WebSphere Message Broker

IBM

CMP Programming

Version 6 Release 1

WebSphere Message Broker

IBM

CMP Programming

Version 6 Release 1

Contents

About this topic collection

This PDF file has been created from the WebSphere Message Broker Version 6.1 (fix pack 6 update, March 2010) information center topics. Always refer to the WebSphere Message Broker online information center to access the most current information. The information center is periodically updated on the document update site and this PDF and others that you can download from that Web site might not contain the most current information.

The topic content included in the PDF does not include the "Related Links" sections provided in the online topics. Links within the topic content itself are included, but are active only if they link to another topic in the same PDF collection. Links to topics outside this topic collection are also shown, but result in a "file not found "error message. Use the online information to navigate freely between topics.

Feedback: do not provide feedback on this PDF. Refer to the online information to ensure that you have access to the most current information, and use the Feedback link that appears at the end of each topic to report any errors or suggestions for improvement. Using the Feedback link provides precise information about the location of your comment.

The content of these topics is created for viewing online; you might find that the formatting and presentation of some figures, tables, examples, and so on are not optimized for the printed page. Text highlighting might also have a different appearance.

Part 1. Developing applications using the CMP

Developing applications that use the Configuration Manager Proxy API

Develop Java™ applications that use the Configuration Manager Proxy to communicate with, deploy to, and manage broker domain components.

Before you start:

Read an overview of the "Configuration Manager Proxy," and how you can use it to manage the resources associated with your broker domain.

Samples are provided to demonstrate typical CMP scenarios. Run and explore the samples to learn about what you can do with the CMP; see "The Configuration Manager Proxy samples" on page 5.

Write applications to run tasks in your broker domain:
- Configuring the environment
- Connecting to a Configuration Manager
- Navigating broker domains
- Deploying resources
- Setting user-defined properties dynamically at run time
- Managing broker domains
- Using advanced features

When you have created your applications, test their operation in your test environment. Check the results of the operations that the programs have performed by using the Broker Administration perspective.

When you have written and tested your applications, distribute them to the computers from which you want your administrators to perform the tasks that you have developed.

Configuration Manager Proxy

The Configuration Manager Proxy (CMP) is an application programming interface that your applications can use to control broker domains through a remote interface to the Configuration Manager.

Your applications have complete access to the Configuration Manager functions and resources through the set of Java classes that constitute the CMP API. For example, you can use the CMP API to interact with the Configuration Manager to perform the following tasks:
- Deploy BAR files
- Deploy Publish/Subscribe topology, topic trees, and broker configuration.
- Modify the Publish/Subscribe topology; add and remove brokers, broker connections, and collectives.
- Manipulate the topics hierarchy.
- Create, modify, and delete execution groups
- Inquire and set status of resources (for example, run state), and be informed if status changes.
- View the broker event log

- View the active subscriptions table.
- Modify domain Access Control Lists, when connected to a Version 6.1 or Version 6.0 Configuration Manager only.

The CMP API is a set of Java classes that sit logically between the user application and the Configuration Manager, inside the Java Virtual Machine (JVM) of the user application. It requires the WebSphere® MQ Classes for Java in order to function, as shown in the following diagram.
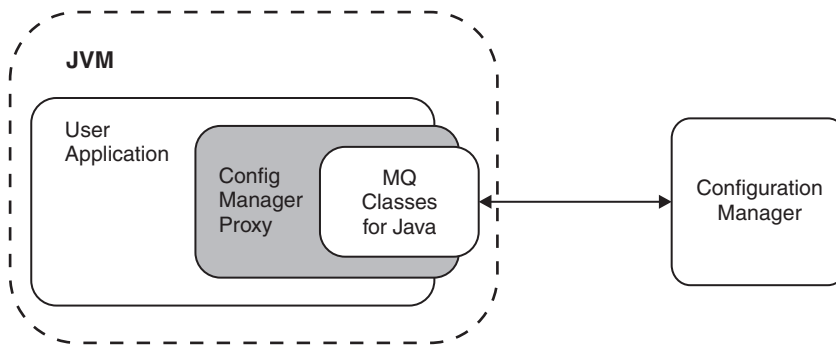
The CMP API application can be on the same physical machine as the Configuration Manager, connected by a JNI (Java Native Interface) connection to the queue manager using the WebSphere MQ Java Bindings transport. Alternatively, it can be distributed over a TCP/IP network, and connected to the queue manager by a WebSphere MQ SVRCONN channel using the WebSphere MQ Java Client transport.

You can use the CMP API to communicate with more than one Configuration Manager from within the same application:
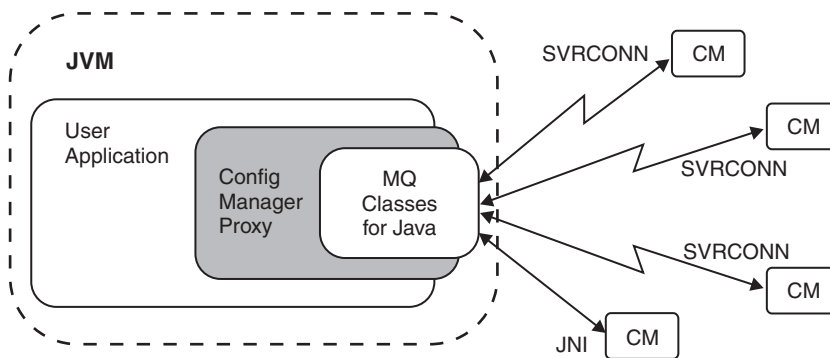
By using the CMP API, you can connect to and manipulate a Configuration Manager in the following products:
- WebSphere Message Broker Version 6.1
- WebSphere Message Broker Version 6.0
- WebSphere Event Broker Version 6.0
- WebSphere Business Integration Message Broker Version 5.0
- WebSphere Business Integration Message Broker with Rules and Formatter Extension Version 5.0

- WebSphere Business Integration Event Broker Version 5.0

A domain controlled by a Version 5.0 or Version 6.0 Configuration Manager can include Version 5.0, Version 6.0, and Version 6.1 brokers; you can deploy to all versions from a single CMP API application.

The Configuration Manager Proxy samples

Explore the samples to learn the basic features that are provided by the Configuration Manager Proxy. Run the samples to deploy a BAR file or manage a broker domain, or use the CMP API Exerciser to implement various tasks.

Deploy BAR

The Deploy BAR sample deploys a BAR file to an execution group, and displays the outcome. Read about this sample in "Running the CMP Deploy BAR sample."

Broker domain management

The broker domain management sample uses the CMP API to display to the screen the complete run state of the domain. "Running the CMP broker domain management sample" on page 6 describes this sample in more detail.

The CMP API Exerciser

The CMP API Exerciser is a graphical interface to the CMP that you can use to view and manipulate brokers. Use the CMP API Exerciser sample to view and manage a Configuration Manager, or record and play back configuration scripts. You can also customize the CMP API Exerciser to tailor the tasks to suit your requirements. See "Running the CMP API Exerciser sample" on page 7 for more information.

Modify CMP API samples

You can modify the CMP API samples, and change various parameters that affect how the samples run. Read "Modifying the CMP samples" on page 11 and follow the guidance given about possible changes.

Running the CMP Deploy BAR sample

Run the CMP Deploy BAR sample to deploy a broker archive file to a Configuration Manager.

Before you start:

The BAR file and execution group name, and the connection details that define the target Configuration Manager, are hard coded into the application. You can run this sample unchanged, or you can modify the values and parameters that it uses to apply them to your own configuration.

If you modify the Deploy BAR sample, you must update and recompile the source file before you run it. The source file for this sample is located in the following directory:

`install_dir/sample/ConfigManagerProxy/cmp/DeployBar.java`

Use the Deploy BAR sample to deploy a BAR file to an execution group, and display the outcome.

1. Run the Deploy BAR sample by entering the appropriate command for your platform:

   - **Windows** On Windows®, open a Command Console and run the following command:

     `install_dir\sample\ConfigManagerProxy\StartDeployBAR.bat`

   - **Linux** **UNIX** **z/OS** On other platforms:

     a. Start a broker command environment by running mqsiprofile, or follow the guidance provided in the `StartDeployBar` shell script to configure the correct CLASSPATH for your environment.

     b. Run the shell script:

        `install_dir\sample\ConfigManagerProxy\StartDeployBAR`

   The default connection parameters used by the sample are shown in the following table.

Connection parameter	Description
`"localhost"`	Host name of the computer where the Configuration Manager is running.
`"BROKER"`	Name of the Configuration Manager queue manager.
`1414`	Port on which the Configuration Manager queue manager is listening
`"BROKER"`	Name of the broker.
`"default"`	Name of the execution group.
`"c://mybar.bar"`	Fully qualified name of the BAR file to deploy.

   The CMP connects to the Configuration Manager that is running on the local computer (defined by `localhost`). The queue manager `BROKER` must be listening on port 1414. Next, the CMP deploys the file `mybar.bar` to the predefined execution group `default` on broker `BROKER`.

2. Check the results of the sample by viewing the broker in the workbench, or by using the CMP API Exerciser.

Next: Run another sample, or work with the CMP API Exerciser.

Running the CMP broker domain management sample

Run the broker domain management sample to display the complete run state of a broker domain.

Before you start:

You can run this sample unchanged, or you can modify the values and parameters that it uses to apply them to your own configuration.

If you modify this management sample, you must update and recompile the source file before you run it. The source file for this sample is located in the following directory:

`install_dir/sample/ConfigManagerProxy/cmp/DomainInfo.java`

Use the broker domain management sample to display the complete run state of the broker domain.

- Run the broker domain management sample by entering the appropriate command for your platform. Replace *connection_file* with the fully qualified file path to the Configuration Manager file (with extension `.configmgr`).

  – `Windows` On Windows, use the Command Console to run the following command:

  install_dir`\sample\ConfigManagerProxy\StartDomainInfo.bat` *connection_file*

  – `Linux` `UNIX` `z/OS` On other platforms:

  1. Start a broker command environment by running mqsiprofile, or follow the guidance provided in the `StartDomainInfo` shell script to configure the correct CLASSPATH for your environment.

  2. Run the shell script:

     install_dir`\sample\ConfigManagerProxy\StartDomainInfo` *connection_file*

  The complete run state of the broker domain is displayed. You can see responses like the following output:

  ```
  (13/08/04 15:47:37) Connecting. Please wait...
  (13/08/08 15:47:38) Successfully connected to the Configuration Manager's
                      Queue Manager.
  (13/08/08 15:47:39) Successfully connected to the Configuration Manager.
  (13/08/08 15:47:41) Broker 'BROKER' is running.
  (13/08/08 15:47:42)   Execution group 'default' on 'BROKER' is running.
  (13/08/08 15:47:43)     Message flow 'flow1' on 'default' on 'BROKER' is
                          running.
  (13/08/08 15:47:44) Disconnected.
  ```

- If you prefer, you can run this sample in interactive mode, which causes the sample to listen for changes to the broker domain.

  1. To run the sample in interactive mode, specify the -i option on the command.

     For example:

     ```
     \sample\ConfigManagerProxy\StartDomainInfo.bat
      c:\myConfigMgr.configmgr  -i
     ```

     In interactive mode, you see the output shown earlier, and the additional response:

     ```
     (13/08/08 15:53:46) Listening for changes to the domain...
     ```

  2. To stop the sample when it is running in interactive mode, force it to end by using CTRL+C.

Next: Run another sample, or work with the CMP API Exerciser.

Running the CMP API Exerciser sample

Run the CMP API Exerciser sample to view and manage a Configuration Manager, customize the CMP API Exerciser, or record and play back configuration scripts.

Before you start:

You can run this sample unchanged, or you can modify the values and parameters that it uses to apply them to your own configuration.

If you modify the CMP API Exerciser sample, you must update and recompile the source file before you run it. The source file for this sample is located in the following directory:
install_dir/sample/ConfigManagerProxy/cmp/exerciser

Use the CMP API Exerciser to complete the following tasks:

- "Viewing and managing a broker domain in the CMP API Exerciser"
- "Customizing the CMP API Exerciser" on page 9
- "Recording and playing back configuration scripts using the CMP API Exerciser" on page 10

Viewing and managing a broker domain in the CMP API Exerciser

Use the CMP API Exerciser sample to view and manage a broker domain.

To view and manage a broker domain by using the CMP API Exerciser, complete the following steps:

1. Start the CMP API Exerciser:

   - Windows On Windows, click **Start** → **IBM WebSphere Message Broker 6.1** → **Java Programming APIs** → **Configuration Manager Proxy API Exerciser**.

   - Linux UNIX z/OS On other platforms:

     a. Start a broker command environment by running mqsiprofile, or follow the guidance provided in the StartConfigManagerProxyExerciser shell script to configure the correct CLASSPATH for your environment.

     b. Ensure that your user ID has writer permission to the current directory. The CMP API Exerciser stores its configuration settings in a file in this directory.

     c. Run the shell script:

        `install_dir\sample\ConfigManagerProxy\StartConfigManagerProxyExerciser`

   The CMP API Exerciser window opens.

2. Connect to a running Configuration Manager by clicking **File** → **Connect to Configuration Manager**.

   The Connect to a Configuration Manager dialog opens.

3. Enter the connection parameters to the Configuration Manager, then click **Submit**.

   Broker domain information is retrieved and displayed in the CMP API Exerciser window. You have now connected to the Configuration Manager.

   The upper left of the screen contains a hierarchical representation of the broker domain to which you are connected. Selecting objects in the tree causes the table on the right to change, reflecting the attributes of the object that you select. The Method column lists CMP API methods that you can call in your own Java applications, and the Result column indicates the data that is returned by calling the CMP API method on the selected object.

4. Run a CMP API method against a broker object. CMP API methods are used to manage objects in a broker domain.

   a. In the navigation tree, right-click a broker.

      A context-sensitive menu opens to show all the available CMP API methods.

   b. Select **List connections**.

      Information is displayed in the log view of the CMP API Exerciser window. For example, the following output could be displayed:

```
(12/08/08 18:24:45) ----> cmp.exerciser.ClassTesterForBrokerProxy.
                     testListConnections(<B1>)
(12/08/08 18:24:45) There are no connections defined.
(12/08/08 18:24:45) <---- cmp.exerciser.ClassTesterForBrokerProxy.
                     testListConnections
```

The first line indicates that the method
`cmp.exerciser.ClassTesterForBrokerProxy.testListConnections()` was
called with the parameter of the AdministeredObject representing the
broker, B1. The second line is output from the method, and the third line
indicates that the method completed.

The available CMP API methods are used to manage the broker domain.

During these steps you connected to a broker domain, viewed the broker domain
information, and performed a management task by using the CMP API Exerciser.

Next: Continue to work with the CMP API Exerciser, or run another sample.

Customizing the CMP API Exerciser

Enable or disable a selection of options to customize the CMP API Exerciser to
meet your requirements.

To customize the CMP API Exerciser, complete the following steps:

1. Start the CMP API Exerciser.

   - Windows On Windows, click **Start → IBM WebSphere Message Broker 6.1 →
     Java Programming APIs → Configuration Manager Proxy API Exerciser**.

   - Linux UNIX z/OS On other operating systems, run the following
     shell script:

     `install_dir\sample\ConfigManagerProxy\StartConfigManagerProxyExerciser`

   The CMP API Exerciser window opens.

2. Customize the CMP API Exerciser by selecting one or more of the following
   options from the **File** menu.

   a. Optional: Click **File → Discover Subcomponent Tree Recursively** to enable
      or disable this option.

      - If you enable this option, the CMP API Exerciser connects to a
        Configuration Manager and discovers all defined broker domain objects.

      - If you disable this option, only the top-level objects are discovered; you
        must select the context-sensitive option, **Discover subcomponents**, to
        iterate down the object hierarchy.

   b. Optional: Click **File → Use Incremental Deployment** to enable or disable
      this option.

      - If you enable this option, all deploy operations cause a delta
        (incremental) deploy where relevant.

      - If you disable this option, all deploy operations cause a complete deploy.

   c. Optional: Click **File → Show Advanced Properties** to enable or disable this
      option.

      - If you enable this option, output from all available methods is displayed
        in the right pane of the CMP API Exerciser.

      - If you disable this option, output from a subset of the available methods
        is displayed in the right pane of the CMP API Exerciser.

   d. Optional: Click **File → Connect Using .configmgr Properties File** to enable
      or disable this option.

      - If you enable this option, a file dialog opens when you connect to a
        Configuration Manager. Use the file dialog to navigate to a file with a
        `.configmgr` extension, which provides the connection parameters to the
        queue manager that hosts the Configuration Manager.

- If you disable this option, you are prompted to enter the queue manager connection parameters, security exit parameters, host name, and port.

e. Optional: Click **File → Enable MQ Java Client Service Trace** to enable or disable this option.

- If you enable this option, a level 5 service trace of the WebSphere MQ Classes for Java runs. A trace dialog opens; specify a file name to which trace records are written.
- If you disable this option, level 5 service tracing of the WebSphere MQ Classes for Java is not done.

f. Optional: Click **File → Enable Config Manager Proxy Service Trace** to enable or disable this option.

- If you enable this option, a service trace of the CMP API run. A trace dialog opens; specify a file name to which trace records are written.
- If you disable this option, service tracing of the CMP API is not done.

g. Optional: Click **File → Set Timeout Characteristics**.

Specify the time, in seconds, that the CMP API Exerciser waits for responses from the Configuration Manager and brokers. The default wait interval is 6 seconds.

Next: Continue to work with the CMP API Exerciser, or run another sample.

Recording and playing back configuration scripts using the CMP API Exerciser

Use the CMP API Exerciser to record and play back configuration scripts.

You can run a script file from the command line, a shell window, or from a batch file. If you run the script by using one of these methods, ensure the first action completed by the script is to connect to a Configuration Manager.

1. Start the CMP API Exerciser.

   - � Windows ▸ On Windows, click **Start → IBM WebSphere Message Broker 6.1 → Java Programming APIs → Configuration Manager Proxy API Exerciser**.

   - ▸ Linux ▸ UNIX ▸ z/OS ▸ On other operating systems, run the following shell script:

     install_dir\sample\ConfigManagerProxy\StartConfigManagerProxyExerciser

   The CMP API Exerciser window opens.

2. Start recording a script by clicking **Scripting → Record New Script**. The Save dialog opens. Type a name for the script file, select an appropriate file location, and click **Save**.

   a. Complete one or more actions on a Configuration Manager by using the CMP API Exerciser.

      If you start recording before you run actions against the Configuration Manager, the first action taken is to connect to the Configuration Manager; however, you can start recording a script at any point during the management of a Configuration Manager.

   b. Optional: Insert a pause by clicking **Scripting → Insert a pause**.

      The CMP API Exerciser pauses so that responses can be returned before the next action is issued. Use this option to avoid naming conflicts if you are deleting and re-creating objects of the same name.

      The Insert a pause dialog opens; specify the duration of the pause in seconds.

c. Stop recording the script by clicking **Scripting** ⇾ **Stop Recording**.
   Information about the actions taken are saved to the script file.
3. To replay the script file:
   a. Click **Scripting** ⇾ **Play Back Recorded Script**.
      The Open dialog opens.
   b. Select the appropriate script file and click **Open**. The script file is replayed.

Next: Continue to work with the CMP API Exerciser, or run another sample.

Modifying the CMP samples

Modify the CMP samples to change the parameters that they use to complete their tasks.

If you change a sample, you must recompile the source code to implement those changes. For example, you might choose to change the default connection parameters. If you modify these and other values, you change the behavior of the sample.

To modify a sample, perform the following steps:
1. Set up the environment by following the instructions provided in "Configuring an environment for developing and running CMP applications."
2. Locate the source file for the sample that you want to change.
   The source files for the samples are located in the following directories:

   Deploy BAR sample
   > *install_dir*/sample/ConfigManagerProxy/cmp/DeployBar.java

   Broker domain management sample
   > *install_dir*/sample/ConfigManagerProxy/cmp/DomainInfo.java

   CMP API Exerciser sample
   > *install_dir*/sample/ConfigManagerProxy/cmp/exerciser.java

3. Open the source file, and modify the appropriate parameters.
4. Save and recompile the source file.

Next: Run the modified sample by following the instructions in the appropriate link:
- "Running the CMP Deploy BAR sample" on page 5
- "Running the CMP broker domain management sample" on page 6
- "Running the CMP API Exerciser sample" on page 7

Configuring an environment for developing and running CMP applications

Prepare the environment in which you want to run your CMP your applications.

Before you start:

To develop and run Java applications that use the CMP API, you must install the following prerequisite software in your local computer environment:
- The WebSphere MQ Classes for Java.
  These classes provide the internal wire protocol that your applications use to communicate with the Configuration Manager.

You can install the classes from the media supplied with WebSphere Message Broker.
- A Java Development Kit (JDK) at a supported Java level. Java support is defined in Additional software requirements.

  WebSphere Message Broker does not supply a JDK; you must acquire and install a suitable product yourself.

To set up your computer in preparation for building and running CMP applications, you must configure your CLASSPATH environment variable so that it includes the WebSphere MQ Classes for Java, and the JAR file that defines the CMP.

Follow the instructions provided for the appropriate environment:
- "Configuring the Windows command-line environment to run CMP applications"
- "Configuring Linux, UNIX, and z/OS command-line environments to run CMP applications" on page 13
- "Configuring the Eclipse environment to run CMP applications" on page 13

You can also run CMP applications, and therefore control one or more Configuration Manager components, from computers on which you have not installed WebSphere Message Broker. For more information, see "Configuring environments without the broker component installed" on page 14.

The JAR file `ConfigManagerProxy.jar` contains the English message catalog for displaying broker (BIP) messages from the Event log of the Configuration Manager. If you want a CMP application to display broker messages in a language other than English, you must also add the directory that contains the localized message catalogs to your CLASSPATH; for example, `C:\Program Files\IBM\MQSI\6.1\ messages` on Windows 2003.

You can also use the CMP to display or log messages from a catalog that you create yourself.

Configuring the Windows command-line environment to run CMP applications

Use system facilities to configure environment variables on your Windows computers to run your CMP applications.

Update the CLASSPATH environment variable:
1. Add the CMP JAR file to your CLASSPATH. For example:

`set CLASSPATH = %CLASSPATH%;%install_dir%\classes\ConfigManagerProxy.jar`

2. Add the WebSphere MQ Classes for Java JAR file `com.ibm.mq.jar` to your CLASSPATH in the same way. If you have installed WebSphere MQ Version 6 on this computer, you must also add the file `connector.jar`.

   On 32-bit operating system editions, the files are typically stored in the directory `C:\Program Files\IBM\WebSphere MQ\java\lib`. On 64-bit operating system editions, the files are typically stored in the directory `C:\Program Files (x86)\IBM\WebSphere MQ\java\lib`.

   For more information about the WebSphere MQ Classes for Java, see the *Using Java* section in the WebSphere MQ information center for the version that you are using.

3. Add your Java development directory to the CLASSPATH in the same way.

Next: Use the tools that are provided by your JDK to build and run your CMP applications.

Configuring Linux, UNIX, and z/OS command-line environments to run CMP applications

Use system facilities to configure environment variables on your Linux®, UNIX®, and z/OS® computers to run your CMP applications.

Update the CLASSPATH environment variable:

1. Add the CMP JAR to your CLASSPATH. For example:

```
export CLASSPATH = $CLASSPATH%;$install_dir/sample/ConfigManagerProxy/ConfigManagerProxySamples.jar
```

2. Add the WebSphere MQ Classes for Java JAR file `com.ibm.mq.jar` to your CLASSPATH in the same way. If you have installed WebSphere MQ Version 6 on this computer, you must also add the file `connector.jar`.

   On 32-bit operating system editions, the files are typically stored in the directory `C:\Program Files\IBM\WebSphere MQ\java\lib`. On 64-bit operating system editions, the files are typically stored in the directory `C:\Program Files (x86)\IBM\WebSphere MQ\java\lib`.

   For more information about the WebSphere MQ Classes for Java, see the *Using Java* section in the WebSphere MQ information center for the version that you are using.

3. Add your Java development directory to the CLASSPATH in the same way.

Next: Use the tools that are provided by your JDK to build and run your CMP applications.

Configuring the Eclipse environment to run CMP applications

Use Eclipse facilities to configure the environment to run your CMP applications.

Update the CLASSPATH environment variable:

1. In your Eclipse environment, select **File** ⇒ **New** ⇒ **Project**. The New Project wizard opens.
2. Select **Java Project** from the options displayed. Click **Next**. The New Java Project window opens.
3. Enter a name for your new project. Click **Next**.
4. Select the **Libraries** tab, and click **Add External Jars**.
5. To set up the build environment, navigate to the *install_dir*/`classes` subdirectory. For example, on Windows 32-bit operating system editions, navigate to the directory `C:\Program Files\IBM\MQSI\6.1\classes`. Select the file `ConfigManagerProxy.jar`, and click **Open**. The file is added to the list in the window for the **Libraries** tab.
6. When you have added the file `ConfigManagerProxy.jar` to the build path, set up the runtime environment. Click **Add External Jars** again, and navigate to the *WebSphere_MQ_installation_directory*/`java/lib` subdirectory. For example, on Linux on x86, navigate to the directory `/opt/mqm/java/lib`.

   a. If you have installed WebSphere MQ Version 7, select the file `com.ibm.mq.jar`, and click **Open**.
   b. If you have installed WebSphere MQ Version 6, select the files `com.ibm.mq.jar` and `connector.jar`, and click **Open**.

The file or files you selected are also added to the list.

For more information about the WebSphere MQ Classes for Java, see the *Using Java* section in the WebSphere MQ information center for the version that you are using.

7. Click **Finish** to close the New Project wizard.

Next: Use the Eclipse development tools to build and run your CMP applications.

Configuring environments without the broker component installed

Install and run CMP applications and other utilities on computers, when you are using a local ID only, on which you have not installed the broker component.

You can run Java applications that use the CMP API even where you have not installed the broker component. These CMP applications include your own applications, and the following command utilities:

- mqsicreateexecutiongroup
- mqsideleteexecutiongroup
- mqsideploy on Windows platforms only
- mqsimode
- mqsireloadsecurity; the CMP application version of mqsireloadsecurity is called mqsireloadsecurityscript.
- mqsistartmsgflow
- mqsistopmsgflow

To install CMP applications in an environment that does not have the broker component installed, complete the following steps:

1. Ensure that the target computer has a compatible Java Runtime Environment (JRE). Because you are not installing the broker component, which includes a JRE, you must use an alternative option.

   Java support is defined in Additional software requirements.

2. Copy the following set of files from a computer that has the broker component installed to the target computer:

   a. `ConfigManagerProxy.jar` from the `classes` directory.

   b. The WebSphere MQ Classes for Java.

      - On Windows, these classes are located in the file `com.ibm.mq.jar`.
      - On other platforms, these classes are located in the component's installation image.

   c. Your CMP application and all configuration files, for example `.configmgr` files.

   d. If you want to run the broker commands on the target computer, complete the following steps:

      1) Copy `ConfigUtil.jar` from the `classes` directory.

      2) Copy the required utility bat files, or shell scripts, from the `bin` directory. Shell scripts have a `.bat` extension on Windows or no extension on UNIX platforms:

         - `mqsicreateexecutiongroup` or `mqsicreateexecutiongroup.bat`
         - `mqsideleteexecutiongroup` or `mqsideleteexecutiongroup.bat`
         - `mqsideploy` or `mqsideploy.bat`

• mqsimode or mqsimode.bat

• mqsireloadsecurity or mqsireloadsecurity.bat

• mqsistartmsgflow or mqsistartmsgflow.bat

• mqsistopmsgflow or mqsistopmsgflow.bat

    e. If you want to display broker (BIP) messages in English environments other than US English, copy all BIPv610*.properties files from the messages directory.

3. On the target computer, use system facilities to update the CLASSPATH environment variable to include the following files:

   • The JAR file that contains the definitions of the CMP classes, ConfigManagerProxy.jar.

   • Your applications that import the CMP classes.

   • The WebSphere MQ Classes for Java, com.ibm.mq.jar, and any additional JAR files required by this package.

   • Any other required JAR files and directories. For example, if you require any of the available command utilities on the target computer, include ConfigUtil.jar; if you require the broker (BIP) messages to be displayed in locales other than US English, include a directory that contains BIPv610*.properties.

4. Ensure that the user ID that the target computer uses has the following authorities:

   • Authority to connect to the queue manager that the Configuration Manager uses.

   • Authority to manipulate broker domain objects.

Next: You can run your CMP applications, and the specified command utilities, on the target computer.

Connecting to a Configuration Manager using the Configuration Manager Proxy

Connect a CMP API application to a Configuration Manager to send requests about resources in the broker domain.

Before you start

Before starting this step, you must have completed "Configuring an environment for developing and running CMP applications" on page 11.

Consider the following program ConnectToConfigManager.java. It connects to a Configuration Manager that is running on the default queue manager on the local computer.

```
import com.ibm.broker.config.proxy.*;

public class ConfigManagerRunStateChecker {

    public static void main(String[] args) {
        displayConfigManagerRunState("localhost", 1414, "");
    }

    public static void displayConfigManagerRunState(String hostname,
                                                    int port,
                                                    String qmgr) {
        ConfigManagerProxy cmp = null;
```

```
        try {
            ConfigManagerConnectionParameters cmcp =
                new MQConfigManagerConnectionParameters(hostname, port, qmgr);
            cmp = ConfigManagerProxy.getInstance(cmcp);
            String configManagerName = cmp.getName();

            System.out.println("Configuration Manager '"+configManagerName+
                "' is available!");
            cmp.disconnect();
        } catch (ConfigManagerProxyException ex) {
            System.out.println("Configuration Manager is NOT available"+
                " because "+ex);
        }
    }
}
```

The first line of the program requests Java to import the CMP API classes. All CMP API classes are in the com.ibm.broker.config.proxy package.

The first line inside the try block of the displayConfigManagerRunState() method instantiates a `ConfigManagerConnectionParameters` object. This method is an interface which states that implementing classes are able to provide the parameters to connect to a Configuration Manager.

The only class that implements this interface is `MQConfigManagerConnectionParameters`, which defines a set of WebSphere MQ connection parameters. The constructor used here takes three parameters:
1. The host name of the Configuration Manager computer
2. The port on which the WebSphere MQ listener service for the Configuration Manager is listening
3. The name of the queue manager that is associated with the Configuration Manager

When you have defined this object, you can connect to the queue manager with those characteristics. The connection is achieved by the static getInstance() factory method just inside the `try` block. When a valid handle to the Configuration Manager is obtained, the application attempts to discover the name of the Configuration Manager (cmp.getName()) and display it.

getName(), and other methods that request information from the Configuration Manager, block until the information is supplied, or a timeout occurs. Therefore, if the Configuration Manager is not running, the application hangs for a period. You can control the timeout period by using the ConfigManagerProxy.setRetryCharacteristics() method. Typically, blocking only occurs when a given resource is accessed for the first time within an application.

Finally, the disconnect() method is called. This method frees up resources associated with the connection in both the CMP API and Configuration Manager.

When a `ConfigManagerProxy` handle is first returned from the getInstance() method, the Configuration Manager service is not necessarily running. It is only when the application uses the handle (by calling getName() in this example) that the application can be assured that a two-way connection with the Configuration Manager is active.

Navigating broker domains using the Configuration Manager Proxy

Explore the status and attributes of the Configuration Manager that your application is connected to, and discover information about the broker domain and its resources.

Before you start

Before starting this step, you must have completed "Connecting to a Configuration Manager using the Configuration Manager Proxy" on page 15.

Each resource that the Configuration Manager can control is represented as a single object in the Configuration Manager Proxy. An application can request status and other information about all resources, including the following objects:

* Brokers
* Execution groups
* Deployed message flows
* Topics
* Collectives
* Subscriptions
* Publish/Subscribe topology
* Broker event log

The Configuration Manager Proxy also handles deployed message sets; these resources are handled as attributes of deployed execution groups.

Collectively known as *administered objects*, these objects provide most of the interface to the Configuration Manager. They are therefore fundamental to an understanding of the Configuration Manager Proxy.

Each administered object is an instance of a Java class that describes the underlying type of object in the Configuration Manager. The valid Java classes are shown in the following table.

Java class	Class function
TopologyProxy	Describes the publish/subscribe topology.
CollectiveProxy	Describes publish/subscribe collectives.
BrokerProxy	Describes brokers.
ExecutionGroupProxy	Describes execution groups.
MessageFlowProxy	Describes message flows that have already been deployed to execution groups; does NOT describe message flows in the Broker Application Development perspective of the workbench.
TopicProxy	Describes topics.
TopicRootProxy	Describes the root of the topic hierarchy.
LogProxy	Describes the broker event log for the current user.
SubscriptionsProxy	Describes a subset of the active subscriptions.
ConfigManagerProxy	Describes the Configuration Manager itself.

Each administered object describes a single object that can be controlled by the Configuration Manager. For example, each execution group within a broker is represented by one instance of the class ExecutionGroupProxy in that application.

A set of public methods is declared in each administered object. Applications can use these methods to inquire and manipulate properties of the underlying Configuration Manager to which the object instance refers. For example, the application can call methods against the object B1, of class BrokerProxy, to request the broker run-state, to start all its message flows, and so on.

To access an administered object through its API, your application must first request a handle to that object from the object that logically owns it. For example, because brokers logically own execution groups, the application must request a handle to the object EG1, of class ExecutionGroupProxy, from the object B1, of class BrokerProxy. It uses the handle to access this execution group and its state and properties.

In the ConnectToConfigManager example, a handle is gained to the object of class ConfigManagerProxy. This object is logically the root of the administered object tree, therefore you can access all other objects in the Configuration Manager directly, or indirectly, from it.

The Configuration Manager directly owns the Publish/Subscribe topology, therefore applications can call a method on objects of the class ConfigManagerProxy to gain a handle to the objects of the class TopologyProxy. Similarly, the topology logically contains the set of all brokers, therefore the application can call methods on objects of the class TopologyProxy to access objects of the class BrokerProxy.

The complete hierarchy of these access relationships is shown in the following diagram.
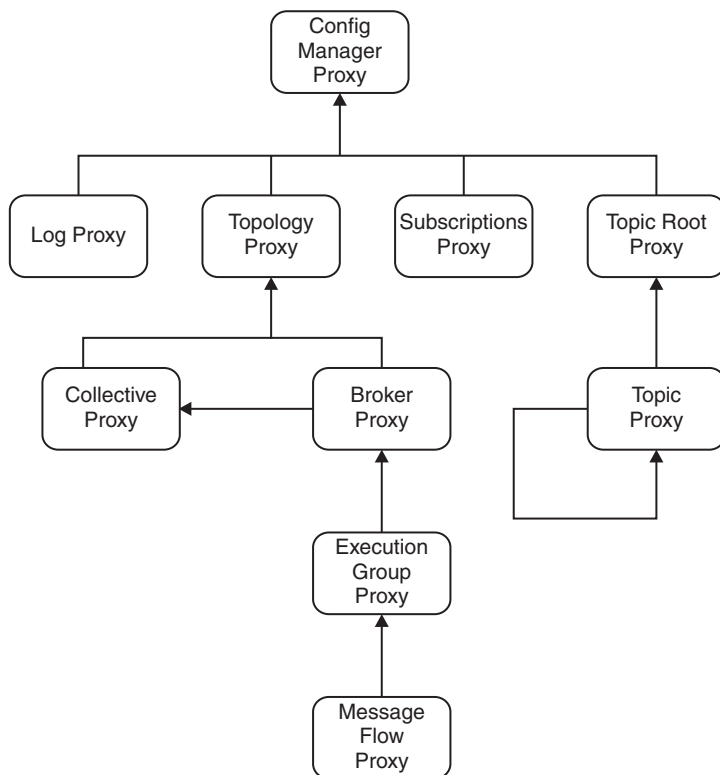
The following program traverses the administered object hierarchy to discover the run state of a deployed message flow. The program assumes that message flow MF1 is deployed to execution group EG1 on broker B1; you can substitute these values in the code for other values that are valid in your domain.

```
import com.ibm.broker.config.proxy.*;

public class GetMessageFlowRunState {

  public static void main(String[] args) {

    ConfigManagerProxy cmp = null;
    try {
      ConfigManagerConnectionParameters cmcp =
        new MQConfigManagerConnectionParameters(
          "localhost",
          1414,
          "");
      cmp = ConfigManagerProxy.getInstance(cmcp);
    } catch (ConfigManagerProxyException cmpex) {
      System.out.println("Error connecting: "+cmpex);
    }

    if (cmp != null) {
      System.out.println("Connected to Config Manager!");
      displayMessageFlowRunState(cmp, "B1", "EG1", "MF1");
      cmp.disconnect();
    }
  }

  private static void displayMessageFlowRunState(
                                  ConfigManagerProxy cmp,
                                  String brokerName,
                                  String egName,
                                  String flowName) {
    try {
      TopologyProxy topology = cmp.getTopology();

      if (topology != null) {
        BrokerProxy b = topology.getBrokerByName(brokerName);

        if (b != null) {
          ExecutionGroupProxy eg =
            b.getExecutionGroupByName(egName);

          if (eg != null) {
            MessageFlowProxy mf =
              eg.getMessageFlowByName(flowName);

            if (mf != null) {
              boolean isRunning = mf.isRunning();
              System.out.print("Flow "+flowName+" on " +
                egName+" on "+brokerName+" is ");

              if (isRunning) {
                System.out.println("running");
              } else {
                System.out.println("stopped");
              }
            } else {
              System.err.println("No such flow "+flowName);
            }
          } else {
            System.err.println("No such exegrp "+egName+"!");
          }
        } else {
          System.err.println("No such broker "+brokerName);
```

```
      }
    } else {
      System.err.println("Topology not available!");
    }
  } catch(ConfigManagerProxyPropertyNotInitializedException
                                               ex) {
    System.err.println("Comms problem! "+ex);
  }
  }
}
```

The method that does most of the work is displayMessageFlowRunState(). This
method uses the handle to the object of class ConfigManagerProxy, gained earlier,
and discovers the run-state of the message flow in the following way:

1. The instance of the ConfigManagerProxy class is used to gain a handle to the
    instance of the TopologyProxy class. Because only one topology exists for each
    Configuration Manager, you do not have to supply an identifier to qualify the
    getTopology() method.

2. If a valid topology is returned, its handle is used to gain a handle to an object
    of the class BrokerProxy. The object required is identified by the name defined
    by the string brokerName.

3. If a valid broker is returned, its handle is used to gain a handle to an object of
    the class ExecutionGroupProxy. The object required is identified by the name
    defined by the string egName.

4. If a valid execution group is returned, its handle is used to gain a handle to an
    object of the class MessageFlowProxy. The object required is identified by the
    name defined by the string flowName.

5. If a valid message flow is returned, its handle is used to request the run-state
    of the message flow, and the result is displayed.

The application does not have to know the names of objects that it will
manipulate. Each administered object contains methods to return sets of objects
that it logically owns. The following example demonstrates this technique by
looking up the names of all brokers within the domain:

```
import java.util.Enumeration;
import com.ibm.broker.config.proxy.*;

public class DisplayBrokerNames {

  public static void main(String[] args) {

    ConfigManagerProxy cmp = null;
    try {
      ConfigManagerConnectionParameters cmcp =
        new MQConfigManagerConnectionParameters(
          "localhost",
          1414,
          "");
      cmp = ConfigManagerProxy.getInstance(cmcp);
    } catch (ConfigManagerProxyException cmpex) {
      System.out.println("Error connecting: "+cmpex);
    }

    if (cmp != null) {
      System.out.println("Connected to Config Manager!");
      displayBrokerNames(cmp);
      cmp.disconnect();
    }
  }
```

```
  private static void displayBrokerNames(ConfigManagerProxy cmp)
  {
    try {
      TopologyProxy topology = cmp.getTopology();

      if (topology != null) {
        Enumeration allBrokers = topology.getBrokers(null);

        while (allBrokers.hasMoreElements()) {
          BrokerProxy thisBroker =
            (BrokerProxy) allBrokers.nextElement();
          System.out.println("Broker "+thisBroker.getName());
        }
      }
    } catch(ConfigManagerProxyPropertyNotInitializedException
                                              ex) {
        System.err.println("Comms problem! "+ex);
    }
  }
}
```

The key method is TopologyProxy.getBrokers(Properties). When supplied with a null argument, this method returns an enumeration of all objects of the class BrokerProxy in the domain. The application uses this method to look at each broker proxy object in turn, and display its name.

The Properties argument of the method TopologyProxy.getBrokers(Properties) can be used to specify the exact characteristics of the brokers that are required. The application can use this argument for nearly all the methods that return administered objects, and is a powerful way of filtering those objects with which the application needs to work.

Examples of those characteristics that can be used to filter object lookup operations are the run-state and short description, as well as more obvious properties such as the name and UUID. To write logic to achieve this objective, you must understand how each administered object stores its information.

The properties of each administered object are stored locally inside the object in a hash table, where each property is represented as a {key, value} tuple. Each key is the name of an attribute (for example, name) and each value is the value (for example, BROKER1).

Each key name must be expressed by using a constant from the AttributeConstants class (com.ibm.broker.config.proxy). A complete set of keys and possible values for each administered object is described in the Java documentation for the AttributesConstant class, or by using the Show raw property table for this object function in the Configuration Manager Proxy API Exerciser sample program. The latter displays the complete list of {key, value} pairs for each administered object.

The Properties argument that is supplied to the lookup methods is a set of those {key, value} pairs that must exist in each administered object in the returned enumeration. Consider the following code fragment:

```
Properties p = new Properties();

p.setProperty(AttributeConstants.OBJECT_RUNSTATE_PROPERTY,
              AttributeConstants.OBJECT_RUNSTATE_RUNNING);

Enumeration e = executionGroup.getMessageFlows(p);
```

If the variable `executionGroup` is a valid object of the class ExecutionGroupProxy, the returned enumeration contains only active message flows (OBJECT_RUN_STATE_PROPERTY equal to OBJECT_RUNSTATE_RUNNING).

When property filtering is applied to a method that returns a single administered object rather than an enumeration of objects, only the first result is returned (which is non-deterministic if more than one match applies). Therefore, the following code:

```
Properties p = new Properties();

p.setProperty(AttributeConstants.NAME_PROPERTY,
              "shares");

TopicProxy t = topicProxy.getTopic(p);
```

is an alternative to the following statement:

```
TopicProxy t = topicProxy.getTopicByName("shares");
```

If multiple {`key, value`} pairs are added to a property filter, all properties must be present in the child object for an object to match. If you want a method to perform a logical OR, or a logical NOT, on a filter, you must write specific application code for this purpose.

When administered objects are first instantiated in an application, the CMP requests the current set of properties for that object from the Configuration Manager. This action occurs asynchronously, therefore the first time a property is requested, the CMP must wait for the information to be supplied by the Configuration Manager. If the information does not arrive within a certain time (for example, if the Configuration Manager is not running), a `ConfigManagerProxyPropertyNotInitializedException` is thrown. The maximum time that the CMP waits is determined by the ConfigManagerProxy.setRetryCharacteristics() method.

Using the Configuration Manager Proxy API to deploy

Deploy to the brokers in your broker domain from a CMP API application.

You can use the Configuration Manager Proxy API for all supported types of deployment.

Deployment type	Description
TopologyProxy.deploy()	Deploys the publish/subscribe topology to all affected brokers.
BrokerProxy.deploy()	Deploys the broker configuration.
ExecutionGroupProxy.deploy()	Deploys a BAR file to an execution group.
TopicRootProxy.deploy()	Deploys the topic hierarchy to all brokers.
ConfigManagerProxy.cancelDeployment()	Cancels all outstanding deploys in the domain.
BrokerProxy.cancelDeployment()	Cancels any outstanding deploy to a specific broker.

The Configuration Manager Proxy API has more information about each of these methods, and you can find an example of the code that you might use for each type of deployment in the appropriate topic in the Deploying section.

You can also check the result of a deployment by using the Configuration Manager Proxy API.

An example

This example adds to the domain a broker called *B2* that is running on queue manager *QMB2* and associates with it an execution group called 'default'. This configuration is then deployed to the broker.

For this example to work successfully, the broker *B2* has been created on the machine running queue manager *QMB2*, and it has not already been deployed to by another Configuration Manager.

```
import com.ibm.broker.config.proxy.*;

public class AddBroker {

  public static void main(String[] args) {
    ConfigManagerProxy cmp = null;
    try {
      ConfigManagerConnectionParameters cmcp =
        new MQConfigManagerConnectionParameters(
        "localhost",
        1414,
        "");
      cmp = ConfigManagerProxy.getInstance(cmcp);
    }
    catch (ConfigManagerProxyException cmpex) {
      System.out.println("Error connecting: "+cmpex);
    }
    if (cmp !=null) {
      System.out.println("Connected to Config Manager");
      addBroker(cmp, "B2", "QMB2", "default");
      cmp.disconnect();
    }
  }

  private static void addBroker(ConfigManagerProxy cmp,
                                String bName,
                                String bQMgr,
                                String egName) {
    TopologyProxy topology = null;
    try {
      topology = cmp.getTopology();
    }
    catch(ConfigManagerProxyPropertyNotInitializedException ex) {
      System.err.println("Comms problem! "+ex);
    }
    if (topology != null) {
      try {
        BrokerProxy b2 = topology.createBroker(bName, bQMgr);
        ExecutionGroupProxy e = b2.createExecutionGroup(egName);
        b2.deploy();
      }

      catch (ConfigManagerProxyException ex) {
        System.err.println("Could not perform an action: "+ex);
      }
    }
  }
}
```

Configuration Manager Proxy Exerciser

You can also use the Configuration Manager Proxy Exerciser to deploy. The exerciser is a graphical interface to the Configuration Manager Proxy that allows you to view and manipulate Configuration Manager domains. For example:

1. Connect to the Configuration Manager: **File → Connect to Configuration Manager**. This action opens the Connect to Configuration Manager dialog.
2. Enter the relevant connection parameters in the dialog. A hierarchical representation of the domain is displayed.
3. You can perform a number of operations. For example:
   - Click an object in the tree to display the attributes of that object.
   - Right-click an object in the tree to call Configuration Manager Proxy methods that manipulate that object. For example, right-clicking a broker opens a drop-down menu that has items such as 'start user trace', 'deploy broker configuration' and 'cancel all outstanding deploys to this broker'.
   - Use the log pane at the bottom of the screen to view useful information relating to the operation being performed.

Checking the results of deployment using the Configuration Manager Proxy

After you have made a deployment, check that the operation has completed successfully on your CMP application.

If you are using a CMP API application, you can find out the result of a publish/subscribe topology deployment operation, for example, by using code like this snippet:

```
TopologyProxy t = cmp.getTopology();

boolean isDelta = true;
long timeToWaitMs = 10000;
DeployResult dr = topology.deploy(isDelta, timeToWaitMs);

System.out.println("Overall result = "+dr.getCompletionCode());

// Display overall log messages
Enumeration logEntries = dr.getLogEntries();
while (logEntries.hasMoreElements()) {
  LogEntry le = (LogEntry)logEntries.nextElement();
  System.out.println("General message: " + le.getDetail());
}

// Display broker specific information
Enumeration e = dr.getDeployedBrokers();
while (e.hasMoreElements()) {

  // Discover the broker
  BrokerProxy b = (BrokerProxy)e.nextElement();

  // Completion code for broker
  System.out.println("Result for broker "+b+" = " +
    dr.getCompletionCodeForBroker(b));

  // Log entries for broker
  Enumeration e2 = dr.getLotEntriesForBroker(b);
  while (e2.hasMoreElements()) {
    LogEntry le = (LogEntry)e2.nextElement();
```

```
    System.out.println("Log message for broker " + b +
      le.getDetail()));
  }
}
```

The `deploy` method blocks other processes until all affected brokers have responded to the deployment request.

When the method returns, the DeployResult object represents the outcome of the deployment at the time when the method returned; the object is not updated by the Configuration Manager Proxy.

If the deployment message cannot be sent to the Configuration Manager, a ConfigManagerProxyLoggedException exception is thrown at the time of deployment. If the Configuration Manager receives the deployment message, log messages for the overall deployment are displayed, followed by completion codes specific to each broker affected by the deployment. The completion code. shown in the following table, is one of the static instances from the CompletionCodeType class.

Completion code	Description
pending	The deployment is held in a batch and is not sent until you call ConfigManagerProxy.sendUpdates().
submitted	The deployment message was sent to the Configuration Manager but no response was received before the timeout period expired.
initiated	The Configuration Manager indicated that deployment has started, but no broker responses were received before the timeout period expired.
successSoFar	The Configuration Manager indicated that deployment has started and some, but not all, brokers responded successfully before the timeout period expired. No brokers responded negatively.
success	The Configuration Manager indicated that deployment has started and all relevant brokers responded successfully before the timeout period expired.
failure	The Configuration Manager indicated that deployment has started and at least one broker responded negatively. You can use getLogEntriesForBroker method of the DeployResult class to get more information about the deployment failure. This method returns an enumeration of available LogEntry objects.
notRequired	The deployment request submitted to the Configuration Manager was not sent to the broker, because the broker configuration is already up to date.

Setting user-defined properties at run time in a CMP application

Use the CMP API to query, discover, and set message flow user-defined properties dynamically at run time.

For user-defined properties on a message flow to be discoverable, the message flow must comply with the following conditions:
- The message flow must contain at least one of the following nodes:
  - JavaCompute
  - Compute
  - Database
  - Filter

– PHPCompute
- The message flow must define the relevant user-defined property and provide an override value.

Tip: Use meaningful names and values for the properties that you define, so that you can understand their purpose and intent quickly. For example, a user-defined property named property01, with an initial value of valueA is not as useful as a property named RouteToAorB with an initial value of RouteA.

To query, discover, and set user-defined properties on a message flow, use the CMP API to issue the following calls. For details about the calls, including the syntax and parameters to use, see the CMP API documentation (Configuration Manager Proxy API).

1. Call MessageFlowProxy.getUserDefinedPropertyNames() to retrieve a list of all the user-defined properties that were defined by the Message Flow editor on the message flow or subflows.

   A string array is returned that contains the property names.

2. Call MessageFlowProxy.getUserDefinedProperty() to retrieve the value of the specified user-defined property.

   The value of the property is returned as a Java.lang.String value.

3. Call MessageFlowProxy.setUserDefinedProperty() to set a new value for the specified user-defined property.

   The property must exist. You cannot change the data type of the existing user-defined property (Java.lang.String); therefore, you must ensure that the new value complies with the existing data type. The value that you set with the MessageFlowProxy.setUserDefinedProperty() call is populated to all relevant nodes in the message flow, including nodes in subflows.

4. Call BrokerProxy.deploy() to refresh the user-defined properties.

   When you use BrokerProxy.deploy() to refresh user-defined properties, the message flows that are affected by the setUserDefinedProperty() call are restarted automatically. The changes to these properties survive if you stop and restart the broker, execution group, or message flow from the command line or workbench. However, the changes do not survive when the message flow BAR file is redeployed. BAR file redeployment redefines the user-defined properties according to the specifications in the BAR file.

Managing broker domains in a CMP application

Manage the brokers in your broker domain from a CMP application.

Before you start

Before you start this task, you must have completed the task "Connecting to a Configuration Manager using the Configuration Manager Proxy" on page 15.

Use the CMP API to change the state of objects in the domain; you can create, delete, modify, and deploy objects stored within the Configuration Manager. The following example sets the long description field of a broker calledB1:

```
import com.ibm.broker.config.proxy.*;

public class SetLongDescription {

  public static void main(String[] args) {
```

```
        ConfigManagerProxy cmp = null;
        try {
          ConfigManagerConnectionParameters cmcp =
              new MQConfigManagerConnectionParameters(
                "localhost",
                1414,
                "");
          cmp = ConfigManagerProxy.getInstance(cmcp);
        } catch (ConfigManagerProxyException cmpex) {
          System.out.println("Error connecting: "+cmpex);
        }

        if (cmp != null) {
          System.out.println("Connected to Config Manager!");
          describeBroker(cmp, "B1", "this is my broker");
          cmp.disconnect();
        }
      }

  private static void describeBroker(ConfigManagerProxy cmp,
                                     String brokerName,
                                     String newDesc)
  {
   BrokerProxy b = null;
    try {
        TopologyProxy topology = cmp.getTopology();
        if (topology != null) {
          b = topology.getBrokerByName(brokerName);
        }
    } catch(ConfigManagerProxyPropertyNotInitializedException
                                              ex) {
          System.err.println("Comms problem! "+ex);
    }

    if (b != null) {
      try {
        b.setLongDescription(newDesc);
      } catch (ConfigManagerProxyException ex) {
       System.err.println("Could not send request to CM: "+ex);
      }
    } else {
      System.err.println("Broker "+brokerName+" not found");
    }
  }
}
```

The setLongDescription() method works by asking the Configuration Manager to
modify a (key, value) property of the broker B1, where the key name represents
the long description tag, and the value is the new long description. Therefore, you
can use the following code as an alternative to calling setLongDescription():

```
Properties p = new Properties();

p.setProperty(AttributeConstants.LONG_DESCRIPTION_PROPERTY,
              newDesc);

b.setProperties(p);
```

When the request to change properties is sent to the Configuration Manager, the
internal properties tables maintained by the CMP are not updated until the
Configuration Manager reports that its copy of the attributes has been changed
successfully. This sequence of events is done to keep all copies of the information
consistent. The process is shown in the following diagram.
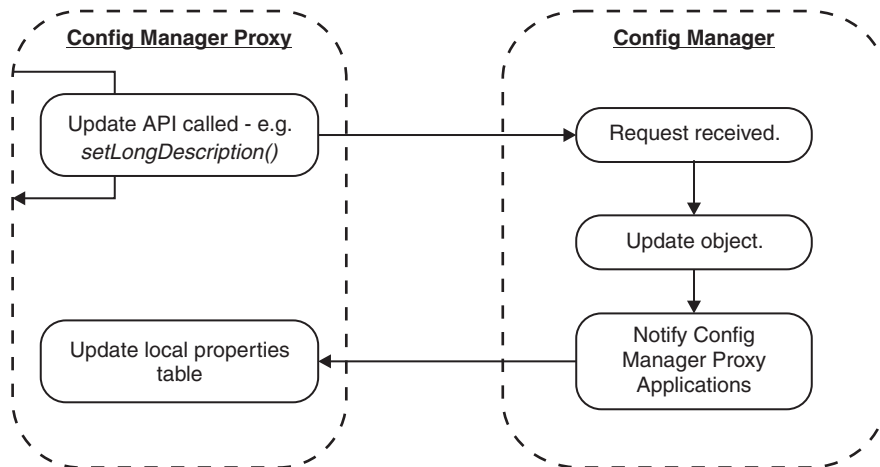
If the current user does not have the necessary permissions, SetLongDescription.java cannot determine if the request gets rejected by the Configuration Manager. The CMP method to set the long description field throws a `ConfigManagerProxyException` only if the message to perform the operation can not be sent to the Configuration Manager.

If the message is sent successfully, the output from the program is exactly the same as it is if the Configuration Manager cannot change the required property. This result occurs because the Configuration Manager processes requests from the CMP asynchronously, and so it could theoretically be a considerable time until the action is performed at the Configuration Manager. If methods such as the one described within this topic did not return control to the program until the completion codes became available, the performance of the CMP application would be wholly dependent on the performance of the Configuration Manager.

Next:

Most state-changing CMP methods return control immediately without informing the calling application of the outcome of the request. To discover this information, see "Checking the results of broker domain management in a CMP application."

Checking the results of broker domain management in a CMP application

When you have made a change to the broker domain, use one of the supplied methods to check if the change has been successful.

Choose one of three methods in the CMP API to determine the outcome of requests to create, delete, modify, and deploy resources:

- If you have initiated a deployment method, you can use the return code from the deployment API; this technique is shown in "Checking the results of broker domain management using the Configuration Manager Proxy with return codes" on page 29.
- You can query an object's most recent completion code; this option is shown in "Checking the results of broker domain management with the most recent completion code in a CMP application" on page 31.
- You can use the administered object notification mechanism; by using this approach, you can code specific routines to handle the responses and take

appropriate action, and improve the efficiency of your program. See "Checking the results of broker domain management with object notification in a CMP application" on page 32.

Checking the results of broker domain management using the Configuration Manager Proxy with return codes

Use return codes from the CMP API calls to determine the outcome of a deployment request.

Only deploy() methods supply a return code that represents the outcome of a request that results in a change of state in the broker domain. The following sample of code shows how to discover the outcome of a topology deploy operation by using the returned `DeployResult` object:

```
...

TopologyProxy t = cmp.getTopology();

boolean isDelta = true;
long timeToWaitMs = 10000;
DeployResult dr = topology.deploy(isDelta, timeToWaitMs);

System.out.println("Overall result = "+dr.getCompletionCode());

// Display overall log messages
Enumeration logEntries = dr.getLogEntries();
while (logEntries.hasMoreElements()) {
    LogEntry le = (LogEntry)logEntries.nextElement();
    System.out.println("General message: " + le.getDetail());
}

// Display broker specific information
Enumeration e = dr.getDeployedBrokers();
while (e.hasMoreElements()) {

    // Discover the broker
    BrokerProxy b = (BrokerProxy)e.nextElement();

    // Completion code for broker
    System.out.println("Result for broker "+b+" = " +
        dr.getCompletionCodeForBroker(b));

    // Log entries for broker
    Enumeration e2 = dr.getLogEntriesForBroker(b);
    while (e2.hasMoreElements()) {
        LogEntry le = (LogEntry)e2.nextElement();
        System.out.println("Log message for broker " + b +
            le.getDetail()));
    }
}
```

In this code the deploy() method is blocked until all affected brokers have responded to the deployment request. However, the method includes a `long` parameter that describes the maximum length of time that the CMP API waits for the responses to arrive.

When the method finally returns, the `DeployResult` object represents the outcome of the deployment at the time that the method returned. In other words, when control is returned to the application, the object has not been updated by the CMP API.

After the deploy() method completes, the example interrogates the returned `DeployResult` object, and displays the overall completion code for the deploy operation. The code can be one of the following values:

(com.ibm.broker.config.proxy.)CompletionCodeType.pending
> The deploy is held in a batch and is not sent until you call ConfigManagerProxy.sendUpdates(). If this message applies it is returned immediately; that is, without waiting for the timeout period to expire.

CompletionCodeType.submitted
> The deploy message was sent to the Configuration Manager, but no response was received before the timeout occurred. If the deployment message cannot be sent to the Configuration Manager, a `ConfigManagerProxyLoggedException` is thrown at deploy time instead.

CompletionCodeType.initiated
> The Configuration Manager replied, stating that deployment started, but no broker responses were received before the timeout occurred.

CompletionCodeType.successSoFar
> The Configuration Manager issued the deployment request and some, but not all, brokers responded with a "success" message before the timeout period expired. No brokers responded negatively.

CompletionCodeType.success
> The Configuration Manager issued the deployment request, and all relevant brokers responded successfully before the timeout period expired. This message is sent as soon as all relevant brokers have responded successfully.

CompletionCodeType.failure
> The Configuration Manager issued the deployment request, and at least one broker responded negatively.

Not all completion codes apply to all deploys. For example, if you deploy to a single, specific, broker, the completion code of `'successSoFar'` is never returned.

The example next displays any log messages from the deployment that cannot be attributed to any specific broker. On a successful deploy, these messages always include a "deploy initiated" log entry that originates from the Configuration Manager, even if the deployment subsequently completed.

Finally, the example displays the completion code and any log messages specific to each broker that is affected by the deployment. If you have called a topology or topic tree deploy, every broker in the domain is included.

The set of completion codes applicable to a response from a specific broker are:

CompletionCodeType.pending
> The deploy is held in a batch, and is not sent until you call ConfigManagerProxy.sendUpdates().

CompletionCodeType.submitted
> The deploy message was sent but no response has yet been received from the Configuration Manager to indicate that deployment has been initiated.

CompletionCodeType.initiated
> The Configuration Manager has replied, stating that deployment has started, but no reply has yet been returned from the broker.

CompletionCodeType.success
> The Configuration Manager issued the deployment request, and the broker successfully applied the deployment changes.

CompletionCodeType.failure
> The Configuration Manager issued the deployment request, and the broker responded by stating that the deployment was not successful. Use getLogEntriesForBroker() for more information about why the deployment failed.

CompletionCodeType.notRequired
> A deployment request was submitted to the Configuration Manager that involved the supplied broker, but the broker was not sent the request because its configuration is already up to date.

See "Running the CMP Deploy BAR sample" on page 5 or "Running the CMP broker domain management sample" on page 6 for use of the CMPAPIExerciser.reportDeployResult() method and examples of how to parse `DeployResult` objects.

Checking the results of broker domain management with the most recent completion code in a CMP application

Use an object's most recent completion code to determine the outcome of a request that your application made against that object.

Most state-changing methods in the CMP API do not provide return code that indicates the success or failure of a specific action. For these methods, you must write different code to discover the outcome of the action. Assuming that administered objects are not shared across threads, the following code fragment can be used to discover the outcome of a request to modify a broker's `LongDescription`, where b is an instance of a `BrokerProxy` object:

```
GregorianCalendar oldCCTime =
                        b.getTimeOfLastCompletionCode();
b.setLongDescription(newDesc);
GregorianCalendar newCCTime = oldCCTime;
while ((newCCTime == null) || (newCCTime.equals(oldCCTime))) {
  newCCTime = b.getTimeOfLastCompletionCode();
  Thread.sleep(1000);
}
CompletionCodeType ccType = b.getLastCompletionCode();
if (ccType == CompletionCodeType.success) {
  // etc.
}
```

In this example, the application initially determines when an action on the broker was last completed, using the getTimeOfLastCompletionCode() method. This method returns the time that the topology last received a completion code or, if no return codes have been received, a null value. The application updates the broker's `LongDescription`, then continually monitors the topology, waiting for the results of the setLongDescription() command to be returned to the CMP. When the results are returned, control breaks out of the `while` loop and the last completion code is determined.

As well as being unsuitable for a multi-threaded application, this algorithm for determining the outcome of commands is inefficient, because it causes the CMP application to wait while the Configuration Manager processes the request.

For a more efficient application, and one that is suitable for a multi-threaded environment, code the alternative approach that uses administered object

notifications; see "Checking the results of broker domain management with object notification in a CMP application."

Checking the results of broker domain management with object notification in a CMP application

Use object notification to determine the outcome of a request that your application made against the object.

The CMP can notify applications whenever commands complete, or whenever changes occur to administered objects. By making use of the OBSERVER design pattern, your CMP application can define a handle to a user-supplied object that has a specific method that is called if an object is modified or deleted, or whenever a response to a previously submitted action is returned from the Configuration Manager.

The user-supplied code must implement the AdministeredObjectListener interface. This interface defines methods that are invoked by the CMP when an event occurs on an administered object to which the listener is registered. The following methods are defined:

processModify(...)

> processModify(...) is called when the administered object to which the listener is registered has one or more of its attributes modified by the Configuration Manager. The following information is included in the notification through the use of the processModify() method arguments:
>
> 1. A handle to the AdministeredObject to which the notification refers.
> 2. A list of strings that contain the key names that have been changed.
> 3. A list of strings that describe new subcomponents that have been created for the object; for example, new execution groups in a broker.
> 4. A list of strings that describe subcomponents that have been removed from the object.
>
> The format of the strings passed to the final two parameters is an internal representation of the administered object. You can turn this representation into an administered object type by using the getManagedSubcomponentFromStringRepresentation() method.
>
> Consider the following additional information:
>
> 1. Strings are passed within these lists to enhance performance; the CMP does not use resource instantiating administered objects, unless they are specifically requested by the calling application.
> 2. The first time you call the processModify() method for a listener, the changed attributes parameter can include a complete set of attribute names for the object, if the application is using a batch method, or if the CMP is experiencing communication problems with the Configuration Manager.

processDelete(...)

> processDelete(...) is called if the object with which the listener is registered is completely removed from the Configuration Manager. Supplied to processDelete(...) is one parameter – a handle to the administered object that has been deleted; when this method returns, the administered object handle might no longer be valid. At about the same time that a processDelete(...) event occurs, a processModify(...) event is sent to listeners of the deleted object's parent, to announce a change in the parent's list of subcomponents.

processActionResponse(...)

> processActionResponse(...) is the event that informs the application that a previous action submitted by that application is complete. Only one processActionResponse(...) event is received for each state-changing operation issued by the CMP application. This event contains the following items of information:
>
> 1. A handle to the administered object for which a request was submitted.
> 2. The completion code of the request.
> 3. A set of zero, or more, informational (BIP) messages associated with the result.
> 4. A set of (`key`, `value`) pairs that describes the submitted request in more detail. Check the documentation for information about how to parse these pairs.

To register a listener, each administered object has a registerListener() method that is used to tell the CMP to call the supplied code whenever an event occurs on that object. You can register the same `AdministeredObjectListener` for notifications from multiple administered objects. You can also register multiple `AdministeredObjectListeners` against the same administered object.

The following example demonstrates this by registering a listener on the topology object, and displaying a message whenever it is modified:

```
import com.ibm.broker.config.proxy.*;
import com.ibm.broker.config.common.CompletionCodeType;
import java.util.List;
import java.util.ListIterator;
import java.util.Properties;

public class MonitorTopology implements AdministeredObjectListener {

  public static void main(String[] args) {

    ConfigManagerProxy cmp = null;
    try {
      ConfigManagerConnectionParameters cmcp =
          new MQConfigManagerConnectionParameters(
            "localhost",
            1414,
            "");
      cmp = ConfigManagerProxy.getInstance(cmcp);
    } catch (ConfigManagerProxyException cmpex) {
      System.out.println("Error connecting: "+cmpex);
    }

    if (cmp != null) {
      System.out.println("Connected to Config Manager!");
      TopologyProxy topology = cmp.getTopology();
      listenForChanges(topology);
      cmp.disconnect();
    }
  }

  private static void listenForChanges(AdministeredObject obj)
  {
    try {
      if (obj != null) {
        obj.registerListener(new MonitorTopology());
        while(true) {
          // thread could do something else here instead
          try {
            Thread.sleep(10000);
```

```
              } catch (InterruptedException ex) {
                // ignore
              }
            }
          }
        }
      } catch(ConfigManagerProxyPropertyNotInitializedException
                                                    ex) {
          System.err.println("Comms problem! "+ex);
      }
    }

    public void processActionResponse(AdministeredObject obj,
                                      CompletionCodeType cc,
                                      List bipMessages,
                                      Properties refProperties) {
      // Event ignored in this example
    }

    public void processDelete(AdministeredObject deletedObject) {
     // Event ignored in this example
    }

    public void processModify(AdministeredObject affectedObject,
                              List changedAttributes,
                              List newChildren,
                              List removedChildren) {

      System.out.println(affectedObject+" has changed:");
      ListIterator e = changedAttributes.listIterator();
      while (e.hasNext()) {
        String changedAttribute = (String) e.next();
        System.out.println("Changed: "+changedAttribute);
      }
      ListIterator e2 = newChildren.listIterator();
      while (e2.hasNext()) {
        String newChildStr = (String) e2.next();
        AdministeredObject newChild =
          affectedObject.getManagedSubcomponentFromStringRepresentation(newChildStr);
        System.out.println("New child: "+newChild);
      }
      ListIterator e3 = removedChildren.listIterator();
      while (e3.hasNext()) {
        String remChildStr = (String) e3.next();
        AdministeredObject removedChild =
          affectedObject.getManagedSubcomponentFromStringRepresentation(remChildStr);
        System.out.println("Removed child: "+removedChild);
      }

    }
}
```

The listenForChanges() method attempts to register an instance of the
MonitorTopology class for notifications of topology changes. If successful, the main
thread pauses indefinitely to prevent the application from exiting once the method
returns. When the listener is registered, whenever the topology changes (for
example, if a broker is added) the processModify() method is called. This method
displays details of each notification on the screen.

You can stop receiving notifications in three ways:
- AdministeredObject.deregisterListener(AdministeredObjectListener)
- ConfigManagerProxy.deregisterListeners()
- ConfigManagerProxy.disconnect()

The first method de-registers a single listener from a single administered object; the other two methods deregister all listeners connected with that `ConfigManagerProxy` instance. In addition, the final method shows that all listeners are implicitly removed when connection to the Configuration Manager is stopped.

You can also implement the AdvancedAdministeredObjectListener interface which, when registered, yields additional information to applications.

Creating domain objects using the Configuration Manager Proxy

Create new objects within the broker domain.

The following example adds a broker called B2, that is running on queue manager QMB2, to the domain, and associates with it an execution group called `default`. Finally, this configuration is deployed to the broker.

This example can work successfully only if the broker B2 already exists on the computer that is running queue manager QMB2, and another Configuration Manager has not deployed to the broker previously.

```
import com.ibm.broker.config.proxy.*;

public class AddBroker {

  public static void main(String[] args) {

    ConfigManagerProxy cmp = null;
    try {
      ConfigManagerConnectionParameters cmcp =
        new MQConfigManagerConnectionParameters(
          "localhost",
          1414,
          "");
      cmp = ConfigManagerProxy.getInstance(cmcp);
    } catch (ConfigManagerProxyException cmpex) {
      System.out.println("Error connecting: "+cmpex);
    }

    if (cmp != null) {
      System.out.println("Connected to Config Manager!");
      addBroker(cmp, "B2", "QMB2", "default");
      cmp.disconnect();
    }
  }
  private static void addBroker(ConfigManagerProxy cmp,
                                String bName,
                                String bQMgr,
                                String egName)
  {
   TopologyProxy topology = null;
    try {
      topology = cmp.getTopology();
    } catch(ConfigManagerProxyPropertyNotInitializedException
                                                   ex) {
        System.err.println("Comms problem! "+ex);
    }

    if (topology != null) {
      try {
        BrokerProxy b2 = topology.createBroker(bName, bQMgr);
        ExecutionGroupProxy e = b2.createExecutionGroup(egName);
        b2.deploy();
      } catch (ConfigManagerProxyException ex) {
```

```
                System.err.println("Could not perform an action: "+ex);
        }
    }
  }
}
```

The critical statements in this example are the three lines inside the try block towards the end of the addBroker() method. The first statement adds the broker to the topology, the second creates the default execution group, and the third deploys the configuration (that is, the new execution group) to the broker.

The createBroker() method assumes that you have already created the physical broker component on the local computer by using the mqsicreatebroker command.

Because requests are processed asynchronously by the Configuration Manager, the BrokerProxy object that is returned from the createBroker() method is a skeleton object when returned to your application, because it refers to an object that might not yet exist in the Configuration Manager.

The ExecutionGroupProxy object e, returned from the createExecutionGroup() method, is in a similar state. In both cases, the application can manipulated the object as if it existed in the Configuration Manager, although the actual creation of the underlying object might not happen for some time.

When the object represented by the skeleton is created in the Configuration Manager, all requests that refer to it can be processed. In this example, when the broker has been added to the topology in the Configuration Manager, the Configuration Manager can honor the request to create the execution group.

If, for any reason, the request to create the object described by the skeleton fails, all requests that use the skeleton also fail. Therefore, if broker B2 cannot be created, all requests that concern the skeleton BrokerProxy object b2 (b2.createExecutionGroup() and b2.deploy()) also fail. However, the CMP API application works, as it does in the successful case, because no exception is thrown.

See "Checking the results of broker domain management in a CMP application" on page 28 for further information about how to detect problems such as these.

Advanced features of the Configuration Manager Proxy

This is part of the larger task of developing Configuration Manager Proxy (CMP) applications and introduces the advanced features of the CMP.

Follow the link for the advanced feature that you require:
- "The Configuration Manager Proxy subscriptions API"
- "Submitting batch requests from a CMP application" on page 38

The Configuration Manager Proxy subscriptions API

View and work with active subscriptions.

This task is part of the larger task of developing Configuration Manager Proxy (CMP) applications, and describes one of the advanced features of the CMP.

You can use the CMP to show and delete the set of active subscriptions in the
domain. The following example gives information on all subscriptions to topics
with names that begin with the string "shares".

```
import java.util.Enumeration;
import com.ibm.broker.config.proxy.*;

public class QuerySubscriptions {

  public static void main(String[] args) {

    ConfigManagerProxy cmp = null;
    BrokerProxy bp = null;
    TopologyProxy tp = null;
    try {
      ConfigManagerConnectionParameters cmcp =
          new MQConfigManagerConnectionParameters(
            "localhost",
            1414,
            "");
      cmp = ConfigManagerProxy.getInstance(cmcp);
      tp = cmp.getTopology();
      bp = tp.getBrokerByName("BROKER_A");
    } catch (ConfigManagerProxyException cmpex) {
      System.out.println("Error connecting: "+cmpex);
    }

    if (cmp != null) {
      System.out.println("Connected to Config Manager!");
      querySubscriptionsByTopic(cmp, "shares%");
      cmp.disconnect();
    }
  }

  private static void querySubscriptionsByTopic(
    ConfigManagerProxy cmp,
    String topic)
  {
    try {
      SubscriptionQuery sq = bp.createSubscriptionQuery();
      sq.setString(SubscriptionParameters.TOPIC, topic); // set the topic
      SubscriptionsProxy matchingSubscriptions = sq.executeQuery();


      Enumeration e = matchingSubscriptions.elements();
      int matches = matchingSubscriptions.getSize();
      System.out.println("Found "+matches+" matches:");

      while (e.hasMoreElements()) {
        Subscription thisSub = (Subscription)e.nextElement();
        System.out.println("-----");
        System.out.println("Broker="+thisSub.getBroker());
        System.out.println("Topic="+thisSub.getTopicName());
        System.out.println("Client="+thisSub.getClient());
        System.out.println("Filter="+thisSub.getFilter());
        System.out.println("Reg date="
          +thisSub.getRegistrationDate());
        System.out.println("User="+thisSub.getUser());
        System.out.println("Sub point="
          +thisSub.getSubscriptionPoint());
      }
    } catch (ConfigManagerProxyException e) {
      e.printStackTrace();
    }
  }
}
```

The class that queries the set of active subscriptions is SubscriptionsQuery(), which defines the query that is used to filter the subscriptions. The parameters that you define in **SubscriptionsParameters** and **SubscriptionsParameters.MQ** can be set on an instance of the class to build up a query. You can include the % character to denote wildcard characters in parameters of type string.

The parameters **SubscriptionsParameters.STARTDATE** and **SubscriptionsParameters.ENDDATE** are of type GregorianCalendar. Use these parameters to constrain the registration time of the matching subscriptions.

In the preceding example, only the **topic** parameter, of type string, is set is shares%. This setting tells the CMP to return all subscriptions whose topic name begins with "shares".

The query is issued when you call the SubscriptionQuery.executeQuery() method. It returns an instance of SubscriptionsProxy, which represents the results of the query. Because this class inherits from the class AdministeredObject, the attributes of this object are supplied asynchronously by the Configuration Manager. Therefore, the methods that interrogate the SubscriptionsProxy attributes can temporarily block while the CMP waits for the information to arrive.

The class Subscription, and its subclass MQSubscription which represents an individual match from the query, is a small data structure that is used for convenience by the SubscriptionsProxy, and does not block or throw exceptions.

Even though SubscriptionsProxy objects are of AdministeredObject type, *you cannot register AdministeredObjectListeners* against them. This characteristic means that when the results of a query are returned from the Configuration Manager, you are not notified if the set of matching subscriptions changes, unless you resubmit the query. The consequence of this behavior is that the results of subscriptions queries are guaranteed correct only at the time the original query was made.

You can delete subscriptions using the SubscriptionsProxy.deleteSubscriptions() method. Because SubscriptionsProxy objects cannot have AdministeredObjectListeners, the outcome of such an action is published to listeners of the ConfigManagerProxy object.

Submitting batch requests from a CMP application

Use the CMP API to group multiple requests that are destined for the same Configuration Manager, and submit them as a single unit of work.

To start a batch, your application must call the beginUpdates() method on the ConfigManagerProxy handle. The CMP API delays submitting any state-changing requests to the Configuration Manager until it is told a batch of requests is ready to be sent.

The sendUpdates() method tells the CMP API to submit as a batch all requests received since the last beginUpdates() call. The clearUpdates() method can be used to discard a batch without submitting it to the Configuration Manager. The application can check if a batch is currently in progress by using the isBatching() method. Only one batch for a CMP API handle can be in progress at any one time.

One advantage of using a batch method is that it provides an assurance that no other applications can have messages processed by the Configuration Manager during the batch. When a Configuration Manager receives a batch of requests, it

processes each request in the batch in the order it was added to the batch (FIFO), and requests from no other CMP application are processed until the entire batch is completed.

Consider the following sequence of commands:

```
BrokerProxy b2 = topology.createBroker("B2", "QMB2");
ExecutionGroupProxy e = b2.createExecutionGroup("default");
b2.deploy();
```

Without using a batch method, the application cannot guarantee the success of these actions. For example, even if each command would otherwise succeed, a second (possibly remote) application could delete the broker B2 after it has been created by the first application, but before the other two commands are processed.

If the sequence is extended to use a batch method, the Configuration Manager is now guaranteed to process all the commands together, therefore no other application can disrupt the logic intended by the application.

```
cmp.startUpdates();
BrokerProxy b2 = topology.createBroker("B2", "QMB2");
ExecutionGroupProxy e = b2.createExecutionGroup("default");
b2.deploy();
cmp.sendUpdates();
```

Another advantage of using a batch method is performance. The CMP typically sends one WebSphere MQ message to the Configuration Manager for each request.

In a situation that requires lots of requests to be sent in quick succession, the use of a batch has a significant effect on performance, reducing both time taken to process the requests, and the memory used. For example, your application might create a number of execution groups on a single broker. Each batch of requests is sent in a single WebSphere MQ message, therefore reducing the processing that is required for each method.

Batch mode does not provide transactional (commit and backout) capability; some requests in a batch might succeed and others fail. If the Configuration Manager processes a request in a batch that fails, it continues to process the next request in the batch until it has attempted all requests in the batch.

Part 2. Appendixes

Appendix. Notices for WebSphere Message Broker

Read the legal notices for WebSphere Message Broker.

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this 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:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 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 improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web

sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM United Kingdom Laboratories,
Mail Point 151,
Hursley Park,
Winchester,
Hampshire,
England
SO21 2JN

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Programming License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information includes examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to

IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

(C) (*your company name*) (*year*). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

Trademarks in the WebSphere Message Broker Information Center

Review the trademark information for WebSphere Message Broker.

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and trademark information at `www.ibm.com/legal/copytrade.shtml`.

Adobe is a registered trademark of Adobe Systems Incorporated in the United States, and/or other countries.

Intel, Itanium, and Pentium are trademarks of Intel Corporation in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

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

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

Index

C

D

T

IBM ®

Printed in USA