on under your own user id and continue to work on the item.

  – If not, the original user can complete work on it, check it in, and re-send an export file that is ready.

- You can *revert to committed.* This means that you can return all development versions of all workspaces’ content to their committed versions. This also:

  – Deletes all development versions in the repository.

  – Unlocks all committed versions in the repository.

  Typically, you would revert to committed in order to:

  – Discard work-in-progress that had been saved in a development version.

  – Ensure that all committed versions are unlocked before you export them.

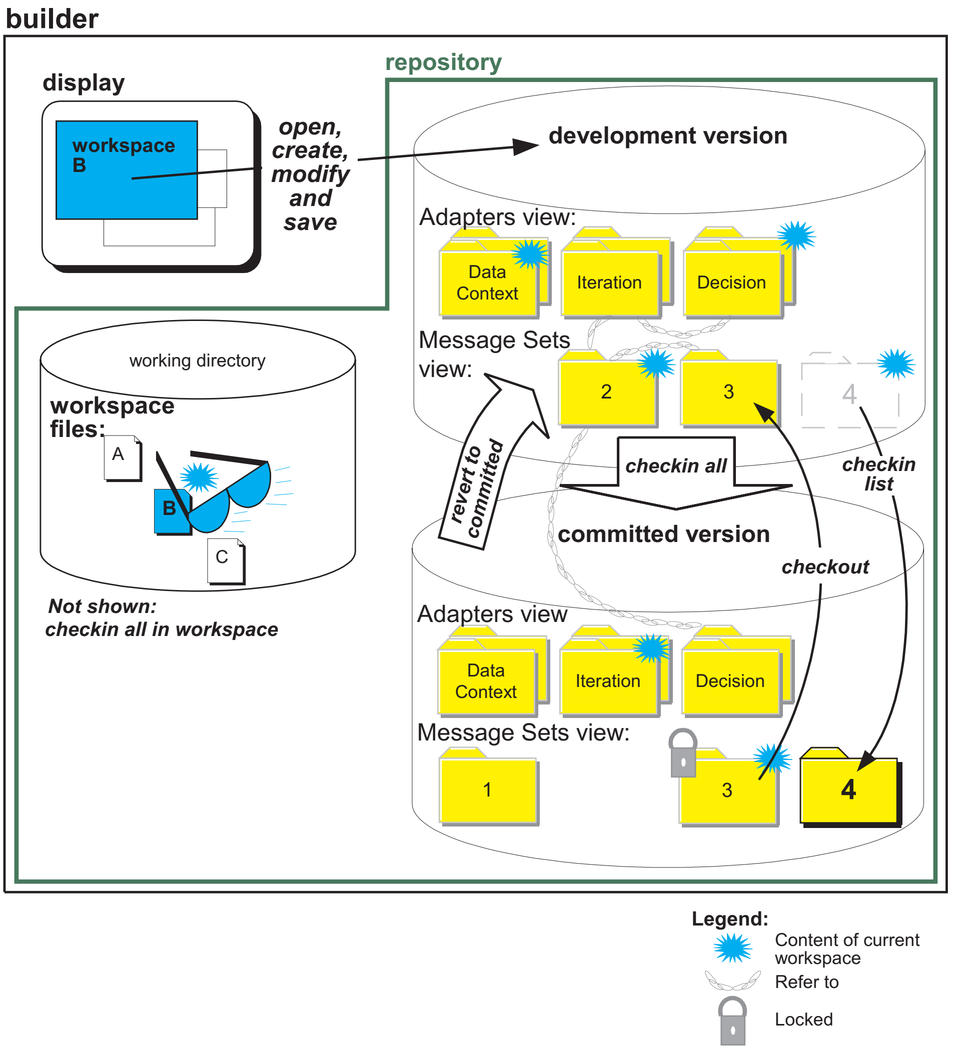
    However, in order to save work-in-progress before exporting, you would **check in → all** instead.

    Revert to Committed works only on things that had been committed earlier. If you create something new and invoke **Revert to Committed**, it remains, either in memory only if it had never been saved or as a development version if it had been saved.

Attention:

- You can import or export the entire workspace and its content. You cannot import or export part of a workspace. You cannot import or export multiple workspaces in one action.
- You can check in or check out only what you need: one or more items within a workspace (although each item must be checked in or out individually).

Figure 12. Managing workspaces and versions of workspace content.



Techniques for managing your workspaces and the repository

CAUTION:

Do not manage workspaces by direct action through the file system, except for backups, restorations and copying export files. Unpredictable results might occur.

You should use these techniques for managing your workspaces and the repository.

Repository

- Back up the repository directory via the file system routinely to protect your work. You could backup the repository directory by zipping and storing it on a network drive or removable media or offsite. Store the backups off site.
- Save the workspace before you back up in order to ensure that all contents of the workspace are protected by the backup.

Save, import and export workspace, and check in and check out

- Before you import any export file, back up the repository directory via the file system.
- If you cancel during the act of importing an export file, the files that were imported to the repository before you cancelled remain imported. In the case when you are importing something new, this might not cause a problem. However, if you were importing a workspace and if you already had a different version of it (including anything that it referred to) in the repository, then there is a risk that the contents of the workspace will not be synchronized. Unpredictable results might occur. To prevent this, back up the repository before you import anything. Then restore the repository if a problem like this occurs.
- Checking in, checking out, and exporting and importing workspaces can take some number of minutes. A very complex structure can require a large number of files to represent it.
- Saving the workspace to create the development version and checking in to create the committed version perform very limited validation. These acts validate only the relationship among nodes in the microflow. They do not validate values in property sheets.
- Importing an export file performs virtually no validation except to check that it contains a workspace file.

Importing an application's interface

- Creating or importing an application's interface might create many files and several directories in the repository. Each message set has its own directory. Within that message set directory are directories for each of the kinds of objects that the message set contains. For example, there is one directory for messages in that message set, one directory for transactions, one for fields in a message, one for valid values and so on. Every object (in other words, every message, transaction, valid value and so on) actually has two files. One file contains the object's content. The other file contains properties of the object, for example, lock status and name.

- Before you import an application's interface, make sure that you have saved the workspace.
  - If a problem happens as a result of the import of the applications's interface, then before you save, you can immediately open the current workspace again. This will restore the workspace to its state before you imported.
  - If you import an application's interface and then attempt to import another one before saving first, an out of memory error might occur. It is a good practice to save after each time you import an application's interface.
- If an out of memory error occurs during an attempt to import an application's interface, you might need to increase the memory available to the builder. Edit the batch file (in the *mqab.bat* file found in the root installation directory) that starts the builder. Find the line `Set MQAB_memory=150M` and increase the 150M to an appropriate value for your system. This value controls the amount of heap space that is allocated to the Java virtual machine that runs the builder. Take into account the memory that is present in the computer when you determine which value to use. Close and re-start the builder. Try again.
- If you import an application's interface:
  - And if you do not plan to change it immediately, it is most efficient to check it in and thus create the committed version. Later you can check out what you want to edit.
  - And if you do plan to change it immediately, it is most efficient to save the workspace thus creating the development version and then edit it and finally check it in.

Tasks: Manage your workspaces and the repository

Generally, you should manage workspaces and the repository in the following ways. Read "Techniques for managing your workspaces and the repository" on page 44 before you perform these tasks.

Build an adapter and export the workspace

1. Open or create a workspace.
2. Before you import an application's interface, make sure that you have saved the workspace.
   a. If a problem happens as a result of the import of the applications's interface, then before you save, you can immediately open the current workspace again. This will restore the workspace to its state before you imported.
3. Create or import an application's interface and model microflows.
4. Save the workspace at any time to protect your work and thereby create the development version of the workspace and its content. Repeat as needed.
5. Generate, compile and test the adapter.
6. When the adapter meets your requirements, check the workspace content in and thereby create the committed version. This deletes the development version.
7. If desired:
   a. Check out the committed version and thereby re-create the development version. The committed version remains but it is locked.
   b. Modify the development version.
   c. When the adapter meets your requirements, check the workspace content in and thereby write over the committed version, replacing the previous committed version. (The new committed version is unlocked.) This deletes the development version.
8. If desired, while the workspace is open, export the workspace and thereby create the export file in the export directory. The export file is in zip format.
   - If you export the committed version while it is checked out and thus locked, the locked status is exported also. See "Note on locked committed version" on page 42.
9. Make the export file available to someone else. For example, you can copy it to a network drive or attach it to an e-mail.

Import a workspace, and modify to build an adapter

1. Acquire an export file. The export file is in zip format.
2. Back up the repository. This will enable you to recover if importing causes a problem.
3. Import the export file and thereby create the committed version of the workspace content. This also creates a workspace file.
4. Open the workspace.
5. If required, check out the committed version of whatever you need to modify and thereby create the development version.
   a. When you attempt to check out the committed version, if you are warned that it is locked, see "Note on locked committed version" on page 42.

   After the workspace content has been checked out, the committed version remains but it is locked.

   Note: You might not need to modify anything. You might need only to add new things. For example, the export file could contain a message set and no adapters and you need only to create an adapter with that message set. In this case, you do not need to check anything out.
6. Modify the development version.
7. Save the workspace at any time to protect your work. Repeat as needed.
8. Generate, compile and test the adapter.
9. When the adapter meets your requirements, check the workspace content in and thereby write over the committed version, replacing the previous committed version. This deletes the development version.
10. If desired, check out and work on the workspace content again.
11. If desired, export the workspace.
    - If you export the committed version while it is checked out and thus locked, the locked status is exported also. See "Note on locked committed version" on page 42.

Chapter 7. Obtaining additional information

There are several sources of information that can be useful when you are using MQSeries Adapter Offering.

Available on the Internet

The MQSeries product family Web site is at:

http://www.ibm.com/software/ts/mqseries/

By following links from this Web site you can:

- Obtain latest information about the MQSeries product family, including the MQSeries Adapter Offering.
- Access the MQSeries books in HTML and PDF formats, including a more recent edition of this book.
- Download MQSeries SupportPacs.

MQSeries Adapter Offering information

See:

- The MQSeries Adapter Kernel's books and online documentation on APIs.
- The MQSeries Adapter Offering's website at http://www.ibm.com/software/ts/mqseries/adapter/information/.

  By following links from this website you can:

  – Obtain the latest information about MQSeries Adapter Offering.
  – Access tutorials, lab exercises and similar resources.

References

You might find these sources of reference information useful:

- The Open Applications Group website at http://www.openapplications.org/.
- Extensible Markup Language (XML) 1.0 W3C Recommendation at http://www.w3.org/TR/1998/Rec-xml-19980210.

These are not IBM websites.

Notices

This information was developed for products and services offered in the United States. IBM may not offer the products, services, or features discussed in this information 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 information. The furnishing of this information 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 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 information. IBM may make

Trademarks

Glossary

The glossary contains *key* terms and their meanings as used in the information.

If a particular concept or term appears in one section only, it might not be contained in the glossary. It might, however, be found via the “Index” on page 65.

The glossary does not contain terms of other IBM products such as MQSeries.

A

access bean. An enterprise bean that is used to encapsulate session and entity beans. See “Enterprise JavaBean” on page 57.

accessor methods. Methods that an object provides to define the interface to its instance variables. The accessor method to return the value of an instance variable is called a *get* method or *getter* method, and the accessor method to assign a value to an instance variable is called a *set* method or *setter* method.

adapter . The output of the MQSeries Adapter Builder. Typically, the user builds each adapter to be specific to *one message type* that is sent from or to an application. Thus, the adapters themselves are not part of MQSeries Adapter Offering. An adapter consists of C source code that compiles to a shared library. When the adapters and the MQSeries Adapter Kernel execute together at run time, they perform the run time functionality of the MQSeries Adapter Offering. Depending on how it was modeled by the user in the MQSeries Adapter Builder, the adapter can contain a wide variety of functionalities such as controlflow, dataflow, sequential navigation, conditional branching including decision and iteration, data typing, storing data context, transformation of data elements, logical operations and custom code. The user may reuse adapters that the user has created.

API. See “application programming interface”.

applet. An application program, written in the Java programming language, that can be retrieved from a Web server and executed by a Web browser. A reference to an applet appears in the markup for a Web page, in the same way that a reference to a graphics file appears; a browser retrieves an applet in the same way that it retrieves a graphics file. For security reasons, an applet’s access rights are limited in two ways: the applet cannot access the file system of the client upon which it is executing, and the applet’s communication across the network is limited to the server from which it was downloaded. Contrast with “servlet” on page 62.

application. In Java programming, a self-contained stand-alone Java program that includes a main() method. Contrast with “applet”.

application-neutral format. See “intermediate formatted message” on page 58.

application programming interface (API). A software interface that enables applications to communicate with each other. An API is the set of programming language constructs or statements that can be coded in an application program to obtain the specific functions and services provided by an underlying operating system or service program.

application-specific interface. An interface that is between the source application and the source adapter or between the target adapter and target application. It is developed, outside of MQSeries Adapter Offering, by the user for either of these purposes:

- To enable the source adapter to acquire the message from the source application.

- To enable the target application to acquire the message from the target adapter.

The exact nature of the application-specific interface depends on the characteristics of the source application and of the target application. Some examples of application-specific interfaces include:

- API calls and user exits
- File reads and writes
- Database triggers
- MQSeries queues

Contrast with "microflow" on page 60 and "routing" on page 62.

argument. A data element, or value, included as a parameter in a method call. Arguments provide additional information that the called method can use to perform the requested operation.

attribute. A specification of a property of a bean. For example, a customer bean could have a name attribute and an address attribute. An attribute can itself be a bean with its own behavior and attributes.

B

bean. A definition or instance of a JavaBeans component. See "JavaBean" on page 58.

BOD. Business Object Document. A representation of a standard business process that flows within an organization or between organizations. Examples are add purchase order, show product availability and add sales order. BODs are defined by the OAG using XML. See "OAG" on page 61 and "XML" on page 64.

BODs can be used by the MQSeries Adapter Offering to define message bodies in its intermediate formatted messages. Although use of BODs is recommended, use of BODs is not required.

builder. Synonymous with MQSeries Adapter Builder.

C

category. An optional grouping of messages or transactions that are related in some way. For example, messages that relate to a particular application.

class. A template that defines properties, operations, and behavior for all instances of that template.

class hierarchy. The relationships between classes that share a single inheritance. All Java classes inherit from the Object class.

class library. A collection of classes.

class method. See "method" on page 60.

CLASSPATH. In your deployment environment, the environment variable that specifies the directories in which to look for class and resource files.

command. A command is a simple adapter component that is used to represent API calls. An example of a command is an API call used to place a message into the runtime infrastructure.

committed version. The workspaces and their content that you have completed work on. You have committed them. You can *export* the committed version to an *export file* in the *export directory*. You can make the export file available to enable someone else to use the workspace and its content on their copy of the builder. You can import a workspace and its content that had previously been exported to an export file.

connection. A wire that connects an output terminal of one microflow node to the input terminal of another. There are two types of connections: control connection and data connection. See "control flow" and "data flow connector" on page 57.

control connection. A connection that provides the sequence relationship between two nodes in a microflow. Control and data is passed from one node to the next via the control connection.

control flow. The collective term for the control relationship between nodes in a microflow. See “control flow connector” on page 56.

D

data context. A data context is a simple adapter component that is used to store data for later access through a data flow.

data flow. See “data flow connector”.

data connection. A connection that establishes a relationship between the data of two nodes in a microflow. It is used when the data is coming from anywhere except the node from which control is being passed. It can be used to support mapping. The collective term for all of the data flow nodes in a microflow is data flow.

data transformation. See “mapping” on page 59.

development version. The workspace content that you can actively work on. You develop on the development version. You can create and open workspaces, you can import or create messages and model new microflows, you can modify messages and microflow, and you can create adapters. You save workspaces, including their content, to protect the work that you have done but before you have completed building the adapter.

DTD. Under XML, Document Type Definition. Usually a file (or several files used together) that contains a formal definition of a particular type of document. It sets out which names can be used for elements within the DTD, where elements may occur within the DTD and how the elements fit together. In MQSeries Adapter Offering, you can use DTDs to define message bodies. See “XML” on page 64 and “intermediate formatted message” on page 58.

E

EJB client. There are two general types of EJB clients:

- *HTTP-based clients* that interact with the EJB server by using either Java servlets or JavaServer Pages (JSP) by way of the Hypertext Transfer Protocol (HTTP).
- *Java applications* that interact directly with the EJB server by using Java remote method invocation over the Internet Inter-ORB Protocol (RMI/IIOP).

See “EJB server”.

EJB container. The EJB container hosts and manages EJBs at run time. EJB containers provide services including support for remote access to the bean, security, persistence, transactions, concurrency, and access to and pooling of resources. EJB containers rely on an execution environment provided by an EJB server. The EJB developer can focus on encapsulating business rules, while the container takes care of everything else.

EJB server. The EJB server provides an environment to support the execution of applications developed using EJBs. It manages and coordinates the allocation of system resources. The EJB server and EJB container provide the run time for an EJB. An EJB server may have multiple EJB containers. The servers have Java Naming and Directory Interface (JNDI), which are accessible naming spaces for the client to locate the enterprise beans. See “Java Naming and Directory Interface” on page 59.

element. A unit of data within a message that has business meaning, for example, street name.

encapsulation. The hiding of a software object’s internal representation. The object provides an interface that queries and manipulates the data without exposing its underlying structure.

enterprise bean. See “Enterprise JavaBean”.

Enterprise JavaBean (EJB). Enterprise JavaBeans define a component model for the development, deployment, and execution of Java applications based on a multi-tier, distributed object architecture. It extends the JavaBeans component model to support server components, including server functionality such as global naming, distributed transactions, persistence, and security. EJB is designed specifically to facilitate the building of robust and scalable server components and separates the concerns of system programming from those of business programming. EJBs define both transient (session beans) and persistent (entity beans) objects.

entity bean. Entity beans are persistent objects. Persistent objects are created, destroyed, loaded, or unloaded from memory. An entity bean has the following features:
- Supports shared access from multiple users
- Participates in transactions
- Represents data in the database
- Lives as long as the data in the database
- Survives EJB server crashes
- Has persistent object references

Contrast with “session bean” on page 62.

event. An action by a user program, or a specification of a notification that may trigger a specific behavior In the JDK, events notify the relevant listener classes to make appropriate actions.

export file. A compressed ″zip″ format file that contains one workspace file and all of the workspace’s content. An export file is self-sufficient as long as all message sets had been in the workspace when it was exported. It contains everything that is required to enable someone to import the workspace and then work on its content ″as is″.

I

inheritance. (1) A mechanism by which an object class can use the attributes, relationships, and methods defined in more abstract classes related to it (its base classes). (2) An object-oriented programming technique that allows you to use existing classes as bases for creating other classes.

instance. A synonym for object, a particular instantiation of a data type. See “object” on page 61.

interface. A set of methods that can be accessed by any class in the class hierarchy.

intermediate formatted message.

The intermediate formatted message consists of message control values plus the application data. An example is an XML document that the source adapter had transformed from the source application’s format to XML.

Message control values are in the header portion and the application data is in the message body.

The intermediate-formatted message is in a format that is output by the source adapter and input by the target adapter. It is trafficked only within the MQSeries Adapter Kernel, MQSeries and optionally MQSeries Integrator. The source application and target application are not required to be aware of the intermediate message format.

The message body is not required to be in an application-neutral format. Although this is not recommended, the format of the message body could be proprietary or otherwise specialized.

introspection. Introspection is used to programmatically find out the interfaces offered by a JavaBean. The interface consists of the properties, methods, and events that the bean exposes for use by other bean.

J

Java. An object-oriented programming language for portable interpretive code that supports interaction among remote objects. Java was developed and specified by Sun Microsystems, Incorporated.

Java 2 Enterprise Edition (J2EE). A Java development standard that provides a component-based approach to the design, development, assembly, and deployment of enterprise applications.

Java archive (JAR). A platform-independent file format that groups many files into one. JAR files are used for compression, reduced download time, and security. Because the JAR format is written in Java, JAR files are fully extensible.

JavaBean. A platform-independent, software component technology for building reusable Java components called ″beans.″ Once built, these beans can be made available for use by other software engineers or can be used in Java applications. Also, using JavaBeans, software engineers can manipulate and assemble beans in a graphical drag-and-drop development environment. The bean interface exposes the events, properties, and methods of the bean and makes them available to other beans. See “Enterprise JavaBean” on page 57.

JavaBeans provide support for:

- *Portability*: Beans can be created and run on any Java platform.
- *Introspection*: The tool that you use to combine beans can automatically discover how a bean works.
- *Properties and customization*: Properties are a bean’s attributes. A developer using a bean can customize the appearance and behavior of a bean by changing its properties.
- *Persistence*: The state of a bean can be saved and then reloaded through the serialization function of the JDK.

Java Database Connectivity (JDBC). A Java specification that defines an API that enables programs to access databases that comply with this standard.

Java Development Kit (JDK). A software package that can be used to write, compile, debug, and run Java applets and applications.

Java Message Service (JMS). An enterprise-capable middleware component based on message-oriented middleware fundamentals.

Java Message Service (JMS) listener. A component provided by the WebSphere Business Integrator product that enables tight integration between MQSeries Adapter Kernel and WebSphere Application Server Advanced Edition.

Java Naming and Directory Interface (JNDI). The Java standard API for accessing directory services, such as LDAP, COS Naming, and others.

Java Runtime Environment (JRE). A subset of the Java Development Kit (JDK) that contains the core executables and files that constitute the standard Java platform. The JRE includes the Java Virtual Machine, core classes, and supporting files.

Java service adapter. An adapter that provides services for the implementation of a business process flow. The Java service adapter represents the flow modeled in one or more microflows. The MQSeries Adapter Builder converts the microflow(s) associated with a Java service adapter into one or more service beans.

In a WebSphere environment, the service bean generated is a session bean that include methods for each of the selected Java microflows. In a non-WebSphere environment, the service bean is the message bean that invokes the microflow bean rather than the session bean.

Java Virtual Machine (JVM). A software implementation of a central processing unit (CPU) that runs compiled Java code (applets and applications).

K

kernel. Synonymous with MQSeries Adapter Kernel.

L

listener. A class that receives and handles events.

M

map. A composed adapter component that models data transformation. A map can also simply copy data from one field in the input message to another field in the output message.

mapping. The act of the user who models data transformation via a Map node between an output terminal on one node and an input terminal on another node. Data transformation can include a variety of functions:

- Associating a field in one message with a field in another message.
- String mapping such as specifying pad characters.
- Date mapping, such as converting a date in one format to a date in another format.
- Putting literal data into messages.
- Adding custom code to perform other data transformation functions.

A map can also simply copy data from one field in the input message to another field in the output message.

message. In MQSeries, including MQSeries Adapter Kernel, a collection of data that is sent by one program and intended for another program.

message control values. A collective term for a set of values in messages that:

1. The kernel uses to control the marshalling and routing of messages during run time.
2. Each adapter uses to control, in part, how it performs its functionality.

Message control values that you can set in the adapter to control routing include:

- Source logical identifier
- Destination logical identifier
- Respond to logical identifier
- Body category
- Body type
- Acknowledgment requested

See the MQSeries Adapter Kernel documentation for details about message control values and routing.

message queue interface (MQI). The programming interface provided by MQSeries queue managers. The programming interface allows application programs to access queueing services.

message set. A collection of messages and the components that make them up.

message set. After you import or when you create and modify the structures of messages in the MQSeries Adapter Builder, they are maintained in the form of message sets in the builder's Message Set view.

method. A fragment of Java code within a class that can be invoked and passed a set of parameters to perform a specific task.

microflow. A directed graph that models the processing of a message as it passes *within* one adapter, from the input of the adapter to the output of the adapter:

- Within the source adapter, the input terminal of the adapter is where the message is received from the source application. The output terminal of the adapter is where information may be returned to the source adapter. Internally, within the adapter, messages may be created, transformed and handed to the kernel to be put on the MQSeries queue.
- Within the target adapter, the output terminal of the adapter is where a reply message may be sent back into the kernel. Internally within the adapter, messages may be created and transformed and sent to a target application or handed to the kernel to be put on the MQSeries queue.

Microflows can be nested. In this case, the most outermost nesting microflow adheres to the standard definition for the microflow, but the nested microflows do not. Instead of interacting directly with the source or target adapters, a nested microflow gets a message from a nesting microflow, processes it, and then passes it back to the nesting microflow.

The microflow is the model of the functionality that is realized in the compiled adapter. A microflow consists of a set of microflow nodes and the connectors that connect them.

Contrast with "routing" on page 62 and "application-specific interface" on page 55.

microflow node. The generic term for any node in the microflow. Each type of microflow node represents a well defined processing stage. A set of microflow nodes is provided with the builder.

One of the microflow components that is provided is called “Microflow”. All of the other microflow nodes and connectors that can potentially be used in a microflow are modeled within the Microflow node.

MQSeries. A family of IBM licensed programs that provides message queuing services.

MQSeries Adapter Builder. The MQSeries Adapter Builder provides support to define and build an adapter for most applications. It does so by providing an intuitive user interface. The user interface is similar to MQSeries Integrator’s user interface. See “adapter” on page 55.

MQSeries Adapter Kernel. A set of APIs and several executable programs, in C and Java, and several configuration files. The kernel works with and supports the adapters. See “adapter” on page 55. In addition to directly supporting adapters, the kernel performs related functions, among the most important: routing of messages and infrastructure services such as message construction, tracing and interfacing with MQSeries.

The kernel is installed on each computer on which a source adapter or a target adapter run.

MQSeries Adapter Offering (MQAO). A set of application integration products that consists of the MQSeries Adapter Builder and the MQSeries Adapter Kernel.

N

node. See “type” on page 63.

null. Within the context of an SQL statement, null refers to a data field that is not initialized.

O

OAG. Open Applications Group. It is a non-profit industry consortium comprised of many prominent stakeholders in the business software component interoperability arena. The OAG defines Business Object Documents (BODs). See “BOD” on page 56.

object. (1) A collection of data and methods that operate on that data, which together represent a logical entity in the system. In object-oriented programming, objects are grouped into classes that share common data definitions and methods. Each object in the class is said to be an instance of the class. (2) An instance of an object class consisting of attributes, a data structure, and operational methods. Each instance has the same properties, attributes, and methods as other instances of the object class, although it has unique values assigned to its attributes. (3) A computer representation of something that a user can work with to perform a task. An object can appear as text or an icon.

operation. (1) A system function, such as an API. (2) A method or service that can be requested of an object.

P

package. A program element that contains related classes and interfaces.

port. A terminal on a Class node that represents a request (in terminal) or response (out terminal) for an exposed method.

property. (1) One of a set of characteristics that define the values and behaviors of objects in the Control Center. For example, microflow nodes have properties. (2) An initial setting or characteristic of a JavaBean; for example, a name, font, text, or positional characteristic.

Q

queue. An MQSeries object. Message queuing applications can put messages on, and get messages from, a queue. A queue is owned and maintained by a queue manager. Local queues can contain a list of messages waiting to be processed. Queues of other types cannot contain messages: they point to other queues, or can be used as models for dynamic queues.

queue manager. A system program that provides queuing services to applications. It provides an application programming interface (the MQI) so that programs can access messages on the queues that the queue manager owns.

R

repository. In MQSeries Adapter Builder, a file system directory that contains definitions of the adapters. It is not used when running an adapter; it is only used by MQSeries Adapter Builder when defining an adapter and generating the adapter code.

Remote Method Invocation (RMI). A Java API that enables you to write distributed Java programs, allowing methods of remote Java objects to be accessed from other Java virtual machines.

routing. The process, performed by the kernel, of sending a message from the source adapter and delivering it to the appropriate target adapter. Routing is performed between the source adapter and the target adapter.

Routing is performed in two stages:

1. The source side of the kernel puts the message on the appropriate MQSeries queue. As an option, if MQSeries Integrator has been configured as the destination, MQSeries Integrator can perform certain brokering functions. Then, whether via MQSeries or via MQSeries and MQSeries Integrator, the message arrives on the appropriate MQSeries queue.
2. The target side of the kernel gets the message from the MQSeries queue and invokes the appropriate target adapter.

The adapter's message control values help control routing. See "message control values" on page 60.

Contrast with "microflow" on page 60 and "application-specific interface" on page 55.

S

session bean. Session beans are transient objects, which represent processes or act as agents performing tasks. A session bean has the following features:

- Executes on behalf of a single client
- Is transaction aware
- Updates data in an underlying database
- Does not represent data that should be stored in a database
- Is relatively short-lived (life typically is that of its client)
- Is destroyed when the EJB server crashes

Contrast with "entity bean" on page 58.

servlet. An application program, written in the Java programming language, that is executed on a Web server. A reference to a servlet appears in the markup for a Web page, in the same way that a reference to a graphics file appears. The Web server executes the servlet and sends the results of the execution (if there are any) to the Web browser. Contrast with "applet" on page 55.

signature. See "transaction" on page 63.

source adapter. An adapter that

- Accepts or otherwise acquires structured data from a source application (typically via an application-specific interface that is developed by a user outside the adapter).
- Processes the structured data according to how the adapter had been modeled.
- Transforms the structured data into an intermediate message format.
- Via the kernel, puts the message onto an MQSeries queue, for delivery to one or more target adapters and thence to the target application.

For each message type, there is one source adapter. Typically, a source application can send multiple message types; therefore in most cases, a source application is supported by multiple source adapters.

See "adapter" on page 55.

source application. Program that is required to send data via a computer network to a program (known as the target application) that typically resides on another computer.

source side of the kernel. In this information, the part of the kernel functionality that begins when the message is received from the source adapter and that ends when the message is put onto the MQSeries queue.

superclass. A class from which other classes or beans are derived.

T

target adapter. An adapter that:

- Receives a message (via the kernel and MQSeries) that had been sent by a source adapter.
- Processes the intermediate-formatted message according to how the adapter had been modeled.
- Transforms the intermediate-formatted message into an application-specific formatted-message that the target application can receive.
- Sends the message to the target application via an application-specific interface.
- Lets the kernel know when it has completed sending the message to the target application, to enable the worker to send an acknowledgment.

If the target application can receive the intermediate-formatted message, then a target adapter might not be required.

For each message type, there is one target adapter. Typically, a target application can accept multiple message types; therefore in most cases, a target application is supported by multiple target adapters.See "adapter" on page 55.

target application. Program that is required to receive data via a computer network from a program (known as the source application) that typically resides on another computer.

target side of the kernel. In this information, the part of the kernel functionality that begins when the message is gotten from the MQSeries queue and that ends when the message is sent to the target adapter.

terminal. The point at which one node in a microflow is connected to another node. Terminals enable you to control the route a message takes. If the operation performed by a node on the message is successful, control is passed to one terminal (or a particular terminal if there are multiple valid terminals). If the operation is not successful, control is passed to a different terminal.

transaction. A transaction represents the relationship between the data content of the messages that are supplied to and produced by a Command node. For example, in a request/reply message relationship, the request message may supply a parameter list to the command and expect the reply message from the command to return a data structure. The relationship between the request and reply data for the command is the transaction.

A transaction has a different meaning when referring to the MQSeries Adapter Kernel's transactional capabilities. In this context, a transaction is a set of operations that must be executed as an indivisible unit of work. If all operations that comprise a transaction are successful, the transaction is committed; that is, all of the operations are performed. If one or more of the operations that compose a transaction fail, the transaction is rolled back; that is, none of the operations are performed. By using MQSeries Adapter Kernel's transactional capabilities, a source adapter can send a series of messages at once, with the assurance that all messages will be sent if the transaction is committed or that no messages will be sent if the transaction is rolled back.

transform. A defined way in which a message of one format is converted into one or more messages of another format. The Map node is used to transform messages.

type. A type represents a template that can be used as a building block in adapter modeling. When a type is dragged onto the Microflow Definition pane, an instance of that type is created and the instance is inserted into the microflow. This instance is referred to as a node. A single type can be used to create one or more nodes (instances) as part of the same microflow. Types are listed in the Adapters view in type collections.

type collection. A collection of types.

V

variable. (1) A storage place within an object for a data feature. The data feature is an object, such as number or date, stored as an attribute of the containing object. (2) A bean that receives an identity at run time. A variable by itself contains no data or program logic; it must be connected such that it receives run-time identity from a bean elsewhere in the application.

W

WebSphere. Pertaining to a family of IBM software products that provide a development and deployment environment for basic Web publishing and for transaction-intensive, enterprise-scale e-business applications.

wire format. This describes the physical representation of a message within the bit-stream.

workspace. A workspace is a view of what you can work with at one time. A workspace is displayed as the graphical space in the builder where you build adapters. The workspace's content is:

- The particular message sets, which are displayed in the Message Sets view.
- The microflows that you have modeled (that are associated with the message sets), the adapters that you have built and the component types provided with the builder, which are displayed in the Adapters view.
- Everything that the message sets, microflows and adapters refer to, such as primitives.

Two versions of each workspace's content can be stored: development version and committed version.

W3C. World Wide Web Consortium. An international industry consortium set up to develop common protocols to promote evolution and interoperability of the World Wide Web.

X

XML. Extensible Markup Language. A W3C standard for the representation of data.

Index

Printed in U.S.A.

GC34-5882-02