WebSphere® Business Integration Server Foundation for z/OS V5.1.1

Using the Common Event Infrastructure

SC23-4947-00

Note
Before using this information, be sure to read the general information under "Notices" on page 95.

Compilation date: September 20, 2004

Contents

About this document

This document includes information relating to technology preview code that is provided to you with the product. Such technology preview code is provided on an as-is basis with no warranty. IBM provides no support for this code. IBM does not warrant that: a) this code meets your requirements, and or b) your applications developed using this code are compatible with subsequent versions of the code. Some or all of the code might not be made generally available by IBM as a product. Production use of the technology preview code is not authorized.

How to send your comments

Your feedback is important in helping to provide the most accurate and highest quality information.

- To send comments on articles in the WebSphere Application Server Information Center
  1. Display the article in your Web browser and scroll to the end of the article.
  2. Click on the **Feedback** link at the bottom of the article, and a separate window containing an e-mail form appears.
  3. Fill out the e-mail form as instructed, and click on **Submit feedback**.
- To send comments on PDF books, you can e-mail your comments to: **wasdoc@us.ibm.com** or fax them to 919-254-0206.

  Be sure to include the document name and number, the WebSphere Application Server version you are using, and, if applicable, the specific page, table, or figure number on which you are commenting.

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

Chapter 1. Introduction to the Common Event Infrastructure

The Common Event Infrastructure is a shared component that can operate either inside or outside WebSphere Application Server. The Common Event Infrastructure provides facilities for the run-time environment to persistently store and retrieve events from many different programming environments. Events are represented using the Common Base Event model, a standard, XML-based format that defines the structure of an event.

The Common Event Infrastructure provides WebSphere Application Server with standard formats and mechanisms for managing event data. The following facilities are provided:

- Standard interfaces and services for WebSphere applications to create event objects, store them, send them, and retrieve them later.
- Facilities that pass event objects to registered applications either directly, in the context of the producing (source) application, or indirectly through Java Message Service (JMS). There are event emitters for Business Process Execution Language (BPEL)-based processes and for Enterprise JavaBeans (EJB) invocations based on deployment descriptor extensions.
- An event browser for browsing stored events.

About the Common Event Infrastructure in WebSphere Application Server

This topic gives an overview of the Common Event Infrastructure as it is implemented in WebSphere Application Server.

The Common Event Infrastructure provides facilities for generation, propagation, persistence, and consumption of events, but it does not define the actual events. Instead, application developers and administrators define event types, event groups, filtering criteria, and correlation criteria.

An *event* occurs when something significant happens in the IT system. For example, an application processing a new customer order or a failure occurring in a critical part of the system. Information about the event is captured in an *event object*. This event object describes an event type, indicates when the application generated the event, and identifies properties that are relevant to the event.

The Common Event Infrastructure in WebSphere Application Server has the following components:

Common base event
: The common base event component supports the creation of events and access to the property data of these events. Event sources use the common base event APIs to create new events that conform to the Common Base Event model. Event consumers use the APIs to read property data from received events. In addition, applications can convert events to and from XML text format, supporting interchange with other tools. The common base event component is part of the Eclipse-based Hyades environment.

Emitter
: The emitter component supports the sending of events. After an event source creates an event and populates it with data, the event source

submits the event to an emitter. The emitter optionally performs automatic content completion and then validates the event to ensure that it conforms to the Common Base Event specification. It also compares the event to configurable filter criteria. If the event is valid and passes the filter criteria, the emitter sends the event to the event server. An emitter can send events to the event server either synchronously (using Enterprise JavaBeans calls) or asynchronously (using a Java Messaging Service queue).

Event correlation spheres
An event correlation sphere is the scope that allows an event consumer to correlate events. Each event includes the identifier of the correlation sphere to which it belongs and the identifier of its parent correlation sphere from the event hierarchy. An emitter is provided (ECSEmitter class) that adds correlation data automatically to events.

Event server
The event server is the conduit between event sources and event consumers. The event server receives events that are submitted to emitters by event sources, stores events them in a persistent data store, and then distributes them asynchronously to subscribed event consumers. In addition, the event server supports synchronous queries of historical events from the persistent store.

Event catalog
The event catalog is a repository of event metadata. Applications use the event catalog to retrieve information about classes of events and the content of these events.

Event catalog application
Any application that stores or retrieves event metadata in the event catalog. This might be a management or development tool; it might also be an event source or event consumer.

Event source
Any application that uses an emitter to send events to the event server.

Event consumer
Any application that receives events from the event server. Event consumers process events outside the environment of the event source. Typically, these event consumers process events from a number of event sources.

Related concepts

"The Common Base Event model"

Related tasks

Chapter 5, "Working with events," on page 31

Chapter 6, "Developing an event source," on page 41

Chapter 7, "Developing an event consumer," on page 51

Chapter 8, "Developing an event catalog application," on page 67

The Common Base Event model

The Common Base Event model is a standard that defines a common representation of events. This standard, developed by the IBM Autonomic Computing Architecture Board, supports the encoding of logging, tracing, management, and business events using a common XML-based format. Using this format you can correlate different types of events that originate from different

applications. The Common Base Event model is part of the IBM Autonomic Computing Toolkit. For more information, see http://www.ibm.com/autonomic.

The Common Event Infrastructure currently supports version 1.0.1 of the specification.

The basic concept of the Common Base Event model is the *situation*. A situation can be anything that happens anywhere in the computing infrastructure, such as a server shutdown, a disk-drive failure, or a failed user login. The Common Base Event model defines a set of standard situation types that accommodate most of the situations that can occur, for example, the StartSituation and the CreateSituation situation types.

An *event* is a structured notification that reports information related to a situation. An event reports the following kinds of information:
- The situation (what has happened)
- The identity of the affected component, for example, the server that has shut down
- The identity of the component that is reporting the situation (which might be the same as the affected component)

The Common Base Event specification defines an event as an XML element that contains properties that provide all three kinds of information. These properties are encoded as attributes and subelements of the CommonBaseEvent root element.

The Common Base Event format is extensible. In addition to the standard event properties, an event can also contain extended data elements, which are application-specific elements with relevant information to the situation. The extensionName attribute labels an event with an optional classification name (an event class), which indicates to applications what sort of extended data elements to expect. The event catalog stores event definitions that describe these event classes and the allowed content of the event.

Related reference

"Common base event properties"

Common base event properties

The Common Base Event specification defines properties for common base events. A common base event has the following types of properties:

- Attributes for the CommonBaseEvent element. See "Attributes of the CommonBaseEvent element that are relevant to WebSphere events" on page 4.
- Identifier of the source component. For common base events that are generated within WebSphere Application Server, this section contains the attributes associated with the sourceComponentId element. These attributes describe the run-time environment that was running when the situation occurred. See "Identification of the event source" on page 4.
- Event context elements. See "Event context elements" on page 5.
- Event classification elements that describe the situation element. See "Situation elements" on page 6.

For more information on the Common Base Event specification, see the *Autonomic Computing Toolkit Developers' Guide* delivered with the IBM Autonomic Computing Toolkit, http://www.ibm.com/autonomic.

Attributes of the CommonBaseEvent element that are relevant to WebSphere events

The CommonBaseEvent element groups the data for a CommonBaseEvent instance. The attributes of the CommonBaseEvent element give basic information about a CommonBaseEvent instance. The element has a number of attributes that are common to all instances of common base events. These attributes are described in detail in the Common Base Event specification.

5.1.1+ For example, in a WebSphere Application Server environment if an event is sent using the ECSEmitter class, the following attributes are required.

Attribute	Description
creationTime	The local time on the WebSphere Application Server at which the event is created. This time is set automatically by the run-time environment.
extensionName	Identifies the structure and content of the event. You can use the event catalog to publish guidelines on how to read events with a certain extension name. For example, all process events sent by process choreographer have the extension name of `WPC:ProcessInstanceEvent`.
globalInstanceId	The globally unique identifier of the common base event instance. This ID is set automatically.
sequenceNumber	The number of common base events generated so far within a millisecond interval. This value is set to 1 but it can be overwritten.
severity	A number that describes the impact that the event has on the creator of the record. The value is set to 10 but it can be overwritten.
version	The version of the Common Base Event specification. This is set to 1.0.1.

The following example shows a typical use of these attributes:

```
<CommonBaseEvent creationTime="2004-06-11T16:22:55.060Z"
                 extensionName="WPC:ProcessInstanceEvent"
                 globalInstandId="CE98004140BBC311D8AC0CE749FC318A97"
                 sequenceNumber="1"
                 severity=10"
                 version="1.0.1">
  ...
</CommonBaseEvent>
```

For more information on the CommonBaseEvent element, see the Common Base Event specification in the *Autonomic Computing Toolkit Developers' Guide* delivered with the IBM Autonomic Computing Toolkit, http://www.ibm.com/autonomic.

Identification of the event source

This topic describes the event data that is automatically provided in common base events that occur in the WebSphere Application Server environment.

By default, common base event instances from WebSphere Application Server contain the sourceComponentId element. The component referred to by this element is the WebSphere Application Server where the situation occurred. The reporterComponentId is not set.

The following XML fragment shows the structure of the sourceComponentId element.

```
<sourceComponentId   component="WBI-SF#Platform 5.1 [BASE 5.1.1 a0421.06]
                              [JDK 1.4.1.1 jdk0409.03] [PME 5.1.0 a0405.03]"
                     componentIdType="ProductName"
                     executionEnvironment="Windows XP[x86]#5.1"
                     instanceId="Gemstone3\Gemstone3\server1"
                     location="Gemstone3.boeblingen.de.ibm.com"
                     locationType="Hostname"
                     processId="3260"
                     componentType="WebSphereApplicationServer"
                     subComponent="WPC"
                     threadId="ORB.thread.pool : 0"
                     componentType:="http://www.ibm.com/namespaces/autonomic/Workflow_Engine"/>
```

All of the attribute values are set automatically by the WebSphere run-time environment unless the caller provides the value explicitly. However, it is not recommended that you supply values explicitly. The attributes are defined as follows:

Attribute	Description
component	Identifies the products in the WebSphere stack. Set to `WBI-SF` followed by the version information of the WebSphere components.
componentIdType	Set to `ProductName`.
executionEnvironment	A string that describes the operating system in which the application server is running.
instanceId	The identifier of the application server. This ID has the format: <cell name>/<node name>/<server name>
location	The host name of the server (or server region for z/OS).
locationType	Set to `Hostname`.
processId	The identifier of the operating system process.
subComponent	The default is `J2EE_Application`.
threadId	The identifier of the current thread.
componentType	The identifier of the component that sent the event as specified in the Common Base Event specification.

Event context elements

If you use the ECSEmitter class to send events, two context elements are added automatically to the event information. These two context elements identify the current and the parent event correlation sphere on behalf of which the event was sent. The following example shows the context elements that are emitted on behalf of a Business Process Execution Language (BPEL) process instance event:

```
<contextDataElements name="ECSCurrentID"
                     type="ECSID"
   <contextValue>_PI:900300fd.385c662b.be7d67f6.c600047</contextValue>
</contextDataElements>
<contextDataElements name="ECSParentID"
                     type="ECSID"
   <contextValue></contextValue>
</contextDataElements>
```

The names of the context data elements are `ECSCurrentID` and `ECSParentID`. The type is a constant string set to `ECSID`. The value of the contextValue subelement is

the identification of the event correlation sphere that is passed to the constructor of the ECSEmitter class. The contextValue element can be empty.

Situation elements

The Common Base Event specification defines elements that give more information about an event. The most important of these elements is the situation element that gives a standardized classification of the situation in which the event occurred. This element is mandatory.

When events are sent through the deployment descriptor of an Enterprise JavaBeans module or through a Business Process Execution Language (BPEL) process, situation elements are added automatically to the event information. If an event is sent using the Java API and the situation is not set, an `OtherSituation` situation is created automatically.

Chapter 2. Planning to use the Common Event Infrastructure

The Common Event Infrastructure provides facilities for the generation, propagation, persistence, and consumption of events, but it does not define the actual events. When you plan how to use the event infrastructure in your system design, you need to understand the business concepts that are relevant, and map them to the appropriate components of your system design. You should provide the semantics of event management by defining event types and event groups, in the context of an architecture of event sources and event consumers.

1. Identify each *event source*. The event source is the application that creates the event. The event source passes the event object to the event infrastructure. The event infrastructure also stores the event object in a database for later retrieval. The role of the event infrastructure is to pass the event object onto any applications that express an interest in receiving it.
2. Identify each *event consumer*. An event consumer is an application that can use the information that is contained in the event object. Event consumers typically process events from a number of event sources.
3. 5.1.1 + Identify the hierarchy of the *event correlation spheres* and the identifiers for these spheres. Event consumers can use event correlation spheres to correlate events. The ECSEmitter class supports a hierarchy of correlation spheres by storing the current identifier and the parent identifier of the correlation spheres of an event in each event.

   For example, a Business Process Execution Language (BPEL) activity opens a correlation sphere for the current activity that identifies the activity with the activity instance ID. The parent correlation sphere is the correlation sphere of the process instance on behalf of which the activity is run. The parent correlation sphere is identified by the process instance ID.
4. Identify each *event group*. An event group defines the characteristics (property values) that all events of interest to a particular type of consumer can contain. Policies, such as access controls and distribution rules are assigned to the event groups to customize the behavior of the event infrastructure for each user group.

WebSphere supplies a default event group that is defined to include all events. This event group is called *Event groups list* and has a Java Naming and Directory Interface (JNDI) name of `com/ibm/events/configuration/event-groups/Default`

The following figure shows the relationship between these objects:

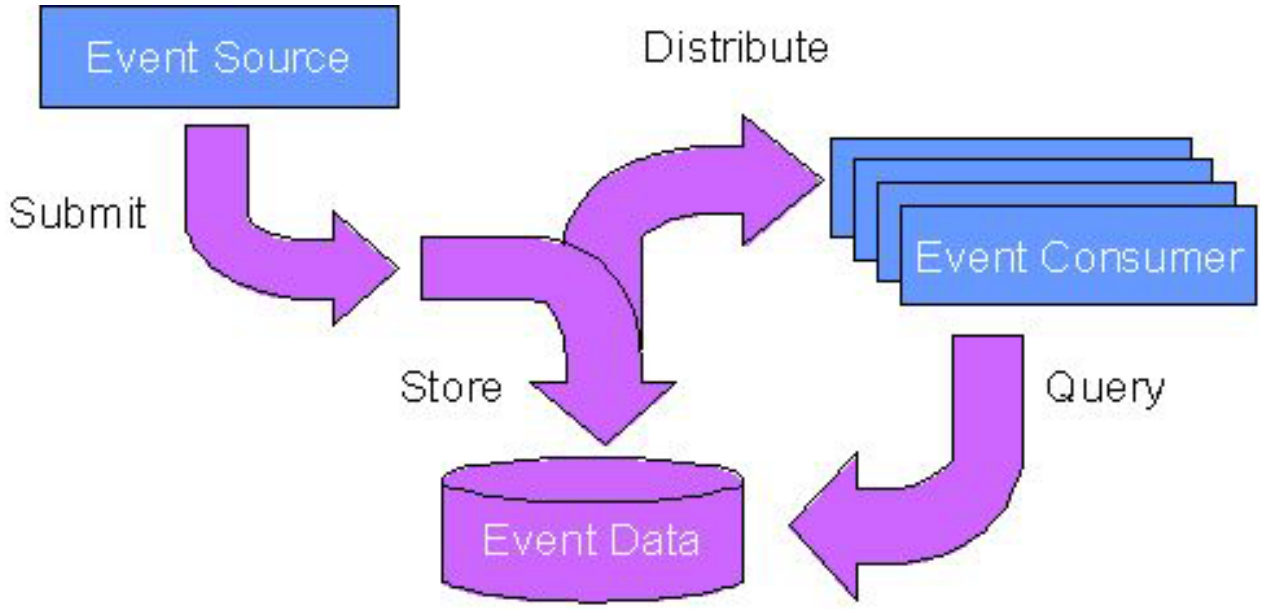

Figure 1. The architecture of an event source (which creates events), an event consumer (which makes use of the event data), and an event group (which defines the characteristics and associated policies for each type of event).

Chapter 3. Installing and configuring the Common Event Infrastructure

You must configure the necessary resources and services before you can use the Common Event Infrastructure.

1. Configure the event database. See "Configuring the event database" on page 11.
2. Deploy the Common Event Infrastructure application. See "Deploying the Common Event Infrastructure application" on page 19.
3. Start the application server.
4. Optional: Deploy a message driven bean. You can deploy a message driven bean in one of the following ways:
   - Use the embedded messaging service and associate the message queue to the default emitter profile. See "Configuring default event messaging" on page 20.
   - Use the WebSphere messaging service. See "Configuring event messaging using another JMS provider" on page 21.

   The Common Event Infrastructure is installed and ready to use. By default, the Common Event Infrastructure service and the application events service are started when the application server starts.
5. Optional: Change the default configuration settings for services and resources. These services and settings include:
   - Common Event Infrastructure service. See "Configuring the Common Event Infrastructure" on page 23.
   - Application events service. See "Configuring the application events service" on page 23.
   - Emitter factory profile. See "Creating an emitter factory profile" on page 24.
   - Event group. See "Creating an event group" on page 25.

Default configuration

The Common Event Infrastructure components are installed as a set of WebSphere Application Server applications, services, and default resources.

You can customize the Common Event Infrastructure by configuring the provided resources or creating additional resources.

The default configuration consists of the following objects:

Common Event Infrastructure service
: A service installed into the WebSphere server. This service enables WebSphere applications and clients to use the Common Event Infrastructure.

Common Event Infrastructure enterprise application
: The enterprise application for the event server. The deployment descriptor of the enterprise application associates the event server with the Common Event Infrastructure resources it uses.

Common Event Infrastructure messaging application
: The enterprise application for the message-driven bean that supports

asynchronous event transmission to the event server. This application is available only if you have configured event messaging.

Common Event Infrastructure Provider
A collection object that contains the resources used by Common Event Infrastructure components, event sources, and event consumers.

Data store profile
A data store profile defines the properties that are used by the default data store plug-in, which is used to persistently store events received by the event server. A default data store profile is provided. Usually, no configuration is necessary for this resource, but in some circumstances you might want to adjust some properties for your environment. You might also need to create additional data store profiles if you want to set up multiple event servers in the same cell.

Event bus transmission profile
An event bus transmission profile defines the properties that are used by emitters to access the event server synchronously using Enterprise JavaBeans (EJB) calls. These profiles are used by emitter factory profiles. A default transmission profile is provided. Usually, no configuration is necessary for this resource.

Event group profile list
An event group profile list is a collection that contains the event group profiles used by the event server. The event group profile list used by an event server is specified in the deployment descriptor of the event server enterprise application. Usually, no configuration is necessary for this resource, but you might need to create additional event group profile lists if you want to set up multiple event servers in the same cell.

Event group profile
An event group profile defines an event group (a logical collection of events). Event groups are used to categorize events according to their content. When querying events from the event server or subscribing to event distribution, an event consumer can specify an event group to retrieve only the events in that group.

A default event group profile is provided. This profile defines an event group that contains all of the events. This event group is associated with the Java Message Service (JMS) topic `jms/cei/notification/AllEventsTopic`. You can create additional event group profiles specifying whatever event criteria are appropriate for your application.

Emitter factory profile
An emitter factory profile defines the properties that are used by emitters. The properties in an emitter factory profile affect the behavior of any emitter that is created using the associated emitter factory. The default emitter factory profile specifies synchronous transmission, no filtering, and sending each event as part of the current transaction. You might want to create an additional emitter factory profile to specify a different transaction mode or transmission profile.

Event server profile
An event server profile defines the properties that are used by the event server. The default event server profile enables event distribution and persistence, and it is configured to use the default data store plug-in. Usually, no configuration is necessary for this resource, but you might

need to create additional event server profiles if you want to set up multiple event servers in the same cell.

JMS Transmission Profile
A JMS transmission profile defines the properties that are used by emitters to access the event server asynchronously using a JMS queue. This profile is referenced by emitter factory profiles. This profile is available only if you have configured event messaging.

Related tasks

"Configuring the Common Event Infrastructure" on page 23

"Configuring the application events service" on page 23
Complete this task to switch the application events service on or off.

"Creating an emitter factory profile" on page 24

"Creating an event group" on page 25

Configuring the event database

Before you can use the Common Event Infrastructure, you must configure the event database.
On the machine hosting your database, create the event database according to the description for your database system. You can create an event database for the following database systems:

- "Configuring a Cloudscape database" on page 12
- "Configuring a DB2 database on a z/OS system" on page 12

The database exists and it is ready to use.

Database configuration logs and messages

The scripts for configuring and removing the event database create two log files:

- The *install_root*/logs/event/event_db.log log file contains detailed trace information.
- The *install_root*/logs/event/event_db_msg.log log file contains any messages generated by the database configuration script.

Log file messages are in the following format:

```
<Date> <month><year> <time><Class> <Methods><Type> <Message>
```

The fields in the message statements are as follows:

Class The name of the class generating the message.

Method
The method generating the log message.

Type The type of message. This can be any of the following:
- Entry
- Exit
- Error
- Information
- Warning

Message
The text of the message.

Configuring a Cloudscape database

Follow these steps to configure a Cloudscape event database.

1. Create a database response file. A database response file is a text file that specifies parameters for configuring the event database. These parameters vary depending on the type of database being used.
2. Run the database configuration scripts. The Common Event Infrastructure provides scripts for configuring or upgrading the event database. These scripts in turn generate customized, database-specific scripts for creating or modifying the necessary database configuration using the parameters in your response file.

Related tasks

"Creating a database response file" on page 13
A database response file is a text file specifying parameters for configuring the event database. These parameters vary depending on the type of database being used.

"Running database configuration scripts on a z/OS system" on page 18
The Common Event Infrastructure provides scripts for configuring or upgrading the event database. These scripts generate customized, database-specific scripts for creating or modifying the database configuration using the parameters that you specify in your response file.

Configuring a DB2 database on a z/OS system

Follow these steps to configure a DB2 event database on a z/OS system.

1. On the z/OS system, use the DB2 administration menu to create a new subsystem.
2. Create a storage group. You also need to specify the storage group name in the database response file; the default value is `sysdeflt`.
3. Grant the necessary permissions to the user ID you want the WebSphere Application Server data source to use. This user ID must have rights to access the database and the storage group that you created. The user ID must also have permission to create new tables, table spaces, and indexes for the database.
4. Create a database response file. A database response file is a text file that specifies parameters for configuring the event database. These parameters vary depending on the type of database being used.
5. Run the database configuration scripts. The Common Event Infrastructure provides scripts for configuring the event database. These scripts generate customized, database-specific scripts for creating the database configuration using the parameters in your response file.

Related tasks

"Creating a database response file" on page 13
A database response file is a text file specifying parameters for configuring the event database. These parameters vary depending on the type of database being used.

"Running database configuration scripts on a z/OS system" on page 18
The Common Event Infrastructure provides scripts for configuring or upgrading the event database. These scripts generate customized, database-specific scripts for creating or modifying the database configuration using the parameters that you specify in your response file.

Creating a database response file

A database response file is a text file specifying parameters for configuring the event database. These parameters vary depending on the type of database being used.

If you are upgrading an existing Cloudscape event database, you must use the same response file you used when you originally configured the database. A backup copy of this response file is created during the Common Event Infrastructure installation and saved as *install_root*/event/dbconfig/CloudscapeResponseFile.bak.

To create a database response file, follow these steps:

1. Using an ASCII text editor, open one of the sample database response files. These files are located in the *install_root*/event/dbconfig directory. Select the sample response file for the database software you are using:

Database	Sample response file
Cloudscape	CloudscapeResponseFile.txt
DB2 Universal Database for z/OS	DB2ZOSResponseFile.txt

2. Modify the parameters in the response file as appropriate for your database configuration.
3. Save the file to your Common Event Infrastructure installation directory. You can give the modified response file any name you want to use; you specify this file when you run the database configuration script.

Cloudscape database response file

A Cloudscape database response file specifies parameters for configuring a Cloudscape event database.

A sample Cloudscape database response file called CloudscapeResponseFile.txt is available in the *install_root*/event/dbconfig directory.

This response file specifies the following parameters:

SHARE_DB=[server| node| cell]
The scope in which the configured database is shared. This is the scope in which Java database connectivity (JDBC) data sources is created. This parameter is optional; the default value is **server**.

WAS_SERVER= *server*
The name of the WebSphere Application Server where the database is installed. This parameter is applicable only if the **SHARE_DB** parameter is set to **server**. If you do not specify a server name, the default value is server1.

DB_NAME= *name*
The name of the event database. This parameter is optional. The default value is event.

JDBC_PROVIDER= *provider*
The name of the JDBC provider to configure. The value must be the name of a JDBC driver supported by WebSphere Application Server Version 5.1, and later. The Cloudscape JDBC Provider (XA) driver is recommended.

DB_TYPE= CLOUDSCAPE
The type of database to be configured. For a Cloudscape database, this must be **CLOUDSCAPE**.

PAGE_CACHE_SIZE= *size*
The number of memory pages to use for caching data. Increasing the page cache size can improve performance, but also requires more memory. See the Cloudscape documentation for more information about caching. This parameter is optional. The default value is `4000`.

LOG_DEVICE= *path*
The path to the location where the transaction logs are written. Using a separate device for logs can improve performance, but it also complicates backup and recovery. This parameter is optional.

DB2 Universal Database response file for z/OS systems

A DB2 Universal Database response file specifies parameters for configuring a DB2 event database on a z/OS system.

A sample DB2 response file for z/OS systems, called `DB2ZOSResponseFile.txt`, is available in the *`install_root`*`/event/dbconfig` directory. This response file specifies the following parameters:

WAS_SERVER= *server*
The name of the WebSphere Application Server where the database is installed. This parameter is applicable only if the **SHARE_DB** parameter is set to **server**. If you do not specify a server name, the default value is `server1`.

SHARE_DB=[server| node| cell]
The scope in which the configured database is shared. This is the scope in which Java database connectivity (JDBC) data sources are created. This parameter is optional. The default value is **server**.

DB_NAME= *name*
The name of the event database. This name must be no longer than 8 characters and must be the name of an existing database. This parameter is optional. The default value is `ceizos`.

JDBC_PROVIDER= *provider*
The name of the JDBC provider to configure. The value must be the name of a JDBC driver supported by WebSphere Application Server Version 5.1, and later. The following drivers are recommended:
- DB2 Universal JDBC Driver Provider (XA)
- DB2 Legacy CLI-based Type 2 JDBC Provider (XA)

JDBC_CLASSPATH= *path*
The path to the JDBC driver (not including file name). This should be one of the following:
- For DB2 Universal JDBC Driver Provider (XA): the path to the `db2jcc_license_cu.jar` and `db2jcc_license_cisuz.jar` files.
- For DB2 Legacy CLI-based Type 2 JDBC Driver (XA), the path to the `db2java.zip` file.

UNIVERSAL_JDBC_CLASSPATH= *path*
For DB2 Universal JDBC Driver Provider or DB2 Universal JDBC Driver Provider (XA), the path to the JDBC driver (not including file name). This should be the path to the `db2jcc_license_cu.jar` file. This parameter is optional.

JDBC_DRIVER_TYPE= *type*
The JDBC driver type. This should be either `2` or `4`.

DB_HOST_NAME= *hostname*
The database server host name. This parameter is required if **JDBC_DRIVER_TYPE** is set to 4. The default value is localhost.

DB_INSTANCE_PORT= *port*
The database instance port number. This parameter is required if **JDBC_DRIVER_TYPE** is set to 4. The default port number is 5027.

EXECUTE_SCRIPTS=[YES| NO]
Specifies whether the database configuration scripts are automatically run. If you are configuring the database on a z/OS system with UNIX System Services, set this value to NO.

DB_TYPE= DB2ZOS
The type of database to configure. For a DB2 for z/OS database, this must be **DB2ZOS**.

EVENT_DB_NAME= *name*
The database name for the event database. This name must be no longer than 8 characters. The default value is event.

CATALOG_DB_NAME= *name*
The database name for the event catalog database. This name must be no longer than 8 characters. The default value is eventcat.

STORAGE_GROUP= *group*
The storage group for the event database and catalog database. This must be the name of an existing storage group. The default value is sysdeflt.

BUFFER_POOL_4K= *name*
The name of the 4K buffer pool. The default value is BP9.

BUFFER_POOL_8K= *name*
The name of the 8K buffer pool. The default value is BP8K9.

BUFFER_POOL_16K= *name*
The name of the 16K buffer pool. The default value is BP16K9.

DAYS_TO_KEEP_EVENTS= *days*
The number of days that events are kept in the database before they are purged. Changes to this value significantly affect the amount of storage allocated for the table spaces that store event data. The default value is 1.

AVERAGE_EVENTS_PER_SECOND= *events*
The average number of events that are stored in the database each second. Changes to this value significantly affect the amount of storage allocated for the table spaces that store event data. The default value is 1.

AVERAGE_NUMBER_CONTEXT_PER_EVENT= *number*
The average number of context elements per event instance. The default value is 1.

AVERAGE_NUMBER_EXTENDED_DATA_ELEMENT_PER_EVENT= *number*
The average number of extended data elements per event instance. Changes to this value significantly affect the amount of storage allocated for the table spaces that store extended data element data. The default value is 5.

AVERAGE_NUMBER_EXTENDED_DATA_ELEMENT_ARRAY_ELEMENTS= *number*
The average number of values for extended data elements that are array data types. The default value is 5.

AVERAGE_NUMBER_MSG_TOKENS_PER_EVENT= *number*
The average number of message tokens per event. The default value is 1.

AVERAGE_ASSOCIATIONS_PER_EVENT= *number*
The average number of event associations per event. The default value is 2.

TABLESPACE_EXTENDED_BINARY_VALUE_PRIMARY= *size*
The primary allocation for the large object (LOB) table space that contains hexBinary extended data element values. This allocation can be small if events do not typically contain hexBinary extended data element values. The default value is 1000.

TABLESPACE_EXTENDED_BINARY_VALUE_SECONDARY= *size*
The secondary allocation for the large object (LOB) table space that contains hexBinary extended data element values. This allocation can be small if events do not typically contain hexBinary extended data element values. The default value is 200.

TABLESPACE_ANY_VALUE_PRIMARY= *size*
The primary allocation for the large object (LOB) table space that contains the values for the *any* element, which is a character large object (CLOB). This allocation can be small if events do not typically contain *any* elements. The default value is 1000.

TABLESPACE_ANY_VALUE_SECONDARY= *size*
The secondary allocation for the large object (LOB) table space that contains the values for the *any* element, which is a character large object (CLOB). This allocation can be small if events do not typically contain *any* elements. The default value is 200.

PERCENTAGE_FREE_SPACE= *percent*
The amount of free space, as a percentage, to leave on each page. Increase this value as the number of inserted rows increases. Free space makes updates more efficient, but a larger value uses more disk space. The default value is 20.

FREE_PAGE= *pages*
The number of pages to fill before leaving a free page. If this parameter is set to 0, free pages are not left. Set this parameter to a nonzero value if a large amount of SQL INSERT processing is expected. (A nonzero value uses more disk space.) The default value is 10.

NUMBER_EVENT_DEFINITIONS= *definitions*
The number of event definitions stored in the event catalog. The default value is 100.

AVERAGE_SOURCE_CATEGORY_PER_EVENT_DEFINITION= *categories*
The average number of source categories per event definition in the event catalog. The default value is 1.

AVERAGE_EXTENDED_DATA_ELEMENT_PER_EVENT_DEFINITION= *definitions*
The average number of extended data element descriptions for each event definition in the event catalog. The default value is 5.

AVERAGE_PROPERTY_DESCRIPTIONS_PER_EVENT_DEFINITION= *definitions*
The average number of property descriptions for each event definition in the event catalog. The default value is 5.

TABLESPACE_HEX_DEFAULT_PRIMARY= *size*
The primary allocation for the large object (LOB) table space that contains the default values for hexBinary extended data elements. The default value is 100.

TABLESPACE_HEX_DEFAULT_SECONDARY= *size*
The secondary allocation for the large object (LOB) table space that contains the default values for hexBinary extended data elements. The default value is `10`.

Upgrading a Cloudscape event database

The Common Event Infrastructure provides a script for upgrading a Cloudscape event database. This script generates customized, database-specific scripts for creating or modifying the database configuration using the parameters that you specify in your response file.

To generate the event database configuration scripts, use the **upgrade_event_database** script. This script is located in the *install_root*`/event/dbconfig` directory. After the command, specify the name of the database response file.

To upgrade an existing Cloudscape event database, run the following command:

```
upgrade_event_database.sh response_file
```

The *response_file* parameter specifies the name of the database response file.

Note: The **upgrade_event_database** script is supported only if you are upgrading a Cloudscape database. For any other database type, you must configure a new event database.

The **upgrade_event_database** command generates the appropriate scripts for configuring the event database and the Java database connectivity (JDBC) provider, based on the parameters in the response file. If the **EXECUTE_SCRIPTS** parameter in your response file is set to **true**, the command also runs the scripts automatically, completing the database configuration. If **EXECUTE_SCRIPTS** is set to **false**, the scripts must be run separately after they are generated. These scripts configure the event database and create two JDBC data sources: one for the event database and one for the event catalog.

The generated scripts for creating the event database are placed in database-specific subdirectories of the *install_root*`/event/dbscripts` directory. The scripts for configuring the JDBC provider are placed in database-specific subdirectories of the *install_root*`/event/dsscripts` directory. The names of the generated scripts depend on the database type and operating system:

Type	Database script	JDBC configuration script
Cloudscape	`cr_event_cloudscape.sh`	`cr_cloudscape_jdbc_provider.sh`

You can create the event database or configure the JDBC provider at any time by running the appropriate script. To configure the JDBC provider, use the appropriate script and specify the scope in which the JDBC provider is to be configured:

```
cr_db_jdbc_provider scope [server_name]
```

The generated scripts use these parameters:

scope The scope in which you want to configure the JDBC provider. The valid values are `cell`, `node`, and `server`.

server_name
The name of the WebSphere Application Server where you want to configure the JDBC provider, if **scope** is `server`. (If **scope** is `cell` or `node`, this parameter is ignored.)

After the event database is configured, you must restart the application server.

Running database configuration scripts on a z/OS system

The Common Event Infrastructure provides scripts for configuring or upgrading the event database. These scripts generate customized, database-specific scripts for creating or modifying the database configuration using the parameters that you specify in your response file.

Configuring the event database on a z/OS system using the UNIX System Services subsystem

To generate and run the event database configuration scripts on the native z/OS system using the UNIX System Services subsystem, follow these steps:

1. Run the **config_event_database** script. This script is located in the *install_root*/event/dbconfig directory. Specify the name of the database response file with the command:

   install_root/event/dbconfig/config_event_database.sh *response_file*

   The *response_file* parameter specifies the name of the database response file.

   The **config_event_database** command generates the appropriate scripts for configuring the event database and Java database connectivity (JDBC) provider, based on the parameters that you specify in the response file.

   The generated scripts for creating the event database are placed in the *install_root*/event/dbscripts/db2zos directory. The script for configuring the JDBC provider is placed in the *install_root*/event/dsscripts directory.
2. Use the SQL Processor Using File Input (SPUFI) to load and run the generated DDL scripts in the following order:
   - *install_root*/event/dbscripts/db2zos/ddl/cr_db.db2
   - *install_root*/event/dbscripts/db2zos/ddl/cr_db_catalog.db2
   - *install_root*/event/dbscripts/db2zos/ddl/cr_tbl.db2
   - *install_root*/event/dbscripts/db2zos/ddl/cr_tbl_catalog.db2
   - *install_root*/event/dbscripts/db2zos/ddl/ins_metadata.db2
   - *install_root*/event/dbscripts/db2zos/ddl/catalogSeed.db2

   The event database is now created.
3. Run the *install_root*/event/dsscripts/cr_db2zos_jdbc_provider script to create the event data source. Specify the scope at which the JDBC provider is to be configured:

   cr_db2zos_jdbc_provider *scope* [*server_name*]

Configuring the event database on a z/OS system from a remote client

To generate and run the event database configuration scripts on a z/OS system from a remote client, follow these steps:

1. Make sure you have the DB2 Connect product installed with the latest fix packs.
2. Catalog the remote database using the following commands, either in a script or in a DB2 command-line window:

```
catalog tcpip node zosnode remote hostname server IP_port system db_subsystem
catalog database db_name as db_name at node zosnode authentication DCS
```

For more information about how to catalog nodes and databases, refer to the DB2 Connect documentation.

3. Verify that you can establish a connection to the remote subsystem by entering the following command:

   ```
   db2 connect to subsystem user userid using password
   ```

4. Bind to the host database with the following commands:

   ```
   db2 connect to db_name user userid using password
   db2 bind path/bnd/@ddcsmvs.lst blocking all sqlerror continue message
        mvs.msg grant public
   db2 connect reset
   ```

   For more information about how to bind a client to a host database, refer to the DB2 Connect documentation.

5. Run the following command to generate the scripts for creating the event database and data source:

   ```
   install_root/event/dbconfig/config_event_database response_file
   ```

   If EXECUTE_SCRIPTS in the database response file is set to `yes`, the scripts are also automatically run.

6. If EXECUTE_SCRIPTS in the database response file is set to `no`, upload and run the generated scripts using SPUFI, as described in the previous section.

Deploying the Common Event Infrastructure application

After installation, you must deploy the event server enterprise application in the WebSphere Application Server.

The event server enterprise application is packaged in the `event-application.ear` EAR file. The `event-application.jacl` script installs this application in the WebSphere Application Server.
To deploy the application, use the wsadmin tool to run the `event-application.jacl` script:

To run the script on a z/OS system, change to the *install_root*`/event/application` directory and run the following command (all on one line):

```
wsadmin.sh -f event-application.jacl -profile event-profile.jacl
  -wsadmin_classpath install_root/event/lib/cei_installer.jar -action action
  -earfile event-application.ear -backendid backend_id
  -node node_name -server server_name
  [-appname app_name] [-trace]
```

The parameters of the `event-application.jacl` script are as follows:

action
: The action to perform. To install the enterprise application, specify `install`. To update an existing event server application that is already installed, specify `update`. During an update, the script makes a backup copy of the existing application EAR file in the current directory. If necessary, you can later use this backup copy to restore the previous version of the application.

backend_id
: The type of database backend to be used by the enterprise application. This must be one of the following values:

- CLOUDSCAPE_V51_1
- DB2UDBOS390_V7_1

node_name
The WebSphere Application Server node in which the event server is installed.

To find out the node name, run the *install_root*/bin/setupCmdLine and then the echo $WAS_NODE command.

server_name
The WebSphere Application Server into which the event server enterprise application is to be deployed.

app_name
The name to use for the Common Event Infrastructure enterprise application. This parameter is optional. The default value is CommonEventInfrastructureServer.

The optional **-trace** parameter causes additional debugging information to display on the standard output.

After the event-application.jacl script completes, the Common Event Infrastructure enterprise application is deployed in the specified node. In a WebSphere Application Server Network Deployment environment, if the application is already installed, the script adds only the deployment information for the specified node and server.

Configuring default event messaging

The create-default-event-message.jacl script provides a way to quickly set up a default messaging configuration that uses the WebSphere embedded messaging feature as the JMS provider. This script sets up all of the configuration objects required for asynchronous event transmission:

- It creates a JMS queue and a queue connection factory using the embedded messaging feature.
- It creates a message listener port associating the JMS queue and queue connection factory.
- It creates a JMS transmission profile using the created queue and connection factory.
- It configures the default emitter factory profile to use the created JMS transmission profile for asynchronous event transmission.
- It deploys the message-driven bean used by the Common Event Infrastructure to receive events sent asynchronously to the event server.

For more information about JMS transmission profiles and emitter factory profiles, see Chapter 6, “Developing an event source,” on page 41.

If you want to use a different JMS provider, use the event-message.jacl script, see “Configuring event messaging using another JMS provider” on page 21.

1. To run the create-default-event-message.jacl script, use the wsadmin tool:

   To run the script, change to the *install_root*/event/application directory and run the following command (all on one line):

   ```
   wsadmin.sh -f create-default-event-message.jacl -profile event-profile.jacl
     -wsadmin_classpath install_root/event/lib/cei_installer.jar
     -earfile event-message.ear -node node_name -server server_name
     [-appname app_name] [-trace]
   ```

The parameters of the `create-default-event-message.jacl` script are as follows:

node_name
The WebSphere Application Server node where you want to install the event messaging enterprise application.

To find out the node name, run the *install_root*`/bin/setupCmdLine` and then the `echo $WAS_NODE` command.

server_name
The WebSphere server into which the event server enterprise application is to be deployed.

app_name
The name to use for the messaging enterprise application. This parameter is optional; the default value is `CommonEventInfrastructureMessageApp`.

The optional **-trace** parameter causes additional debugging information to be displayed on the standard output.

After you start the script, you are prompted for your JMS user ID and password.

2. Stop and restart the application server.

The configuration is successful if the `SystemOut.log` file shows that the CommonEventInfrastructureMessageApp application is up and running.

```
ApplicationMg A WSVR0200I: Starting application: CommonEventInfrastructureMessageApp
EJBContainerI I WSVR0207I: Preparing to start EJB jar: EventServerMdb.jar
EJBContainerI I WSVR0037I: Starting EJB jar: EventServerMdb.jar
MDBListenerIm I WMSG0042I: MDB Listener CommonEventInfrastructureMessageApp-ListenerPort
                           started successfully for JMSDestination jms/cei/messageq
ApplicationMg A WSVR0221I: Application started: CommonEventInfrastructureMessageApp
```

Configuring event messaging using another JMS provider

You can use the *event-message.jacl* script to configure event messaging using any Java Messaging Service (JMS) provider. You must create the necessary JMS resources separately.

Before you can configure event messaging using the `event-message.jacl` script, you must first create a JMS queue and connection factory, using the appropriate interfaces for your JMS provider.

This script sets up the configuration objects required for asynchronous event transmission using a JMS provider that you have configured separately:

- It creates a JMS transmission profile using the JMS queue and connection factory you specify.
- It creates a message listener port associating the JMS queue and queue connection factory.
- It creates an emitter factory profile using the created JMS transmission profile for asynchronous event transmission.
- It deploys the message-driven bean used by the Common Event Infrastructure to receive events sent asynchronously to the event server.

For more information about JMS transmission profiles and emitter factory profiles, see Chapter 6, “Developing an event source,” on page 41.

1. Create a JMS connection factory.

Set the JNDI name of the connection factory to jms/cei/messageqcf.

2. Create a JMS queue.

   Set the JNDI name of the queue to jms/cei/messageq.

3. Copy the cei_installer.jar file from the lib directory to the classes directory.

   Copy the file from the *install_root*/event/lib directory to the *install_root*/event/classes directory.

4. Change to the classes directory.

5. Use the wsadmin tool to run the event-message.jacl script.

   To run the script, change to the *install_root*/event/application directory and run the following command (all on one line):

   ```
   install_root/bin/wsadmin.sh -f event-message.jacl -profile event-profile.jacl
     -action install -earfile event-message.ear -node node_name
     -server server_name
     -qjndi queue -qcfjndi connection_factory
     [-eventprofilescope scope] -appname app_name [-trace]
   ```

   The parameters of the event-message.jacl script are as follows:

   node_name
   The WebSphere Application Server node where you want to install the event messaging enterprise application.

   To find out the node name, run the *install_root*/bin/setupCmdLine and then the echo $WAS_NODE command.

   server_name
   The WebSphere Application Server into which the event messaging enterprise application is to be deployed.

   queue
   The JNDI name of the JMS queue to be used by the messaging enterprise application. This queue is used for asynchronous message transport to the event server.

   This parameter must be set to jms/cei/messageq.

   connection_factory
   The JNDI name of the JMS connection factory to be used by the messaging enterprise application.

   This parameter must be set to jms/cei/messageqcf.

   scope
   The scope of the configuration profile objects to be created for event messaging. This parameter is optional. If you specify a scope, a JMS transmission profile and emitter factory profile are created at the specified scope. The valid values are cell, node, and server.

   app_name
   The name to use for the messaging enterprise application. This parameter must be set to CommonEventInfrastructureMessageApp.

   The optional **-trace** parameter causes additional debugging information to be displayed on the standard output.

   After you start the script, you are prompted for your JMS user ID and password.

6. Stop and restart the application server.

The configuration is successful if the SystemOut.log file shows that the CommonEventInfrastructureMessageApp application is up and running.

```
ApplicationMg A WSVR0200I: Starting application: CommonEventInfrastructureMessageApp
EJBContainerI I WSVR0207I: Preparing to start EJB jar: EventServerMdb.jar
EJBContainerI I WSVR0037I: Starting EJB jar: EventServerMdb.jar
MDBListenerIm I WMSG0042I: MDB Listener CommonEventInfrastructureMessageApp-ListenerPort
                           started successfully for JMSDestination jms/cei/messageq
ApplicationMg A WSVR0221I: Application started: CommonEventInfrastructureMessageApp
```

Configuring the Common Event Infrastructure

Use the Common Event Infrastructure service to process events in WebSphere applications and processes.

You can use the Common Event Infrastructure service **Startup** property to specify whether the service is started automatically for an application server.

To configure the Common Event Infrastructure service **Startup** property for an application server, use the administrative console to complete the following steps:

1. Start the administrative console.
2. In the navigation pane, click **Servers > Application Servers**. A list of the application servers is displayed in the content pane.
3. In the Content pane, select the application server that you want to configure. The properties for the application server are displayed in the content pane.
4. In the Additional Properties table, select **Common Event Infrastructure service**. The Common Event Infrastructure properties are displayed in the content pane.
5. Select or clear the **Startup** property as needed:
   Selected
   [Default] The Common Event Infrastructure service starts when the application server starts. This enables applications that generate events to run on such an application server.
   Cleared
   The Common Event Infrastructure service does not start when the application server starts. Applications that generate events cannot start on such an application server.

   Any attempt to start an application that uses Common Event Infrastructure is rejected and a message is issued. The server continues to start without the application.
6. Click **OK**.
7. To save your configuration, click **Save** on the task bar of the administrative console.
8. Stop and then restart the application server for the changes to take effect.

Configuring the application events service

Complete this task to switch the application events service on or off.

The application events service provides access to the common event infrastructure for WebSphere applications. This service ensures that information about the WebSphere server is automatically included in each event passed to the Common Event Infrastructure.

You can use the Events Service **Startup** property to specify whether the service is started automatically for an application server.

To configure the Events Service **Startup** property for an application server, use the administrative console to complete the following steps:

1. Start the administrative console.
2. In the navigation pane, click **Servers > Application Servers**. A list of the application servers is displayed in the content pane.
3. In the Content pane, select the application server that you want to configure. The properties for the application server are displayed in the content pane.
4. In the Additional Properties table, select **Application Events Service**. The events service properties are displayed in the content pane.
5. Select or clear the **Startup** property as needed:
   Selected
   [Default] The application events service starts when the application server starts. This enables applications that specify use of the Common Event Infrastructure in the deployment descriptors to run on such an application server.
   Cleared
   The application events service does not started when the application server starts. Applications that specify use of the Common Event Infrastructure in their deployment descriptors cannot start on such an application server.

   Any attempt to start an application that uses events is rejected and a message is issued. The server continues to start without the application.
6. Click **OK**.
7. Review the Java Naming and Directory Interface (JNDI) name of the event emitter profile factory that is used to submit events to the Common Event Infrastructure. The name that is provided is part of the WebSphere default profile. Unless you have generated an alternative profile, accept the default name.
8. To save your configuration, click **Save** on the task bar of the administrative console.
9. Stop and then restart the application server for the changes to take effect.

Creating an emitter factory profile

An emitter factory profile defines properties that are used for an emitter factory, which event sources use to create emitters. The properties in an emitter factory profile affect the behavior of any emitter that is created using the associated emitter factory. You can use the default emitter factory profile or create additional profiles for your event sources to use. You might want to create an additional emitter factory profile to specify a different transaction mode or synchronous transmission profile. For more information about how these options affect the behavior of the emitter, see Chapter 6, “Developing an event source,” on page 41.

To create an emitter factory profile, follow these steps:

1. In the WebSphere administrative console, click **Resources > Common Event Infrastructure Provider > Emitter Factory Profile > New**.
2. Specify the properties of the new profile. Refer to the online help for the Emitter Factory Profile Settings page for detailed information about these properties.

3. Click **OK** to save your changes and create the emitter factory profile.

Event sources can now use the configured emitter factory to obtain emitters.

Creating an event group

An event group defines a logical collection of events based on the content of their property data. You can use an event group to query events from the event server. You can also associate an event group with a Java Message Service (JMS) destination for asynchronous event distribution.

To create an event group, follow these steps:

1. Optional: Set up one or more JMS destinations for the event group. An event group can be associated with one JMS topic, and one or more JMS queues. Refer to the documentation for your JMS provider for information on how to create JMS destinations and connection factories and bind them into a Java Naming and Directory Interface (JNDI) namespace.
2. Create a new event group profile. In the WebSphere administrative console, click **Resources > Common Event Infrastructure Provider > Event Group Profile List >** ***event_group_profile_list*** **> Event Group Profiles > New**.
3. Specify the properties of the event group profile. Refer to the online help for the Event Group Profile Settings page for detailed information about these properties.
4. Click **OK** to save your changes and create the event group profile.

Event consumers can now specify the event group when querying events. If event distribution is enabled in the event server profile, events belonging to the event group are also published to JMS destinations that are specified in the event group profile. Event consumers can then receive events asynchronously by subscribing to the appropriate destinations.

Chapter 4. Administering the Common Event Infrastructure

You can perform the following administrative tasks to control the operation of the Common Event Infrastructure components at run time.

- "Logging and tracing in the WebSphere environment"
- "Removing the Common Event Infrastructure configuration"

Logging and tracing in the WebSphere environment

For components that run within the WebSphere Application Server environment, you can enable logging and tracing using the WebSphere administrative console.

To enable logging and tracing in the WebSphere environment, follow these steps:

1. In the WebSphere administrative console, click **Troubleshooting > Logs and Trace**.
2. In the list of servers, click the server where the Common Event Infrastructure is installed.
3. Click **Diagnostic Trace**.
4. In the **Trace Specification** field, click **Modify**.
5. In the list of groups, click **CommonEventInfrastructure**.
6. Click the trace type you want to enable.
7. Click **Apply**.

The trace specification is updated to reflect the trace type that you selected. For example, to turn on all tracing for the Common Event Infrastructure components, the trace specification is `CommonEventInfrastructure=all=enabled`.

Log files are written to the location configured for the application server.

Removing the Common Event Infrastructure configuration

If you need to uninstall the Common Event Infrastructure, you must first remove the deployed enterprise applications and the database configuration.

To remove the Common Event Infrastructure, follow these steps:

1. Remove the Common Event Infrastructure application.
2. Remove the event messaging application.
3. Remove the event database.

Removing the Common Event Infrastructure application

If you need to remove the event server enterprise application and resources from WebSphere Application Server, you can use the `event-application.jacl` script.

If you prefer, you can remove the event server enterprise applications manually using the administrative console rather than using the `event-application.jacl` script. If use the administrative console, you must also manually remove the Common Event Infrastructure resources. For more information about these resources, see "Default configuration" on page 9.

To remove the event server enterprise application, use the wsadmin tool to run the `event-application.jacl` script.

To run the script, go to the *install_root*`/event/application` directory and run the following command (all on one line):

```
wsadmin.sh -f event-application.jacl -profile event-profile.jacl
  -wsadmin_classpath install_root/event/lib/cei_installer.jar -action uninstall
  -node node_name -server server_name
  [-appname app_name] [-trace]
```

The `event-application.jacl` script uses these parameters:

node_name
The WebSphere Application Server node from which you want to remove the event server enterprise application.

server_name
The WebSphere Application Server from which you want to remove the event server enterprise application. This parameter is optional. If you do not specify a server, the enterprise application is removed from all servers in the node.

app_name
The name of the deployed event server enterprise application you want to remove. This parameter is optional. If you do not specify an application name, all registered Common Event Infrastructure enterprise applications are removed.

The optional **-trace** parameter causes additional debugging information to display on the standard output.

Removing the event messaging enterprise application

Before uninstalling the Common Event Infrastructure, you must remove the event messaging enterprise application.
To remove the event messaging enterprise application, use the wsadmin tool to run the `event-message.jacl` script.

To run the script, go to the *install_root*`/event/application` directory and run the following command (all on one line):

```
wsadmin.sh -f event-message.jacl -profile event-profile.jacl
  -wsadmin_classpath install_root/event/lib/cei_installer.jar -action install
  -node node_name -server server_name
  [-eventprofilescope scope] -appname app_name [-trace]
```

The parameters of the `event-message.jacl` script are as follows:

node_name
The WebSphere Application Server node from which you want to remove the event messaging enterprise application.

To find out the node name, run the *install_root*`/bin/setupCmdLine` and then the `echo $WAS_NODE` command.

server_name
The WebSphere Application Server from which you want to remove the event messaging enterprise application. This parameter is optional. If you do not specify a server, the application is removed from all servers in the specified node.

scope
The scope of the Common Event Infrastructure configuration profile objects to be removed. This parameter is optional. If you specify a scope, a the JMS transmission profile and emitter factory profiles in the specified scope are removed. The valid values are `cell`, `node`, and `server`.

app_name
The name of the deployed messaging enterprise application you want to remove. This parameter is required.

The optional **-trace** parameter causes additional debugging information to display on the standard output.

Removing the event database

If you need to remove the event database, you can use the provided scripts. You must remove the database before you uninstall the Common Event Infrastructure.

When the database is configured, the configuration script also creates scripts for removing the database and the Java database connectivity (JDBC) provider. The scripts for removing the event database are placed in database-specific subdirectories of the *install_root*`/event/dbscripts` directory. The scripts for removing the JDBC provider are placed in database-specific subdirectories of the *install_root*`/event/dsscripts` directory.

Note: The event database can be shared among multiple event servers using the same JDBC provider configuration. Therefore, remove the JDBC provider configuration only if you have uninstalled the associated event database.

To remove the event database and JDBC provider, run the appropriate scripts from the following table.

Type	Operating system	Database script	JDBC configuration script
Cloudscape	z/OS (Windows script)	`rm_event_cloudscape.bat`	`rm_cloudscape_jdbc_provider.bat`
Cloudscape	z/OS (Linux/UNIX script)	`rm_event_cloudscape.sh`	`rm_cloudscape_jdbc_provider.sh`
DB2	z/OS (Windows script)	`rm_event_db2zos.bat`	`rm_db2zos_jdbc_provider.bat`
DB2	z/OS (Linux/UNIX script)	`rm_event_db2zos.sh`	`rm_db2zos_jdbc_provider.sh`

You can remove the event database or JDBC provider at any time by running the appropriate script. To remove the JDBC provider, use the appropriate script and specify the scope in which you want to remove the JDBC provider:

`rm_`*db*`_jdbc_provider` *scope* [*server_name*]

The generated scripts use these parameters:

scope The scope in which you want to remove the JDBC provider. The valid values are `cell`, `node`, and `server`.

server_name
The name of the WebSphere Application Server from which you want to remove the JDBC provider, if **scope** is `server`. (If **scope** is `cell` or `node`, this parameter is ignored.

Chapter 5. Working with events

The Common Event Infrastructure represents events as Java objects. Specifically, each event is an instance of a class that implements the org.eclipse.hyades.logging.events.cbe.CommonBaseEvent interface, which is a Java representation of the Common Base Event specification. The org.eclipse.hyades.logging.events.cbe package is part of the Eclipse-based Hyades environment, which is a set of standards and open-source tools for testing, tracing, and monitoring. For more information, see http://www.eclipse.org/hyades/.

For more information about the XML Schema specification, see http://www.w3.org/XML/Schema.

Working with events involves the following tasks.

1. "Creating an event object" on page 32
2. "Retrieving data from a received event" on page 38
3. Optional: "Converting XML events" on page 39
4. "Accessing event instance metadata" on page 39

Related concepts

"The Common Base Event model" on page 2

Life cycle of an event

The typical life cycle of an event is as follows:

1. To send an event, an event source creates a CommonBaseEvent instance, populates it with property data, and then submits it to an emitter.
2. The emitter optionally uses the content completion mechanism (if implemented) to populate the event with required property data. The emitter then validates the event and checks it against the currently configured filter criteria. If the event is valid and passes the filter criteria, the emitter sends the event to the event server.
3. If persistence is enabled, the event server stores the event in a persistent data store.
4. If publishing is enabled, the event server publishes the event to one or more Java Messaging Service (JMS) destinations. Event consumers subscribing to these destinations then receive notifications of the new event. The event consumers then use the notification helper to convert the received JMS messages back into a CommonBaseEvent instance.

   An event consumer might also submit a query to retrieve the event from the data store. Typically, a consumer uses the query interface to retrieve historical events, especially during startup processing.

   After receiving the event, an event consumer reads the event property data and processes the event.
5. When it is no longer needed, the event can be purged from the data store.

Related tasks

"Sending events" on page 43

Chapter 7, "Developing an event consumer," on page 51

Event property data

The Common Base Event specification, which is based on the XML Schema definition language, defines two kinds of event property data:

- Properties that are represented by simple data types, encoded in XML as attributes of the CommonBaseEvent element.

  These properties include globalInstanceId, severity, and msg. The CommonBaseEvent Java class represents these values as strings or integers, as appropriate.
- Properties that are represented by complex data types and encoded in XML as subelements of the CommonBaseEvent element.

  These properties include situation, sourceComponentId, and extendedDataElements, each of which has nested properties of its own. These complex types are represented by specialized Java classes defined in the org.eclipse.hyades.logging.events.cbe package. For example, the sourceComponentId property is represented by a ComponentIdentifier instance.

Most event properties are defined as optional by the Common Base Event specification, but the following properties are required:

- version (a string attribute)
- creationTime (an XML Schema dateTime attribute; see http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/#dateTime)
- sourceComponentId (a complex ComponentIdentification element)
- situation (a complex Situation element)

The version attribute is defined as optional by the Common Base Event specification, but if it is not specified, the default value `1.0` is assumed. Because the Common Event Infrastructure supports only version 1.0.1 of the specification, you must specify `1.0.1`.

For more information on the Common Base Event specification, see the *Autonomic Computing Toolkit Developers' Guide* delivered with the IBM Autonomic Computing Toolkit, http://www.ibm.com/autonomic.

If you try to send an event that is missing any of these properties, the emitter rejects the event and throws an EventsException exception.

The CommonBaseEvent class defines getter and setter methods for each property, and helper methods to simplify creation of complex properties. An event source uses the setter methods (or the helper methods) to populate an event with property data before submitting it to an emitter. An event consumer uses the getter methods to retrieve the property data from a received event.

Creating an event object

To create new events in your event source, you use an *event factory*, which is an object that returns new CommonBaseEvent instances or the specialized classes that represent complex property data types.

1. Access an event factory.
   - Create an event factory using the event-factory factory. Use this approach if no appropriate event factory is already available. When you create a new

event factory, you can optionally specify a content handler to provide automatic content completion. See "Creating a new event factory."

- Use an existing event factory that is bound into a Java Naming and Directory Interface (JNDI) namespace. Use this approach if an administrator has provided an event factory for you to use. This approach ensures that any events that you create conform to the appropriate business rules, because the event factory might be configured with a content handler. See "Getting an event factory by JNDI lookup" on page 34.
- Access the event factory indirectly by using the com.ibm.websphere.cem.ECSEmitter class to create a common base event object. Use this approach if an event factory is bound into the JNDI namespace; this is usually done automatically in the WebSphere Application Server context. See 5.1.1 + "Creating and populating an event using the ECSEmitter class" on page 34.

2. Create and populate an event.
   - Create and populate an event using the ECSEmitter class. The option uses the event factory indirectly. See 5.1.1 + "Creating and populating an event using the ECSEmitter class" on page 34.
   - Create and populate an event using the event factory directly. See "Creating and populating an event using the event factory directly" on page 36.
   - Set property data automatically. See "Setting property data automatically" on page 37.

Related concepts

"Event property data" on page 32

Creating a new event factory

To create a new event factory, use the event-factory factory, implemented as the EventFactoryFactory class. This class has no instances. Instead, this class provides two static methods to create event factories. The choice of which method to use depends upon whether you want to use a content handler to set property data automatically.

- To create a generic event factory with no content handler, use the createEventFactory static method.

```
EventFactory eventFactory =
     (EventFactory) EventFactoryFactory.createEventFactory();
```

- To create an event factory with a content handler, use the createEventFactory(ContentHandler) method and specify the content handler you want to use.

```
EventFactory eventFactory =
     (EventFactory) EventFactoryFactory.createEventFactory(contentHandler);
```

In both cases, the returned object is an event factory that you can use to create new events.

You can now create event objects and populate them with property data.

Related tasks

"Setting property data automatically" on page 37

"Creating and populating an event using the event factory directly" on page 36

Getting an event factory by JNDI lookup

If an administrator has bound an existing event factory into a Java Naming and Directory (JNDI) for event sources to use, perform a standard JNDI lookup to retrieve the event factory:

```
Context context = new InitialContext();
EventFactory eventFactory =
     (EventFactory) context.lookup("com/ibm/events/EventFactory");
```

The returned object is the provided event factory. If the event factory is configured with a content handler, an instance of the content handler is also created locally. For more information about content handlers and JNDI, see "Setting property data automatically" on page 37.

Create event objects and populate them with property data.

Related tasks

"Creating and populating an event using the event factory directly" on page 36

Creating and populating an event using the ECSEmitter class

If an event factory is bound into the Java Naming and Directory Interface (JNDI) namespace, you can access the event factory indirectly. You can use the com.ibm.websphere.cem.ECSEmitter class to create and populate a common base event. This class provides the following methods:

- createCommonBaseEvent method. If you use this method, you need provide only the extension name and the situation properties for the common base event. All other properties are set automatically.
- addUserDataEvent method. If you use this method, all of the mandatory properties are set automatically. The extension name is set to `ECS:UserDataEvent` and the situation is set to `ReportSituation`. You can set extended data elements for the common base event by passing a set of properties.

You can create and populate a common base event in one of the following ways.

- Use the createCommonBaseEvent method to create and populate an event.

  The following code fragment starts a new event correlation sphere, newECSID, and then uses the createCommonBaseEvent method to create an event object. For more information on event correlation spheres, see "Event context elements" on page 5

  ```
  ECSEmitter myEmitter = new ECSEmitter("JNDI Emitter Factory Name", "newECSID");
  CommonBaseEvent myEvent = myEmitter.createCommonBaseEvent("myEventType");

  // get situation object
  Situation mySituation = myEvent.getSituation();

  // set situation properties
  mySituation.setCategoryName("ReportSituation");
  mySituation.setReportSituation("EXTERNAL", "STATUS");

  // add other information to the the event

  // send the event
  myEmitter.sendEvent(myEvent);
  ```

  This example uses the constructor method of the ECSEmitter class to create an emitter, passing the JNDI name of an existing Common Event Infrastructure emitter and the identification of a new event correlation sphere.

The new emitter is then used to create a common base event that contains a situation object that is accessed using the getSituation call. The setCategoryName and setReportSituation methods are used to set the mandatory data in the situation object to emit an event with a `ReportSituation`. To create other situation types, use different category names in the setCategoryName call and different setter method calls for the situation.

All other mandatory information is provided automatically by the run-time environment. If mandatory information is set explicitly in the common base event, this information is not overwritten with the default information. The event is now valid and can be submitted to an emitter using the sendEvent method.

In an actual application, a useful event needs to include more information than is shown in this example, but these properties are the minimum required by the Common Base Event specification and the Common Event Infrastructure.

- Use the addUserDataEvent method to create and populate an event.

  The following code fragment uses the addUserDataEvent method to create an event object in the current event correlation sphere.

```
ECSEmitter myEmitter = new ECSEmitter("JNDI Emitter Factory Name", null);

// prepare a set of user data properties
Properties myUserData = new Properties();
myUserData.setProperty("UserData1","UserDataValue1");
myUserData.setProperty("UserData2","UserDataValue2");

// create and send the event
myEmitter.addUserDataEvent(myUserData);
```

  This example uses the constructor method of the ECSEmitter class to create an emitter, passing the JNDI name of an existing Common Event Infrastructure emitter. An event correlation sphere identifier is not passed (`null`) and therefore a new event correlation sphere is not started. If an event correlation sphere exists, the user data event is added to this correlation sphere.

  A set of user data properties is then prepared. Name and value pairs are added to a property list.

  The last step in the example creates and sends a common base event using the addUserDataEvent method of the new emitter. The extensionName property of the new common base event is set to `ECS:UserDataEvent`, the situation is set to `ReportSituation`, and all other mandatory information is provided automatically by the run-time environment.

  Related concepts

  "Event property data" on page 32

  Related tasks

  "Obtaining an emitter" on page 42

  Related reference

  "Identification of the event source" on page 4
  This topic describes the event data that is automatically provided in common base events that occur in the WebSphere Application Server environment.

  "Event context elements" on page 5

Creating and populating an event using the event factory directly

5.1.1+ If you do not use the com.ibm.websphere.cem.ECSEmitter class to create and populate an event, you must use the event factory directly. The following code fragment uses an event factory to create an event and populates it with the minimal required property data:

```
CommonBaseEvent event = eventFactory.createCommonBaseEvent();

event.setVersion("1.0.1");                    // set version

long currentTime = System.currentTimeMillis(); // get current time
event.setCreationTimeAsLong(currentTime);      // and set creationTime

// set sourceComponentId (a complex type)
event.setSourceComponentId("Windows",          // application
                           "svchost.exe",      // component
                           "tlntsvr.exe",      // subcomponent
                           "http://www.ibm.com/namespaces/autonomic/Windows",
                                               // componentType
                           "win386_svc",       // componentIdType
                           "9.45.72.138",      // location
                           "IPV4"              // locationType
                           );

// create situation object
Situation situation = eventFactory.createSituation();

// set situationType to AvailableSituation (a complex type)
situation.setAvailableSituation("EXTERNAL",          // reasoningScope
                                "NOT AVAILABLE",     // availabilityDisposition
                                "STARTABLE",         // operationDisposition
                                "FUNCTION_PROCESS"); // processingDisposition

// set situation
event.setSituation(situation);
```

This example uses an event factory to create an event instance. First the version property is set. Then the current system time is retrieved and the setCreationTimeAsLong(long) method is used to set the value of the creationTime property. An alternative approach is to use the setCreationTime(String) method, which sets the creation time using the XML dateTime format, for example, `"2004-07-29T13:12:00-05:00"`.

The next required property, sourceComponentId, is a complex property that is represented by a ComponentIdentification instance, which has properties of its own. However, it is not necessary to directly instantiate or interact with this object. Instead, the next statement in the example uses the setSourceComponentId helper method, to specify the nested properties. The helper method uses these values to create a ComponentIdentification instance, which it then uses to set the value of the sourceComponentId property of the event.

Similar helper methods exist for setting other complex properties, for example, the setMsgDataElement, addAssociatedEvent, and addExtendedDataElement methods. Many of these methods exist in multiple versions with different signatures, making it possible to specify property values in different ways. Refer to the Javadoc API documentation for complete information on these methods.

The last required property in the example, situation, is another complex property. In this case, the situation object must be instantiated directly using an event factory. The example then uses a helper method to set the situationType property, which is itself a complex subelement.

In an actual application, a useful event needs to include more information than is shown in this example, but these properties are the minimum required by the Common Base Event specification and the Common Event Infrastructure. The event is now valid and can be submitted to an emitter.

Related concepts

"Event property data" on page 32

Related tasks

5.1.1 + "Creating and populating an event using the ECSEmitter class" on page 34

"Obtaining an emitter" on page 42

Related reference

"Identification of the event source" on page 4
This topic describes the event data that is automatically provided in common base events that occur in the WebSphere Application Server environment.

Setting property data automatically

In some situations, you might want some event property data set automatically for every event that you create. This is a way to fill in certain standard values that do not change (such as the application name), or to set some properties based on information that is available from the run-time environment (such as creation time or thread information). You can also set policies that govern event content according to business rules. For example, you might require that events with a particular extension name have the severity set to a certain value.

You can set property data automatically by creating a *content handler*. A content handler is an object that automatically sets the property values of each event based on any arbitrary policies that you want to use. The Common Event Infrastructure does not restrict how a content handler can modify event data, so long as the event still conforms to the Common Base Event specification.

To ensure that all event sources comply with the same policies, you can create an event factory associated with a content handler (using the EventFactoryFactory class) and then bind the created event factory into a Java Naming and Directory Interface (JNDI) namespace. Instead of creating event factories, event sources can perform JNDI lookups to access the existing event factory, without any knowledge of the content handler. If your business rules change, you can modify the content handler in one place.

An event source does not need to do anything to set property data automatically. If an event factory is associated with a content handler, each event that the factory creates carries a reference to that content handler. When the event is submitted to an emitter, the event calls the completeEvent method of the content handler, passing a reference to itself. This ensures that the correct policies are applied to the event after the event source is finished setting event-specific properties, but before the event is validated and processed by the emitter.

When an event is transmitted from one process to another, the reference to the content handler is not transmitted with it. This is because content completion relies upon the environment where the event originates, and the necessary information might not be available elsewhere. This restriction does not affect calls between applications that are local to one another, for example, a call to an enterprise bean using its local interface.

To create a content handler, follow these steps:

1. Create a new Java class that mplements the org.eclipse.hyades.logging.events.cbe.ContentHandler interface. This interface defines a single completeEvent(CommonBaseEvent) method. The parameter is the event for which the content is to be set. In your implementation of this method, you can use the getter and setter methods of the CommonBaseEvent interface to process the event property data in accordance with any policies that apply.

   When an event source uses JNDI to retrieve an event factory, the content handler and the event factory are returned. For this reason, the content handler must be serializable.

   The following example is a simple content handler that automatically sets the extension name of each event:

   ```
   import java.io.Serializable;
   import org.eclipse.hyades.logging.events.cbe.*;

   public class BusinessContentHandler
        implements ContentHandler, Serializable {
          public void completeEvent(CommonBaseEvent event)
             throws CompletionException {
                 event.setExtensionName("business");
          }
       }
   ```

2. Associate the content handler with an event factory. To do this, specify the content handler when you create the event factory:

   ```
   import org.eclipse.hyades.logging.events.cbe.*;

   EventFactory eventFactory =
        (EventFactory) EventFactoryFactory.createEventFactory(contentHandler);
   ```

   The returned event factory is permanently associated with the specified content handler.

Retrieving data from a received event

When an event source receives an event, it can then use the getter methods of the CommonBaseEvent interface to retrieve the event property data. For example, the following code fragment retrieves a single event and then reads the content of the msg property.

```
CommonBaseEvent event = eventAccess.queryEventByGlobalInstanceId(eventId);
String eventMessage = event.getMsg();
```

If the property that you want to retrieve is a complex property (a CommonBaseEvent subelement in the Common Base Event specification), the returned value is an instance of the specialized class representing the complex data type. You can then use the getter methods of the returned object to retrieve the property data from that object. For example, the following code fragment retrieves

the value of componentId property, which is a complex property. The code then retrieves the content of the nested component property, which is a string, to read the name of the source component.

```
CommonBaseEvent event = eventAccess.queryEventByGlobalInstanceId(eventId);
ComponentIdentification componentId = event.getSourceComponentId();
String componentName = componentId.getComponent();
```

Related tasks

"Querying events from the event server" on page 55

Converting XML events

In addition to creating new events, an event source might convert events in XML format that it received from other applications. Similarly, an event consumer might convert events to XML format for forwarding to another application. The org.eclipse.hyades.logging.events.cbe.EventFormatter class provides methods you can use to convert between CommonBaseEvent instances and XML.

Use the EventFormatter class to convert a CommonBaseEvent instance into a string containing either an XML document or an XML fragment. Similarly, you can convert from an XML document or fragment to a CommonBaseEvent instance.

For more information about the EventFormatter class, refer to the Javadoc documentation in the org.eclipse.hyades.logging.events.cbe package.

Accessing event instance metadata

The org.eclipse.hyades.logging.events.cbe package, which provides the classes and interfaces that are required for working with event objects, is based on the Eclipse Modeling Framework (EMF). EMF is a Java framework used to generate application code that is based on a structured data model. It also provides interfaces in the generated code that can be used to access metadata that describes the data model. Refer to the Eclipse Modeling Framework documentation at http://www.eclipse.org/emf for more information about EMF.

By using these interfaces, EMF-compatible tools can interact with CommonBaseEvent event data without any prior knowledge of the data model or access to the implementation. This interaction makes it possible for development tools to generate code that transfers data from other data models into the CommonBaseEvent model. Application developers can then focus on writing code that uses the data rather than the code that builds the data.

For example, consider an event source that monitors network events and describes its own data model in terms of EMF. With access to both data models, a development tool can display the fields of the event source data model alongside the fields of the CommonBaseEvent data model. A developer can then use a graphical interface to indicate how the fields in the event source model map to fields in the CommonBaseEvent model. For example, a Workstation.name field in the event source data model might correspond to the CommonBaseEvent.sourceComponentId.location field in the CommonBaseEvent data model. Because both data models are described using standard EMF interfaces, the tool can generate code that handles the transfer of data between the two models.

The following code fragment is a simple example of how a development tool might use EMF interfaces to query information about the CommonBaseEvent data model and then use that information to interact with an event instance. This example can be part of a simple event consumer. The example iterates through all of the fields of an event instance and prints the name and the value of each field.

```
// event is a valid CommonBaseEvent instance

// Get list of event instance structural features (fields)
List features = event.eClass().getEAllStructuralFeatures();

// iterate through list; print names and values
for (int i = 0 ; i < features.size() ; i++)
  {
    EStructuralFeature feature = (EStructuralFeature)features.get(i);
    Object value = eObj.eGet(feature);
    System.out.println(feature.getName() + ":" + value);
  }
```

The CommonBaseEvent data model is described in the `cbe.ecore` and `cbe.genmodel` EMF files. These files are included with the Common Event Infrastructure SDK. You can import the files into an Eclipse-based development environment and then use EMF to generate code that interacts with CommonBaseEvent objects.

Chapter 6. Developing an event source

If your event source is running with Java 2 security enabled, you must modify your policy file to enable the correct processing of globally unique identifiers (GUIDs). Add an entry to the file that allows read, write, and delete access to the `guid.lock` file:

```
permission java.io.FilePermission "${java.io.tmpdir}${/}guid.lock",
  "read, write, delete";
```

An *event source* is any application that uses an emitter to send events to the event server. The following applications are examples of event sources:

- An adapter or monitor that generates events related to monitored resources
- An application that generates notification events
- An application that forwards events from other sources

An event source is implemented in the Java programming language, using either the Java 2 Platform, Standard Edition (J2SE) or the Java 2 Platform, Enterprise Edition (J2EE). An event source must submit valid events conforming to the Common Base Event model. Each event is represented as a Java object.

An administrator can create multiple emitter factory profiles, each one defining a different emitter configuration. An event source obtains an emitter using the emitter factory associated with an existing emitter factory profile. Therefore, all of the emitters that are created by a particular emitter factory have the same default behavior.

1. Obtain an emitter. See "Obtaining an emitter" on page 42.
2. Send the event. When you send a event you can specify different emitter settings. See "Sending events" on page 43.
3. Free emitter resources. When an event source finishes sending events, free the resources that the emitter is using. See "Freeing emitter resources" on page 47.
4. Filter events. Event filtering provides a way to reduce traffic by screening out events that are not important. See "Filtering events" on page 47.

Related concepts

"The Common Base Event model" on page 2

Emitters and emitter factories

An event source does not interact directly with the event server. Instead, it interacts with an object called an *emitter*, an implementation of the com.ibm.events.emitter.Emitter interface. An emitter is a local object that provides methods for sending events.

In general, the emitter handles the details of event transmission. The developer of an event source does not need to know the event server location, the filter settings, or the underlying transmission mechanism. These details are governed by the *emitter factory*, an object that is configured by an administrator and bound in a Java Naming and Directory Interface (JNDI) namespace. An emitter factory is an instance of com.ibm.events.emitter.EmitterFactory and is used to create emitter objects. It also defines the behavior of the emitters it creates.

An emitter factory includes the following settings:

- The preferred *transaction mode*. This setting specifies whether the emitter attempts to send each event in a new transaction or within the current transaction. An event source can change this setting for a particular emitter or event submission, but the profile specifies the default value. This setting is valid only in a Java 2 Platform, Enterprise Edition (J2EE) container. The Java 2 Platform, Standard Edition (J2SE) platform does not provide transaction controls.
- The preferred *synchronization mode*. This setting specifies whether events are sent using synchronous or asynchronous transmission. *Synchronous transmission* means that the sendEvent method does not return control to the caller until the event is processed. *Asynchronous transmission* means that the method returns immediately after the event is submitted, and the caller has no further information about event processing. An event source can change this setting for an emitter or for an event submission, but the default value is specified by the profile.
- The *transmission profiles* to use. A transmission profile is a configuration object that defines a specific transmission mechanism for sending events to the event server. An emitter factory profile can specify two transmission profiles, one for synchronous transmission and one for asynchronous transmission. An event source cannot change the transmission profiles used by an emitter.
- The filter configuration to use for the emitter. The filter configuration defines what filtering plug-in is used to filter events submitted to the emitter. The Common Event Infrastructure includes a default filter plug-in, but you can also implement your own filter plug-in if you want to use a different filtering engine.

Related tasks

"Filtering events" on page 47

Obtaining an emitter

Before you can obtain an emitter, there must be at least one emitter factory profile configured. For each emitter factory profile, an emitter factory is automatically created and is accessible using the Java Naming and Directory Interface (JNDI) name of the emitter factory profile. For more information, see "Default configuration" on page 9.

You can obtain an emitter in one of the following ways:

- 5.1.1 + Use the constructor method of the com.ibm.websphere.cem.ECSEmitter class
- Use the registered Common Event Infrastructure emitter factory directly

5.1.1 + In a WebSphere environment, it is recommended that you use the ECSEmitter class. This approach guarantees that the chain of correlation information is preserved.

You can obtain an emitter in one of the following ways.

- 5.1.1 + Use the constructor method of the ECSEmitter class to obtain an emitter. The following code fragment obtains an emitter configured with the default profile.

```
import com.ibm.websphere.cem.ECSEmitter;

ECSEmitter myEmitter =
   new ECSEmitter("com/ibm/events/configuration/emitter/Default", null);
```

In this example, the identifier of an event correlation sphere is not passed (`null`). This means that a new correlation sphere is not started for the event.

- Use the registered Common Event Infrastructure emitter factory directly to obtain an emitter.
  1. Perform a JNDI lookup specifying the name of the emitter factory that you want to use for your emitter. This is the JNDI name that was specified by an administrator when the emitter factory profile was defined.
  2. Call the getEmitter method of the emitter factory.

     If your event source is a Java 2 Platform, Standard Edition (J2SE) client application running in a secure environment, and the emitter profile you are using specifies asynchronous transmission profiles, you must specify a Java Message Service (JMS) user name and password when getting an emitter. To do this, use the getEmitter(String, String) method, passing the JMS user name and password that you want to use. For more information, refer to the Javadoc API documentation for the com.ibm.events.emitter.Emitter class.

     The following code fragment obtains an emitter that is configured with the default profile:

```
import javax.naming.*
import com.ibm.events.emitter.*

Context context = new InitialContext();
EmitterFactory emitterFactory =
  (EmitterFactory) context.lookup
    ("com/ibm/events/configuration/emitter/Default");
Emitter emitter = emitterFactory.getEmitter();
```

     The returned object is an emitter that is configured according to the options that are defined in the emitter factory profile. If the emitter factory cannot obtain an emitter, it throws an EmitterException exception.

Sending events

An event source sends events in the form of Java objects. Specifically, each event is an instance of a class implementing the org.eclipse.hyades.logging.events.cbe.CommonBaseEvent interface, which is a Java representation of the Common Base Event specification.

5.1.1 + To send an event, use the sendEvent methods of the ECSEmitter class or the Emitter interface. If you use the ECSEmitter class, you can use the addUserDataEvent method. This method creates a common base event with predefined properties before the event is sent. See “Creating and populating an event using the ECSEmitter class” on page 34 for more information.

When you submit an event to an emitter, the following things happen:

1. The emitter calls the complete method of the event, triggering optional content completion. See “Setting property data automatically” on page 37 for more information.
2. The emitter assigns a sequence number and a global instance identifier to any event that does not already have them. The emitter then validates the event to ensure that it conforms to the Common Base Event specification.
3. If filtering is active, the emitter checks the event against the current filter criteria to determine whether the event should be sent or discarded.
4. If the event is valid and passes the filter criteria, the emitter sends the event to the event server for persistence and distribution to event consumers.

If the event is not valid, or if the emitter encounters a problem when it sends the event to the event server, an exception is thrown.

The current Common Base Event specification allows only one extended data element with a given name at each level of the event containment hierarchy, but this restriction is not enforced by the Common Event Infrastructure.

You can send an event in the following ways.

- Send an event with the current emitter settings. See "Sending an event with the current emitter settings."
- Send an event that overrides the current emitter settings. See "Overriding the current emitter settings."
- Change the emitter settings. See "Changing the emitter settings" on page 46.

Related concepts

"The Common Base Event model" on page 2

Sending an event with the current emitter settings

If you do not need to specify a particular transmission mode or transaction mode, you can send an event using the current emitter settings. These settings are initially defined by an administrator in the emitter factory profile, but they can later be changed by event consumers.

You can send an event with the current emitter settings in one of the following ways.

- Use the sendEvent(CommonBaseEvent) method to send an event using the current emitter settings.

  ```
  String eventId = emitter.sendEvent(event);
  ```

  5.1.1+ In this example, `emitter` is an Emitter instance or an ECSEmitter instance, and `event` is a CommonBaseEvent instance. The returned value, `eventId`, is the globally unique identifier of the event, which is the value of the *globalInstanceId* field of the CommonBaseEvent instance. If the event does not have a global instance identifier, the emitter assigns one automatically.

- 5.1.1+ Use the addUserDataEvent(Properties) method of the ECSEmitter class to send a user data event using the current emitter settings.

  ```
  emitter.addUserDataEvent(userData);
  ```

  In this example, `emitter` is an ECSEmitter instance and `userData` is a java.util.Properties instance.

If an event is submitted to an emitter, this action does not guarantee that the event is sent to the event server, because the filter settings might cause the event to be discarded. A successful call to the sendEvent method means only that the event was successfully processed by the emitter.

Overriding the current emitter settings

When sending an event, you can specify options that override the current transaction mode and the synchronization mode that are configured for the emitter. These settings are initially defined by an administrator in the emitter factory profile, but they can be changed by event consumers.

An emitter might not support all synchronization and transaction modes. The available modes are subject to the following limitations:

- The synchronization modes supported by an emitter are defined by the emitter factory profile. You can find out which modes are supported by a particular emitter by calling the isSynchronizationModeSupported method. See the Javadoc API documentation for com.ibm.events.emitter.Emitter for more information.
- Transactions are supported only in a J2EE container.

If you attempt to use a mode that is not supported, the emitter throws a TransactionModeNotSupportedException or SynchronizationModeNotSupportedException exception.
To override the emitter settings, use the sendEvent(CommonBaseEvent, int, int) method.

```
String eventId = emitter.sendEvent(event,
                                   synchronizationMode,
                                   transactionMode);
```

The parameters are as follows:

event
The event object, a CommonBaseEvent instance, that you want to send.

synchronizationMode
An integer constant that is defined by the SynchronizationMode interface. This value is one of the following constants:
- SynchronizationMode.ASYNCHRONOUS (send the event asynchronously)
- SynchronizationMode.SYNCHRONOUS (send the event synchronously)
- SynchronizationMode.DEFAULT (use the current emitter setting)

transactionMode
An integer constant that is defined by the TransactionMode interface:
- TransactionMode.NEW (send the event in a new transaction)
- TransactionMode.SAME (send the event in the current transaction)
- TransactionMode.DEFAULT (use the current emitter setting)

The event is sent with the options you specify. These options apply only to the single event that is sent. No changes are made to the emitter settings, and subsequent event submissions are not affected.

The returned value, eventId, is the globally unique identifier of the event, which is the value of the globalInstanceId field of the CommonBaseEvent instance. If the event does not have a global instance identifier, the emitter assigns one automatically.

If an event is submitted to an emitter, this action does not guarantee that the event is sent to the event server, because the filter settings might cause the event to be discarded. A successful call to the sendEvent method means only that the event was successfully processed by the emitter.

The following example overrides the emitter setting to send an event in a new transaction, but it does not override the synchronization mode:

```
String eventId = sendEvent(event,
                           SynchronizationMode.DEFAULT,
                           TransactionMode.NEW);
```

Changing the emitter settings

An event source can make changes to the transaction mode and synchronization mode configured for the emitter. These settings are initially defined by the emitter factory profile. In addition, an event source can query the current transaction mode to determine what setting is currently in effect for the emitter.

You can change the emitter settings in the following ways:

- "Changing the synchronization mode"
- "Changing the transaction mode"
- "Querying the transaction mode" on page 47

Changing the synchronization mode

An event source can change the synchronization mode that is used by an emitter. This change remains in effect for subsequent event submissions, but it does not change the preferred synchronization mode defined in the emitter factory profile.

The synchronization modes supported by an emitter are defined by the emitter factory profile. You can find out which modes are supported by a particular emitter by calling the isSynchronizationModeSupported method. See the Javadoc API documentation for com.ibm.events.emitter.Emitter for more information. If you use a mode that is not supported, the emitter throws a SynchronizationModeNotSupportedException exception.
To change the synchronization mode, use the setSynchronizationMode(int) method.

```
emitter.setSynchronizationMode(synchronizationMode);
```

The *synchronizationMode* is an integer constant defined by the interface SynchronizationMode:

- SynchronizationMode.ASYNCHRONOUS (send the event asynchronously)
- SynchronizationMode.SYNCHRONOUS (send the event synchronously)
- SynchronizationMode.DEFAULT (send the event using the current emitter settings)

Changing the transaction mode

An event source can change the transaction mode that is used by an emitter. This change remains in effect for subsequent event submissions, but it does not change the transaction mode defined in the emitter factory profile.

Note: Transactions are supported only in a Java 2 Platform, Enterprise Edition (J2EE) container.

To change the transaction mode, use the setTransactionMode(int) method.

```
emitter.setTransactionMode(transactionMode);
```

The *transactionMode* is an integer constant defined by the TransactionMode interface:

- TransactionMode.NEW (send the event in a new transaction)
- TransactionMode.SAME (send the event in the current transaction)
- TransactionMode.DEFAULT (send the event using the current emitter settings)

Querying the transaction mode

An event source can query the transaction mode that is used by an emitter.

Note: Transactions are supported only in a J2EE container.
To query the current transaction mode, use the getTransactionMode method.

```
int transactionMode = emitter.getTransactionMode();
```

The returned value is an integer that corresponds to one of the transaction mode constants:

- TransactionMode.NEW
- TransactionMode.SAME

Freeing emitter resources

If your event source has finished sending events with a particular emitter, you should free the resources the emitter is using. If you are using the ECSEmitter class for correlation-aware events, you should also close the correlation sphere.

1. To free the emitter resources, use the close method:

   ```
   emitter.close();
   ```

   This method releases all that resources that the emitter uses.

2. 5.1.1 + Optional: Close the correlation sphere.

   ```
   emitter.releaseAndEndECS(id);
   ```

   The correlation sphere is closed. You cannot send anymore events with this emitter.

Filtering events

An emitter can optionally be configured to filter events at the source. Event filtering provides a mechanism for reducing event traffic by screening out events that are not important. Each time an event source submits an event to an emitter, the emitter checks the event against the current filter criteria. If the event passes the filter criteria, the emitter sends the event to the event server; otherwise, the emitter discards the event. In any case, an event source cannot change the filter settings, which are configured by an administrator.

The emitter filter is implemented as a separate component called a *filter plug-in*. If you want to use a different filter mechanism, you can implement your own filter plug-in.

In the Common Event Infrastructure configuration, each emitter factory is associated with a *filter factory*. A filter factory is an object used to create instances of a filter plug-in. When you create an emitter using an emitter factory, the emitter is automatically associated with an instance of the specified filter plug-in, which provides filtering of events submitted to that emitter.

- Filter events with the default filter plug-in. The Common Event Infrastructure includes a default filter plug-in, which provides filtering of submitted events based on XPath event selectors. See “Filtering events with the default filter plug-in” on page 48.

- Filter events with a custom filter plug-in. If you do not want to use the default filter plug-in, you can implement your own plug-in to filter events. See "Implementing a filter plug-in."

Filtering events with the default filter plug-in

The Common Event Infrastructure includes a default emitter filter plug-in. You can configure this plug-in with an XPath event selector to define which events are sent to the event server and which are discarded. For example, the filter settings might specify that only events with a severity greater than 20 (harmless) should be sent.

To filter events using the default filter plug-in, follow these steps:

1. In the WebSphere administrative console, navigate to the **Common Event Infrastructure Provider > Filter Factory Profile** page.
2. Create a new filter factory profile. For more information, see the online help for the administrative console.
3. In the **Filter Configuration String** field, specify an XPath event selector that describes the events you want to use for filtering events. Events that match this event selector are sent to the event server; events that do not match the event selector are discarded by the emitter.
4. Navigate to the **Common Event Infrastructure Provider > Emitter Factory Profile** page.
5. Create a new emitter factory profile, or go to an existing emitter factory profile. For more information, see the online help for the administrative console.
6. In the **Filter Factory JNDI Name** field, specify the JNDI name of the new filter factory profile you created.

Event sources can now use the new emitter factory to create instances of an emitter using the new filter configuration. If you want to adjust the filter settings for event sources that use this emitter factory, you can modify the event selector that is specified in the filter factory.

The default filter plug-in uses the Apache JXPath component to process XPath event selectors. If Java 2 security is enabled, you must modify your policy file to include an entry that allows read access to the `jxpath.properties` file:

```
permission java.io.FilePermission
  "${was.install.root}${/}java${/}jre${/}lib${/}jxpath.properties",
  "read";
```

Related tasks

"Writing event selectors" on page 61

Implementing a filter plug-in

If you want to use your own filtering engine as an emitter filter, you can implement a custom filter plug-in by following these steps:

1. Develop your filter plug-in as a Java class that implements the com.ibm.events.filter.Filter interface. This interface defines the following methods:

   isEventEnabled(CommonBaseEvent) method
   Returns a boolean value indicating whether the specified event passes the filter criteria. Each time an event is submitted to an emitter, the emitter calls this method, passing the submitted event. If the return

value is `true`, the emitter sends the event to the event server for persistence and distribution. If the return value is `false`, the emitter discards the event.

getMetaData method
Returns information about the filter plug-in, such as the provider name and the version number.

close method
Frees all of the resources used by the filter plug-in. This method is called when the close method of an emitter is called.

2. Develop a filter factory class that implements the interface com.ibm.events.filter.FilterFactory. This interface defines a single getFilter method, which returns an instance of your filter class (an implementation of the Filter interface).
3. Bind an instance of your filter factory into a Java Naming and Directory Interface (JNDI) namespace. During initialization, an emitter performs a JNDI lookup to access the filter factory.
4. In the WebSphere Application Server administrative console, modify your emitter factory profile or create a new profile. In the **Filter Factory JNDI Name** field, specify the JNDI name of your FilterFactory implementation. For more information about emitter factory profiles, see the online help for the administrative console.

When you create an emitter using the emitter factory profile that specifies your filter factory, the new emitter uses an instance of your filter implementation. You can now send events using the standard emitter interfaces, and your filter plug-in is used.

Chapter 7. Developing an event consumer

An *event consumer* is any application that receives events from the event server. This might be an application that receives asynchronous event notifications, or it might be an application that queries and processes historical event data from the persistent data store. The event consumer receives events in the form of Java objects and then uses the CommonBaseEvent interface to retrieve event property data, or convert the event to another supported format (such as XML) for forwarding to another application.

1. Receive an event.
   - Use the Java Messaging Service (JMS) interface to subscribe to a queue or topic, and receive events asynchronously as JMS messages. This is the most efficient approach for an event consumer that needs to receive events as they arrive at the event server. You can implement the event consumer as a standard Java class or as a Message-Driven Bean (MDB). See "Developing an event consumer as a message-driven bean (MDB)" on page 52 and "Developing a non-MDB event consumer" on page 54.
   - Use the Event Access interface to query historical events from the persistent data store and retrieve the requested events synchronously. This is useful for startup processing. By querying the data store for historical events, an event consumer can determine current state information before beginning to receive new events through JMS. In addition to receiving events, an event consumer can also purge old events from the data store. See "Querying events from the event server" on page 55.
2. Write event selectors to define event groups, specify filter criteria, and query the event server. See "Writing event selectors" on page 61.
3. Implement a data store plug-in for the persistent storage of events. See "Implementing a data store plug-in" on page 64.

Java Messaging Service interface and event consumers

By using the Java Messaging Service (JMS) interface, you can implement your event consumer using standard Java tools and programming models, and you can avoid the performance disadvantages of directly querying the event data store. Instead of interacting with the Common Event Infrastructure directly, your event consumer subscribes to JMS destinations (queues and topics) and receives events in the form of JMS messages.

The Common Event Infrastructure organizes events in event groups, which are logical collections of events defined in the Common Event Infrastructure configuration. A particular event consumer typically needs to receive only events from specific event groups.

The configuration profile for each event group associates that event group with one or more JMS destinations through which notifications related to that event group are distributed. The relationships between event groups and JMS destinations are as follows:

- An event group can be associated with multiple queues.
- An event group can be associated with only one topic. (Multiple event consumers can subscribe to the same topic, so publishing the same event group to more than one topic is redundant.)

- A JMS destination (queue or topic) should typically be associated with only one event group.

To receive messages from an event group, a JMS consumer subscribes to the appropriate destination. Each event is then delivered in the form of a JMS message containing an event notification. This notification can then be converted into a CommonBaseEvent instance.

In addition to the standard JMS interfaces, a JMS event consumer interacts with a facility called the notification helper. The notification helper translates between Common Event Infrastructure entities (events and event groups) and equivalent JMS entities (messages and destinations). The notification helper provides the following functions:

- It can identify the JMS topic or queues associated with a specified event group. Your event consumer can then use the appropriate destination to create subscriptions.
- It can convert a JMS message notification into a CommonBaseEvent instance.
- It can filter events at the consumer. Each notification helper can be associated with an event selector that specifyies which events should be returned to consumers. When a consumer uses the notification helper to convert an event notification into an event instance, the event instance is returned only if it matches the specified event selector.

  The notification helper uses the Apache JXPath component to process XPath event selectors. If Java 2 security is enabled, you must modify your policy file to include an entry allowing read access to the `jxpath.properties` file:

```
permission java.io.FilePermission
  "${was.install.root}${/}java${/}jre${/}lib${/}jxpath.properties",
  "read";
```

Related tasks

"Developing an event consumer as a message-driven bean (MDB)"

"Developing a non-MDB event consumer" on page 54

Developing an event consumer as a message-driven bean (MDB)

A J2EE event consumer is implemented as a message-driven bean, which is associated with a JMS destination and a connection factory at deployment time. To receive events, follow these steps:

1. Obtain a notification helper.

   A Java Message Service (JMS) event consumer uses a notification helper to identify JMS destinations associated with an event group, to convert received JMS messages into CommonBaseEvent instances, and to filter received events. To obtain a notification helper, use a notification helper factory, which is a NotificationHelperFactory instance that is bound into a Java Naming and Directory Interface (JNDI) namespace. The following code fragment uses a notification helper factory to obtain a notification helper.

```
// Get notification helper factory from JNDI
InitialContext context = new InitialContext();
Object notificationHelperFactoryObject =
  context.lookup("com/ibm/events/NotificationHelperFactory");
NotificationHelperFactory nhFactory = (NotificationHelperFactory)
  PortableRemoteObject.narrow(notificationHelperFactoryObject,
                              NotificationHelperFactory.class);
```

```
// Create notification helper
NotificationHelper notificationHelper =
  nhFactory.getNotificationHelper();
```

2. Optional: Specify the event selector.

   If you want to filter received events, you can use the setEventSelector method to set an event selector on the notification helper. Your event consumer can then use the notification helper to check received events against the event selector. The following code fragment sets an event selector that specifies events with a severity greater than 30 (warning).

   ```
   notificationHelper.setEventSelector("CommonBaseEvent[@severity > 30]");
   ```

3. Convert received messages into CommonBaseEvent instances.

   In the onMessage method of your listener, you can use the notification helper to convert each received JMS message into an event represented by a CommonBaseEvent instance. To do this, use the getCreatedEvent(Message) method of the NotificationHelper interface.

   Each message received by an event consumer has a property called the notification type. This is an integer with a value of one of the notification type constants that are defined by the com.ibm.events.notification.NotificationHelper interface. Currently, the only supported notification type is CREATE_EVENT_NOTIFICATION_TYPE, which indicates a notification of a new event. However, additional notification types might be added in future releases, so an event consumer should generally check this field using the NotificationHelper.getNotificationType method before processing received notifications.

   ```
   public void onMessage(Message msg) {
     int msgType = notificationHelper.getNotificationType(msg);
     if(msgType == NotificationHelper.CREATE_EVENT_NOTIFICATION_TYPE)
     {
       CommonBaseEvent event = notificationHelper.getCreatedEvent(msg);
       ...
     }
   ```

   If the received event does not match the event selector specified on the notification helper, the returned value is null.

4. Process the event.

   Your consumer can then process the event as appropriate.

   ```
   if (event != null) {
        // Process the event
        .....
        }
   ```

In its deployment descriptor, a message-driven bean must be associated with a listener port, which specifies a JMS destination and a connection factory. You must create a listener port for your event consumer before you deploy the MDB. This listener port must specify the destination and the connection factory associated with the event group from which you want to receive events. These parameters are defined in the event group profile.

Note: Do not use the CommonEventInfrastructure_ListenerPort listener port when you deploy your MDB. This listener port is used by the event server and is not intended for use by event consumers.

Developing a non-MDB event consumer

To write an event consumer that is not a message-driven bean, follow these steps:

1. Obtain a notification helper. A Java Message Service (JMS) event consumer uses a notification helper to identify JMS destinations associated with an event group, to convert received JMS messages into CommonBaseEvent instances, and to perform filtering of received events. To obtain a notification helper, use a notification helper factory, which is a NotificationHelperFactory instance that is bound into a Java Naming and Directory Interface (JNDI) namespace. The following code fragment uses a notification helper factory to obtain a notification helper.

   ```
   // Get notification helper factory from JNDI
   InitialContext context = new InitialContext();
   Object notificationHelperFactoryObject =
     context.lookup("com/ibm/events/NotificationHelperFactory");
   NotificationHelperFactory nhFactory = (NotificationHelperFactory)
     PortableRemoteObject.narrow(notificationHelperFactoryObject,
                                 NotificationHelperFactory.class);

   // Create notification helper
   NotificationHelper notificationHelper =
     nhFactory.getNotificationHelper();
   ```

2. Optional: Specify the event selector. If you want to filter received events, you can use the setEventSelector method to set an event selector on the notification helper. If you specify an event selector, the notification helper returns only events that match the event selector. The following code fragment sets an event selector that specifies events with a severity greater than 30 (warning).

   ```
   notificationHelper.setEventSelector("CommonBaseEvent[@severity > 30]");
   ```

3. Use the notification helper to find the JMS destination to subscribe to.

   Each event group can be associated with a single JMS topic and any number of JMS queues. You can query the notification helper to find out what destinations are associated with a particular event group. To find the topic associated with an event group, use the getJmsTopic(String) method of the NotificationHelper interface, specifying the name of the event group:

   ```
   MessagePort msgPort = notificationHelper.getJmsTopic("critical_events");
   ```

   To find the queues associated with an event group, use the getJmsQueues(String) method:

   ```
   MessagePort[] msgPorts = notificationHelper.getJmsQueues("critical_events");
   ```

   The returned object is either a single MessagePort object representing a JMS topic or an array of MessagePort objects representing JMS queues. A MessagePort instance is a wrapper object containing the JNDI names of the destination and its connection factory.

4. Connect to the destination. Use the getter methods of the MessagePort interface to retrieve the JNDI names of the destination and the connection factory. You can then use the standard JMS interfaces to connect to the destination. The following code fragment subscribes to a JMS topic:

   ```
   String connectionFactoryName = msgPort.getConnectionFactoryJndiName();
   String destinationName = msgPort.getDestinationJndiName();

   // create connection and session
   ConnectionFactory connectionFactory =
        (ConnectionFactory) context.lookup(connectionFactoryName);
   Connection connection = connectionFactory.createConnection();
   Session session = connection.createSesion(false,
   ```

```
                                                    CLIENT_ACKNOWLEDGE);

// Create consumer and register listener
Topic topic = (Topic) context.lookup(destinationName);
MessageConsumer consumer = session.createConsumer(topic);
consumer.setMessageListener(this);
connection.start();
```

5. Convert received event notification messages into CommonBaseEvent instances.

   In the onMessage method of your listener, you can use the notification helper to convert each received JMS message into an event that is represented by a CommonBaseEvent instance. To do this, use the getCreatedEvent(Message) method of the NotificationHelper interface.

   Each message received by an event consumer has a property called the notification type. This is an integer, the value of which is one of the notification type constants defined by the com.ibm.events.notification.NotificationHelper interface. Currently, the only supported notification type is CREATE_EVENT_NOTIFICATION_TYPE, which indicates a notification of a new event. However, additional notification types might be added in future releases, so an event consumer should generally check this field using the NotificationHelper.getNotificationType method before processing received notifications.

   ```
   public void onMessage(Message msg) {
     int msgType = notificationHelper.getNotificationType(msg);
     if(msgType == NotificationHelper.CREATE_EVENT_NOTIFICATION_TYPE)
     {
       CommonBaseEvent event = notificationHelper.getCreatedEvent(msg);
       ...
     }
   ```

   If the received event does not match the event selector specified on the notification helper, the returned value is null.

6. Process the event. Your consumer can then process the event as appropriate.

   ```
   if (event != null) {
        // Process the event
        .....
        }
   ```

Querying events from the event server

An event consumer can synchronously retrieve historical events from the persistent data store by querying the event server. The persistent data store is implemented as a separate component called a *data store plug-in*. The Common Event Infrastructure includes a default data store plug-in, which supports event queries based on a subset of XPath syntax. If you want to use a different data store, you can implement your own data store plug-in.

To query the event server, use the event access interface.

1. Create an event access bean. This bean is a Java 2 Platform, Enterprise Edition (J2EE) stateless session bean. The bean interface provides methods for querying the event server. An event consumer uses an instance of the event access bean for all synchronous event queries. See “Creating an event access bean” on page 56.
2. Query events. You can query events in the following ways:
   - Specify a global instance identifier to retrieve a specific single event. See “Querying events by global instance identifier” on page 56.

- Specify an event group to retrieve events associated with that event group. You can optionally refine the query by specifying an additional event selector. This action retrieves only those events that match both the event group and the event selector. See "Querying events by event group" on page 57.
- Specify a known event and an association type to retrieve events that are associated with the known event. See "Querying events by association type" on page 59.
- Query and purge events. See "Purging events from the data store" on page 60.

Creating an event access bean

The event access interface is implemented as a stateless session bean using the Enterprise JavaBeans architecture. To query the event server using the event access interface, an event source must first create an instance of the event access session bean. The event access bean can be either local or remote.

To create an instance of the event access session bean, use the appropriate home interface: com.ibm.events.access.EventAccessHome or com.ibm.events.access.EventAccessLocalHome.

```
// use home interface to create remote event access bean
InitialContext context = new InitialContext();
Object eventAccessHomeObj = context.lookup("ejb/com/ibm/events/access/EventAccess");
EventAccessHome eventAccessHome = (EventAccessHome)
  PortableRemoteObject.narrow(eventAccessHomeObj,
                              EventAccessHome.class);
eventAccess = (EventAccess) eventAccessHome.create();
```

Querying events by global instance identifier

The Common Base Event specification defines a globalInstanceId event property that can be used as a primary key for event identification. The content of this property is a globally unique identifier that is generated either by the application or by the emitter. Although the Common Base Event specification defines the globalInstanceId property as optional, the event emitter automatically assigns an identifier to any event that does not already have an identifier.

You can retrieve a specific single event from the event server by querying with the globalInstanceId property of the event that you want to retrieve. This query can be useful for testing purposes (to confirm that events are stored in the event database), or to retrieve an event associated with one that was received previously.

To query an event by the global instance identifier, use the queryEventByGlobalInstanceId method of the event access bean.

1. Optional: Create an event access bean.
2. Call the queryEventByGlobalInstanceId(String) method of the EventAccess bean, specifying the global instance identifier of the event that you want to retrieve.

   ```
   CommonBaseEvent event = eventAccess.queryEventByGlobalInstanceId(eventId);
   ```

   The returned object is the event with the specified global instance identifier. If there is no matching event in the persistent data store, the returned object is null.

Related tasks

"Creating an event access bean" on page 56
"Sending events" on page 43

Querying events by event group

You can associate an event with one or more *event groups*. An event group is a logical grouping of events that match a particular event selector. Event groups are defined in the event infrastructure configuration. For more information about event groups, see "Default configuration" on page 9.

You can use the event access interface to retrieve events that belong to a specified event group. You can restrict the query results by specifying an additional event selector. You can also query events without retrieving them.

You can query event groups in the following ways:

- Query a limited number of events from an event group.
- Query all events from an event group.
- Query whether an event exists.

Querying a limited number of events from an event group

To query a limited number of events from an event group, use the queryEventsByEventGroup(String, String, boolean, int) method of the EventAccess bean.

1. Optional: Create an event access bean.
2. Call the EventAccess.queryEventsByEventGroup(String, String, boolean, int) method.

```
CommonBaseEvent[] events = eventAccess.queryEventsByEventGroup(eventGroup,
                                                               eventSelector,
                                                               ascendingOrder,
                                                               maxEvents);
```

The parameters of this method are as follows:

eventGroup
A string that contains the name of the event group that you want to query for events. This name must be the name of an existing event group defined in the event infrastructure configuration.

eventSelector
A string that contains an optional event selector that refines the query. The query returns events that match both the specified event group and the additional event selector. An event selector is specified in the form of an XPath expression. If you do not want to specify an additional event selector, this parameter can be null.

ascendingOrder
A boolean value that specifies whether the returned events are sorted in ascending or descending order according to the value of the creationTime property. If this parameter is *true*, the events are sorted in ascending (chronological) order. If this parameter is *false*, the events are sorted in descending (reverse chronological) order.

maxEvents
An integer that specifies the maximum number of events that you want returned.

The returned object is an array that contains the events from the specified event group.

If the number of matching events exceeds the query threshold that is defined in the data store profile, a QueryThresholdExceededException exception is thrown. The default query threshold is `100 000`.

The following code fragment returns all of the events that belong to an event group called *critical_hosts* with a severity greater than 30 (warning). No more than 5000 matching events should be returned:

```
CommonBaseEvent[] events =
        eventAccess.queryByEventGroup("critical_hosts",
                                      "CommonBaseEvent[@severity > 30]",
                                      true,
                                      5000);
```

Related tasks

"Creating an event access bean" on page 56

"Writing event selectors" on page 61

Querying all events from an event group

To query all events from an event group, use the queryEventsByEventGroup(String, String, boolean) method of the EventAccess bean.

1. Optional: Create an event access bean.
2. Call the EventAccess.queryEventsByEventGroup(String, String, boolean) method.

   ```
   CommonBaseEvent[] events = eventAccess.queryEventsByEventGroup(eventGroup,
                                                                  eventSelector,
                                                                  ascendingOrder);
   ```

   The parameters of this method are as follows:

   eventGroup
   A string that contains the name of the event group that you want to query for events. This name must be the name of an existing event group defined in the event infrastructure configuration.

   eventSelector
   A string that contains an optional event selector that further refines the query. The query returns events that match both the specified event group and the additional event selector. An event selector is specified in the form of an XPath expression. If you do not want to specify an additional event selector, this parameter can be null.

   ascendingOrder
   A boolean value that specifies whether the returned events are sorted in ascending or descending order according to the value of the creationTime property. If this parameter is *true*, the events are sorted in ascending (chronological) order. If this parameter is *false*, the events are sorted in descending (reverse chronological) order.

   The returned object is an array that contains the events from the specified event group.

   If the number of matching events exceeds the query threshold that is defined in the data store profile, a QueryThresholdExceededException exception is thrown. The default query threshold is `100 000`.

The following code fragment returns all of the events that belong to an event group called *critical_hosts* with a severity greater than 30 (warning):

```
CommonBaseEvent[] events =
        eventAccess.queryByEventGroup("critical_hosts",
                                      "CommonBaseEvent[@severity > 30]",
                                      true);
```

Related tasks

"Creating an event access bean" on page 56

"Writing event selectors" on page 61

Querying the existence of events in an event group

In some situations, you might want to find out whether any events exist in a particular event group without actually retrieving the events. To do this, use the eventExists method of the event access bean.

1. Optional: Create an event access bean.
2. Call the eventExists(String, String) method of the EventAccess bean.

   ```
   boolean hasEvents = eventAccess.eventExists(eventGroup,
                                               eventSelector);
   ```

   The parameters of this method are as follows:

   eventGroup
   : A string that contains the name of the event group that you want to check for events. This name must be the name of an existing event group defined in the event infrastructure configuration.

   eventSelector
   : A string that contains an optional event selector that refines the query. The query checks for events that match both the specified event group and the additional event selector. An event selector is specified in the form of an XPath expression. If you do not want to specify an additional event selector, this parameter can be null.

The returned boolean object equals `true` if any events exist that match the specified event group and event selector, `false` if none exist.

The following code fragment checks for the existence of any events in an event group called *critical_hosts* and retrieves any events that exist.

```
if (eventAccess.eventExists("critical_hosts",null)) {
   CommonBaseEvent[] events =
     eventAccess.queryByEventGroup("critical_hosts",
                                   null,
                                   true);
   }
```

Related tasks

"Creating an event access bean" on page 56

"Writing event selectors" on page 61

Querying events by association type

The Common Base Event specification defines properties that establish relationships between events. The associatedEvents property is a complex element that contains one or more subelements of the AssociatedEvent type, each representing an associated event. Each AssociatedEvent element, contains

subelements that identify the type of association and the application that established the association. Examples of association types might include CausedBy or Correlated.

By specifying the global instance identifier of a known event and a type of association, you can retrieve events that satisfy the specified association. To query events by association type, use the EventAccess.queryEventsByAssocation(String, String) method.

1. Optional: Create an event access bean.
2. Call the EventAccess.queryEventsByAssociation(String, String) method.

```
CommonBaseEvent[] events = eventAccess.queryEventsByAssocation(associationType,
                                                               eventId);
```

The parameters of this method are as follows:

associationType
The type of association. This is the name of an association type specified by the associationEngineInfo property.

eventId
The global instance identifier of a known event.

The returned object is an array that contains the events that satisfy the specified type of association with the known event. Only events that are still in the event database at the time of the query are returned (an associated event might be purged from the database).

The following code fragment returns all of the events from the event database that have a CausedBy association with a known event:

```
String eventId = causeEvent.getGlobalInstanceId();
CommonBaseEvent[] resultEvents = eventAccess.queryEventsByAssociation("CausedBy",
                                                                      eventId);
```

Related tasks

"Creating an event access bean" on page 56

Purging events from the data store

An event consumer or an administrative tool can purge events from the data store using the event access interface. You can purge all of the events from the data store, or you can limit the purge to event groups, event selectors, or both.
To purge events from the data store, use the purgeEvents method of the event access bean.

```
int purged = eventAccess.purgeEvents(eventGroup,
                                     eventSelector,
                                     transactionSize);
```

The parameters are as follows:

eventGroup
A string that contains the name of the event group that includes the events you want to purge. This name must be the name of an existing event group that is defined in the event infrastructure configuration. If you do not want to specify an event group, this parameter can be null.

eventSelector
A string that contains an optional event selector that identifies the events to purge. An event selector is specified in the form of an XPath expression. If you do not want to specify an event selector, this parameter can be null.

transactionSize
A nonzero integer that specifies the number of events that you want to purge in a single database transaction. In most cases, you can use the DEFAULT_PURGE_TRANSACTION_SIZE constant, which is defined by the EventAccess interface.

The purgeEvents method purges all of the events that match all of the criteria that you specify. If the *eventGroup* and *eventSelector* parameters are both null, all of the events in the data store are purged. Events that arrive after the purge operation starts are not purged. The returned value is an integer that specifies how many events were purged.

If the value of the *transactionSize* parameter exceeds the maximum purge transaction size defined in the data store profile, a PurgeThresholdExceededException exception is thrown and no events are purged. The default maximum purge transaction size is `100 000`.

Related tasks

"Writing event selectors"

Writing event selectors

An event selector is a regular expression that defines a set of events based on the property data (attributes or subelements) of these events. For example, an event selector might specify all of the events from a particular host with a severity that is greater than 30 (warning). Use event selectors to define event groups, specify filter criteria, and query the event server.

Because the Common Base Event specification is based on XML, event selectors are written using a subset of the XPath syntax. The specific syntax you can use for an event selector depends on how the event selector is to be used, as summarized by the following table.

Event selector purpose	Syntax
Event group definition	Limited to the XPath subset supported by the default data store plug-in
Event query and purge through the event access interface	Limited to the XPath subset supported by the default data store plug-in
Emitter filter configuration	Any valid XPath
Subscription through the NotificationHelper interface	Any valid XPath

The default data store plug-in uses a subset of the XPath syntax. However, if you are using a different data store plug-in, it might support a different subset of the XPath syntax. The event selectors you write for event group definitions and for the event access interface must use the syntax that is supported by your data store plug-in.

- Write XPath event selectors.
- Write event selectors for the default data store plug-in.

Writing XPath event selectors

XPath is a standard language that is used to identify parts of an XML document. For more information, see the XPath specification at http://www.w3.org/TR/xpath.

A simple XPath event selector that specifies an attribute value takes the following form:

```
CommonBaseEvent[@attribute = value]
```

The *value* can be either a numeric value or a string enclosed in single or double quotation marks.

You can also specify an attribute of a subelement:

```
CommonBaseEvent[/subelement/@attribute = value]
```

When using XPath operators, remember the following rules:

- When used to compare XML dateTime values, the comparison operators perform logical comparisons that recognize time zone differences.
- Logical operators and function names must be specified using all lowercase letters (for example, `and` rather than `AND`).
- Operators must be separated with white space from the surrounding attribute names and values (`@severity > 30` rather than `@severity>30`).
- Parentheses can be used to change operator precedence.

The following examples are valid XPath event selectors.

`CommonBaseEvent[@extensionName = 'ApplicationStarted']`	All events with the extensionName attribute `ApplicationStarted`
`CommonBaseEvent[sourceComponentId/ @location = "server1"]`	All events containing a sourceComponentId element with the location attribute `server1`
`CommonBaseEvent[@severity]`	All events with a severity attribute, regardless of its value
`CommonBaseEvent[@creationTime < '2003-12-10T12:00:00-05:00' and @severity > 30]`	All events created before noon EST on 10 December 2003 and with severity greater than 30 (warning):
`CommonBaseEvent[contains(@msg, 'disk full')]`	All events with the phrase `disk full` occurring within the msg attribute
`CommonBaseEvent[(@severity = 30 or @severity = 50) and @priority = 100]`	All events with a severity attribute equal to 30 or 50, and a priority attribute equal to 100.

Related tasks

"Writing event selectors for the default data store plug-in"

Writing event selectors for the default data store plug-in

If your event selector might be used to define an event group or to query the persistent data store, it is subject to the restrictions of the default data store plug-in. These restrictions are as follows:

- An event property can be specified only on the left side of an operator or an XPath function. The value on the right side of an operator must be a literal value. The following example is not a valid event selector:

```
CommonBaseEvent[30 < @priority and
                contains('this message', @msg)]
```

The example can be rewritten as follows:

```
CommonBaseEvent[@priority > 30 and
                contains(@msg, 'this message')]
```

- Only the following XPath functions are supported:
  - contains
  - starts-with
  - false
  - true
  - not
- The union operator (|) is not supported.
- An event selector must take the following form:

  ```
  CommonBaseEvent[predicate_expression]
  ```

  Only a single predicate expression can be associated with the CommonBaseEvent element. Stacked predicates are not supported, for example:

  ```
  CommonBaseEvent[@extensionName = "server_down"][@severity = 10]
  ```

- A predicate can be only be associated with the last step of a location path. The following example is not a valid event selector:

  ```
  CommonBaseEvent[contextDataElement[@contextValue = "myContextValue"]
                  /@contextId = "myContextId"]
  ```

  The example can be rewritten as follows:

  ```
  CommonBaseEvent[contextDataElement[@contextValue = "myContextValue"
                  and @contextId = "myContextId"]]
  ```

- If an event selector refers to properties of extended data elements that are at different levels of the XML containment hierarchy, these elements must be grouped together by level. The following example is not a valid event selector, because the references to the type and value attributes (both top-level) of *extendedDataElements* are separated:

  ```
  CommonBaseEvent[extendedDataElements[@type = 'int' and
                                       children/@type = 'intArray' and
                                       children/@name = 'myName' and
                                       @value = 10]]
  ```

  The example can be rewritten as follows, grouping the top-level and second-level attributes together:

  ```
  CommonBaseEvent[extendedDataElements[@type = 'int' and
                                       @value = 10 and
                                       children/@type = 'intArray' and
                                       children/@type = 'myName']]
  ```

- Node indexes are not supported, for example:

  ```
  CommonBaseEvent[extendedDataElements[1]]
  ```

- Wildcard characters are not supported, for example:

  ```
  CommonBaseEvent[extendedDataElements/*/children/values = "text"]
  ```

- When referring to the values property of an extended data element, you must specify the value and the type of the property:

  ```
  CommonBaseEvent[extendedDataElements[values = "myVal"
    and @type = "string"]]
  ```

  You can specify the type for multiple comparisons within a compound expression by grouping the comparisons with parentheses:

  ```
  CommonBaseEvent[extendedDataElements[(values = "myVal" or
                                        values = "yourVal") and
                                        @type = "string"]]
  ```

In this example, the type expression applies to both parts of the compound expression in parentheses. You cannot override this restriction by specifying a different type expression inside the parentheses.

You can group multiple related types by using the `starts-with` or `contains` functions. For example, the following expression matches a property with either the string or the stringArray type:

```
CommonBaseEvent[extendedDataElements[values = "myVal" and
                                     starts-with(@type, 'string')]]
```

Related tasks

"Writing XPath event selectors" on page 62

Implementing a data store plug-in

To use your own data store for the persistent storage of events, implement a custom data store plug-in by following these steps:

1. Develop your data store plug-in as an enterprise bean with the provided local interface. Your data store plug-in must implement the com.ibm.events.datastore.DataStoreLocal interface. The DataStoreLocal interface defines the following methods. Refer to the Javadoc API documentation for more information.

   createEvent(CommonBaseEvent) method
   Stores a new event in the data store.

   eventExists(String) method
   Returns a boolean that indicates whether any events currently in the data store match the specified event selector.

   purgeEvents(String[]) method
   Deletes events that match the specified global instance identifiers.

   queryEventByGlobalInstanceId(String) method
   Returns the event with the global instance identifier that matches the specified value (or null if no matching event is found).

   queryEvents(String, boolean) method
   Returns an array of events that match the specified event selector.

   queryEvents(String, boolean, int) method
   Returns an array of events that match the specified event selector, limiting the array to the specified size.

   queryEventsByAssociation(String, String) method
   Returns an array of events that satisfy the specified relationship to a known event.

   queryGlobalInstanceIds(String, int) method
   Returns an array of global instance identifiers for events that match a specified event selector, limiting the array to the specified size.

   getMetaData method
   Returns metadata that describes the data store plug-in, including the version of the Common Base Event specification it supports.

   A data store plug-in must also satisfy the following requirements:

   - It must use XPath syntax, or a subset of XPath syntax, for specifying event selectors.
   - It must store all of the data associated with each received event.

- Its query methods must return event objects that are identical to those originally stored.

2. Deploy your data store plug-in in WebSphere Application Server. See the WebSphere Application Server documentation for more information about how to deploy an application.
3. In the administrative console, modify the default event server profile. In the **Data Store JNDI Name** field, specify the Java Naming and Directory Interface (JNDI) name of your data store plug-in. For more information about the event server profile, see the online help for the administrative console.

When you start the Common Event Infrastructure server, the event server uses the specified JNDI name to access the local home interface of the data store enterprise bean. It then uses the local home interface to create an instance of the data store plug-in bean.

Chapter 8. Developing an event catalog application

The *event catalog* is a repository of event metadata. This metadata consists of event definitions, which describe classes of events and their allowed content. (This is distinct from the event instance metadata that you can access using the Eclipse Modeling Framework interfaces that is described in "Accessing event instance metadata" on page 39.) Applications can use the event catalog to manage their enterprise-specific event definitions. However, these applications must implement validation logic to ensure that events conform to these definitions.

Events defined according to the Common Base Event specification can be categorized into event classes based upon the extension name (the value of the extensionName attribute). Using the event catalog, you can define the permitted content of a particular class of event by specifying the extended data elements that events of that class can contain, and the permitted values for other Common Base Event properties. An event definition defines constraints on event content that extend those of the Common Base Event specification.

Use the Event Catalog interfaces to create, delete, and query event definitions. You cannot modify an event definition. You can also list existing event definitions in a readable format, and import and export event definitions in XML format.

Event definitions

Event definitions are defined hierarchically and inherit the definitions of the parent definitions. A single root event definition, `event`, defines the basic requirements of any event that conforms to the Common Base Event specification. All other event definitions inherit from this root definition. By default, this root event definition is automatically installed in the event catalog, along with event definitions for event catalog notification events.

Currently, event definitions do not support all of the constraints that are required to fully describe the Common Base Event specification, for example, the requirement that the globalInstanceId property must begin with an alphabetic character. Therefore, an event might conform to the root event definition and still not pass validation by the event emitter.

An event definition contains several kinds of information:

Name The name of the event definition, which is the same as the extension name of the events described by the definition. All events with a particular extension name share the same event definition.

Parent The name of the parent event definition. Any event definition (with the

exception of the root definition `event`) has a parent event definition from which it inherits property descriptions and extended data element descriptions (although some aspects of the inherited data can be overriden). The parent can be any valid event definition that exists in the event catalog.

Property descriptions
Descriptions of the permitted common base event properties for the event definition. A property description can describe any property defined in the Common Base Event specification as a simple type, including properties of complex subelements.

Extended data element descriptions
Descriptions of the permitted extended data elements for the event definition. An extended data element description defines the name and type of the extended data element. The description can also define default values, how many of the extended data element are allowed, and descriptions of child extended data elements.

Represented as an XML document, an event definition takes the following general form:

```
<eventDefinition name="eventDefinitionName"
                 parent="parentEventDefinitionName">
     <property name="propertyName" ... />
     <extendedDataElement name="extendedDataElementName"
                          type="type" ... />
</eventDefinition>
```

Related concepts

"Change notification" on page 72

Property descriptions

A property description describes a property that an event can contain. This can be any property that is defined by the Common Base Event specification as a simple type. A property description cannot describe a complex property, such as msgDataElement, but it can describe a simple property that is a child of a complex property. An event definition can contain any number of property descriptions (including none).

A property description includes the following fields:

name The name of the property. This must be the name of an attribute of the CommonBaseEvent element, or an attribute of a complex subelement of the CommonBaseEvent element. Some examples are severity, priority, and globalInstanceId.

path An XPath location path specifying the path to the property, if the property is not an attribute of the CommonBaseEvent element. The path identifies the parent property of the property that is described. These are examples:

- To describe a property of the CommonBaseEvent element, such as severity, do not specify a path. A null path specifies a top-level property.
- To describe a property of the msgDataElement element, which is a complex property of the CommonBaseEvent element, specify the path `msgDataElement`.
- To describe a property of the msgHelp element, which is a complex property of msgDataElement, specify the path `msgDataElement/msgHelp`.

The path can also describe a specific instance of a repeated property. For example, if an event definition describes several contextDataElements properties, you might specify a property called businessContext, and use the path `contextDataElements[@name='businessContext']`.

defaultValue
The default value of the property. The default value represents the value that is used during content completion for an event that is missing a required property. Therefore, it is meaningful for a property description to be required and to define a default value. This field is optional.

required
A boolean value that specifies whether the property is required. If this field is equal to `true`, the property is required. This field is optional. If a value is not specified, the property is assumed to be optional.

permittedValue
A permitted value for the property. If an event definition allows only certain values for a property, each one is represented by a permittedValue field in the property description. A property description can include any number of permitted values. This field is optional and must not be specified if you specify values for the minValue or maxValue fields.

minValue
maxValue
The minimum and maximum permitted values for the property. If an event definition allows a range of values for a property, these fields define the lower and upper limits of that range. If you specify only minValue, the permitted range has no upper limit. Similarly, if you specify only maxValue, the permitted range has no lower limit. These fields are optional. Do not specify values for these fields if you specify values for the permittedValue fields.

Extended data element descriptions

An extended data element description describes an extended data element that an event of a particular event class can contain. An event definition can contain any number of extended data element descriptions (including none).

An extended data element description includes the following fields:

name The name of the extended data element. This defines the value of the name attribute of the element.

type The data type of the extended data element. This defines the value of the type attribute of the element. This must be one of the following supported data types:
- noValue
- byte
- short
- int
- long
- float
- double
- string
- dateTime
- boolean
- byteArray
- shortArray

- intArray
- longArray
- floatArray
- doubleArray
- stringArray
- dateTimeArray
- booleanArray
- hexBinary

defaultValue
The default value of the extended data element, or multiple default values if the type is an array. The default value represents the value that is used during content completion for an event that is missing a required extended data element. This field is optional.

minOccurs
The minimum number of instances of the extended data element that must appear. This field is optional. The default value is 1.

maxOccurs
The maximum number of instances of the extended data element that can appear. This field is optional. The default value is 1.

The current Common Base Event specification allows only one extended data element with a given name at each level of the event containment hierarchy. This restriction will not be included in future versions of the specification and is not enforced by the Common Event Infrastructure.

Inheritance

By default, an event definition inherits the property descriptions and extended data element descriptions of the parent definition. However, a child event definition can override these inherited descriptions, subject to certain restrictions. When you add an event definition to the event catalog, the catalog verifies that the new event definition does not violate the rules that govern inheritance. If an event does not adhere to the rules, an InheritanceNotValidException exception is thrown. Similarly, if you replace an existing event definition that has descendants, the event catalog verifies the validity of the existing inheritance relationships and throws an InheritanceNotValidException exception if any of these relationships are no longer valid. In either case, the new event definition is not added to the catalog unless all of the inheritance relationships are valid.

An event definition can be either *unresolved* or *resolved*:

- An unresolved event definition includes only those property definitions and extended data element descriptions that are defined within the event definition.
- A resolved event definition includes the data in the unresolved event definition and the property definitions and extended data element descriptions it inherits.

Overriding inherited property descriptions

A child event definition inherits each property description from its parent without change unless it already has a locally defined property description of the same name and path (note that case is significant). If the child has a property description of the same name and path, the fields of the child description can override the fields of the parent description as follows:

Default value
The child can override the default value specified by the parent property description. If the child does not specify a default value, it inherits the value from the parent.

Required or optional
The child always overrides the parent. However, if the parent defines a property as required, the child must also specify that the property is required. An inherited required property cannot be redefined as optional.

Permitted values or minimum and maximum values
If the parent defines permitted values or minimum and maximum values, the child can override these by specifying either permitted values or minimum and maximum values. An event definition can contain only permitted values or minimum and maximum values, not both. For example:

- If the parent defines minimum and maximum values, but the child defines permitted values, the minimum and maximum values defined by the parent are ignored.
- If the parent defines permitted values, but the child defines minimum and maximum values, the permitted values defined by the parent are ignored.
- If the parent defines only a maximum value, but the child defines only a minimum value, the child inherits the maximum value defined by the parent.
- If the child does not specify permitted values or minimum and maximum values, the values specified by the parent are inherited.

Overriding inherited extended data element descriptions

A child event definition inherits each extended data element description from its parent without change unless it already has a locally defined extended data element description of the same name. If the child does have an extended data element description of the same name, the fields of the child description can override the fields of the parent description as follows:

Type The child must specify the same type as the parent.

Minimum occurrence
The child always overrides the parent.

Maximum occurrence
The child always overrides the parent.

Default values
The child can override the default values specified by the parent extended data element description. If the child does not specify default values, it inherits the values from the parent.

Default hexadecimal value
The child can override the default hexadecimal value specified by the parent extended data element description. If the child does not specify a default hexadecimal value, it inherits the value from the parent.

Nested extended data element description
The child can override a nested extended data element description by defining a nested description of the same name. If the child overrides an inherited nested description, the same rules apply to overriding the

individual fields. If the child does not specify a nested extended data element description of the same name, it inherits the nested description from the parent.

Change notification

Each time an event definition is added, removed, or replaced, the event catalog sends an event to the event server that indicates that this action happened. An event consumer can subscribe to these events to receive notification of changes in the event catalog. By default, the event catalog uses the default emitter factory to obtain an emitter for sending these events. You can change the emitter factory in the event catalog configuration.

The event catalog can send three classes of notification events, using the following extension names:
- cei_event_definition_added
- cei_event_definition_replaced
- cei_event_definition_removed

These three event classes inherit property descriptions from a common cei_event_definition parent class. Event definitions for all four event classes are automatically loaded into the event catalog during installation, with the default root event definition.

When an event definition is removed from the event catalog, any children or other descendants of that event definition are also removed. The event catalog sends a separate change notification event for each event definition that is removed.

Each change notification event contains the following properties:

Property	**Value**
version	`1.0.1`
globalInstanceId	A globally unique identifier for the event.
creationTime	Current date and time when the event is generated.
severity	10 (information)
priority	10 (low)
sourceComponentId	Identification of the event catalog component and event server host machine.
situation	Situation data, including one of the following values for the situation category: • CreateSituation (event definition added) • ConfigureSituation (event definition replaced) • DestroySituation (event definition removed)
extensionName	One of the following values: • cei_event_definition_added • cei_event_definition_replaced • cei_event_definition_removed
extendedDataElements	A single extended data element with one attribute, eventDefinitionName. This attribute is a string that specifies the name of the event definition that has been added, replaced, or removed.

Creating an event definition

An event definition is an instance of the com.ibm.events.catalog.EventDefinition class. To create an event definition, create an instance of this class and then populate it with property descriptions and extended data element descriptions. To create a new, empty event definition, create an instance of the EventDefinition class:

```
EventDefinition definition = new EventDefinition(name, parent);
```

The parameters of this constructor are as follows:

name
The name of the event definition. This is the value of the extensionName attribute for the events that you are describing.

parent
The name of the parent event definition. If you do not want your event definition to inherit any property descriptions or extended data element descriptions other than those required by the Common Base Event specification, set this parameter to `event`. If this parameter is null, the new event definition is defined as a root event definition. A root event definition can only be added to the catalog if it is empty, or if you intend to replace the current root event definition.

The returned object is a new unresolved event definition that contains no property descriptions or extended data element descriptions.

The following code fragment creates a new event definition called `insurance_claim_start_auto`, which is a child of the event definition `insurance_claim_start`:

```
EventDefinition definition = new EventDefinition("insurance_claim_start_auto",
                                                 "insurance_claim_start");
```

You can now populate the event definition with property descriptions and extended data element descriptions. After you create an event definition, you can add it to the event catalog.

Related tasks

"Adding property descriptions to an event definition"

"Adding extended data element descriptions to an event definition" on page 74

"Adding an event definition to the event catalog" on page 76

Adding property descriptions to an event definition

A property description is an instance of the com.ibm.events.catalog.PropertyDescription class. To add a property description to an event definition, you must create a property description and then set the values of its fields. You can then add the property description to the event definition.

1. To create a new property description, create an instance of the PropertyDescription class, specifying the name and path of the property.

   ```
   PropertyDescription propDesc = new PropertyDescription(name, path);
   ```

   The parameters of this constructor are as follows:

name
The name of the property. This must be the name of a simple property either of the CommonBaseEvent element or one of its children.

path
An XPath location path that specifies the path to the property. For a top-level property of the CommonBaseEvent element, such as severity or priority, *path* should be null.

The returned object is a new PropertyDescription object.

2. Populate the fields of the property description. The PropertyDescription class provides a setter method for each of the fields in a property description. Refer to the Javadoc API documentation for more information about these methods. For example, to specify that a property is required, set the *required* property to `true` using the setRequired(boolean) method:

   ```
   propDesc.setRequired(true);
   ```

3. Add the property description to the event definition using the EventDefinition.addPropertyDescription method.

   ```
   definition.addPropertyDescription(propDesc);
   ```

   If the event definition already includes another property description with the same name and path, a DescriptionExistsException exception is thrown.

The following code fragment creates a property description, populates it with data, and adds it to an event definition.

```
PropertyDescription propDesc = new PropertyDescription("severity",null);
propDesc.setRequired(true);
propDesc.setMinValue('30');

// definition is a valid event definition
definition.addPropertyDescription(propDesc);
```

Adding extended data element descriptions to an event definition

An extended data element description is an instance of the ExtendedDataElementDescription class. To add an extended data element description to an event definition, you must create an extended data element description and then set the values of its fields. You can also add nested (child) extended data element descriptions, which describe nested extended data elements. You can then add the extended data element description to the event definition.

1. To create an extended data element description, create an ExtendedDataElementDescription instance, specifying the name and type of the extended data element.

   ```
   ExtendedDataElementDescription edeDesc =
        new ExtendedDataElementDescription(name, type);
   ```

   The parameters of this constructor are as follows:

   name
   The name of the extended data element. This must be the value of the name property of the extended data element that you want to describe.

   type
   The data type of the extended data element. This must be one of the following integer constants that are defined by the org.eclipse.hyades.logging.events.cbe.ExtendedDataElement class:
   - TYPE_BOOLEAN_ARRAY_VALUE
   - TYPE_BOOLEAN_VALUE

- TYPE_BYTE_ARRAY_VALUE
- TYPE_BYTE_ARRAY
- TYPE_DATE_TIME_ARRAY_VALUE
- TYPE_DATE_TIME_VALUE
- TYPE_DOUBLE_ARRAY_VALUE
- TYPE_DOUBLE_VALUE
- TYPE_FLOAT_ARRAY_VALUE
- TYPE_FLOAT_VALUE
- TYPE_HEX_BINARY_VALUE
- TYPE_INT_ARRAY_VALUE
- TYPE_INT_VALUE
- TYPE_LONG_ARRAY_VALUE
- TYPE_LONG_VALUE
- TYPE_NO_VALUE_VALUE
- TYPE_SHORT_ARRAY_VALUE
- TYPE_SHORT_VALUE
- TYPE_STRING_ARRAY_VALUE
- TYPE_STRING_VALUE

The returned object is a new ExtendedDataElementDescription object.

2. Populate the fields of the extended data element description. The ExtendedDataElementDescription class provides a setter method for each of the fields in an extended data element description. Refer to the Javadoc API documentation for more information about these methods. For example, to specify that an extended data element must occur at least once, you can set the maxOccurs property to 4 using the setMaxOccurs(int) method:

   ```
   edeDesc.setMaxOccurs(4);
   ```

3. Optional: To add a child extended data element description, use the ExtendedDataElementDescription.addChild method.

   ```
   edeDesc.addChild(childEdeDesc);
   ```

   The childEdeDesc parameter must be a valid extended data element description.

4. Add the extended data element description to the event definition using the EventDefinition.addExtendedDataElementDescription method.

   ```
   definition.addExtendedDataElementDescription(edeDesc);
   ```

   If the event definition already includes another extended data element description with the same name and path, a DescriptionExistsException exception is thrown.

The following code fragment creates an extended data element description, populates it with data, and adds it to an event definition.

```
ExtendedDataElementDescription edeDesc =
     new ExtendedDataElementDescription("age", TYPE_SHORT_VALUE);
edeDesc.setMinOccurs(1);
edeDesc.setMaxOccurs(1);

// definition is a valid event definition
definition.addExtendedDataElementDescription(edeDesc);
```

Creating an event catalog bean

The event catalog is implemented as a stateless session bean using the Enterprise JavaBeans architecture. To access the event catalog, an event catalog application must first create an instance of the event catalog session bean.
Use the home interface to create an instance of the event catalog session bean.

```
//use home interface to create event catalog bean
InitialContext context = new InitialContext();
Object eventCatalogHomeObj =
     context.lookup("ejb/com/ibm/events/catalog/EventCatalog");
EventCatalogHome eventCatalogHome = (EventCatalogHome)
  PortableRemoteObject.narrow(eventCatalogHomeObj,
                              EventCatalogHome.class);
eventCatalog = (EventCatalog) eventCatalogHome.create();
```

Adding an event definition to the event catalog

After you have created a new event definition and populated it with property descriptions and extended data element descriptions, you can add it to the event catalog. After an event is added to the event catalog, the event definition cannot be modified, but it can be replaced.
To add an event definition to the event catalog, use the addEventDefinition method.

```
boolean result = eventCatalog.addEventDefinition(definition, replace)
```

The parameters of this method are as follows:

definition
The event definition you want to add. This must be a valid EventDefinition instance.

replace
A boolean value that indicates whether the specified event definition replaces an existing definition that has the same name.

If the *replace* parameter is `false`, the name of the specified event definition must not match the name of an existing event definition in the catalog. If the name exists, an EventDefinitionExistsException exception is thrown.

If the *replace* parameter is `true`, the new event definition replaces any existing event definition with the same name that is already in the catalog. However, to preserve the inheritance hierarchy, the new event definition must name the same parent as the old event definition. If the parent is not the same, a ParentNotValidException exception is thrown.

The returned boolean indicates whether an existing event definition was replaced. This is equal to true only if *replace* is equal to `true` and an event definition with the same name was replaced by the new definition.

When an event definition is added to the event catalog, the event catalog sends an event to the event server notifying event consumers that this change has taken place.

If you attempt to add an event definition that violates inheritance rules, an InheritanceNotValidException exception is thrown and the event definition is not added to the catalog. This can happen if a new event definition overrides inherited

property or extended data element descriptions in ways that are not valid, or if replacing an existing event definition causes descendants to override inherited descriptions in ways that are not valid.

Related concepts

"Change notification" on page 72

"Inheritance" on page 70

Removing an event definition from the catalog

When an event definition is removed, the event catalog does not check the event server to determine whether any existing events in the event data store are described by that event definition. Therefore, you ensure that an event definition is no longer needed before you remove it from the event catalog.

If an event definition is no longer needed, you can remove it from the event catalog.
To remove an event definition from the event catalog, use the removeEventDefinition method.

```
eventCatalog.removeEventDefinition(name)
```

The *name* parameter is the name of the event definition that you want to remove from the event catalog. If no matching event definition exists in the event catalog, an EventDefinitionNotFoundException exception is thrown.

When an event definition is removed from the event catalog, its children and all other descendants are also removed. For each event definition that is removed, the event catalog sends an event to the event server notifying event consumers that this change has taken place.

Related concepts

"Change notification" on page 72

Querying event definitions

You can use the methods of the event catalog bean to query existing event definitions. Queries exist for retrieving event definitions by name and for retrieving event definitions that satisfy specific inheritance relationships.
To query event definitions, use the appropriate method of the com.ibm.events.catalog.EventCatalog class. Methods exist for querying specific event definition or querying multiple event definitions based on name or inheritance. You can also query the root event definition. You can perform the following queries:

- Query an event definition by name
- Query an event definition by pattern
- Query the parent of an event definition
- Query the ancestors of an event definition
- Query the children of an event definition
- Query the descendants of an event definition
- Query the root event definition

Querying an event definition by name

To query a specific event definition by name, use the getEventDefinition method:

```
EventDefinition definition =
  eventCatalog.getEventDefinition(name, resolve);
```

The parameters of this method are as follows:

name
A string that specifies the name of the event definition you want to query.

resolve
A boolean value that indicates whether you want the returned event definition to be resolved (`true`) or unresolved (`false`).

The returned object is the event definition that matches the specified name. If no matching event definition exists in the catalog, the returned object is null.

Related concepts

"Inheritance" on page 70

Querying event definitions by pattern

To query all of the event definitions whose names match a specified pattern, use the getEventDefinitions method:

```
EventDefinition[] definitions =
  eventCatalog.getEventDefinitions(pattern, resolve);
```

The parameters of this method are as follows:

pattern
A string that specifies the pattern to be compared to the names of the event definitions. In this string, a percent character (`%`) matches any sequence of zero or more characters, and an underscore (`_`) matches any single character. All other characters are treated literally. For example, the pattern `insurance%` matches all of the event definitions with names that begin with the word `insurance`.

You can use a backslash (`\`) escape character to specify a literal percent or underscore character. For example, the pattern `insurance\_` matches all of the event definitions with names that begin with the string `insurance_`. To specify a backslash as part of the pattern, type two backslashes.

resolve
A boolean value that indicates whether you want the returned event definitions to be resolved (`true`) or unresolved (`false`).

The returned object is an array that contains all of the event definitions that match the specified pattern. If no matching event definitions exist in the event catalog, the returned array is empty.

Related concepts

"Inheritance" on page 70

Querying the parent of an event definition

To query the immediate parent of a specified event definition, use the getParent method:

```
EventDefinition definition =
  eventCatalog.getParent(name, resolve);
```

The parameters of this method are as follows:

name
A string that specifies the name of the event definition for which you want to query the parent.

resolve
A boolean value that indicates whether you want the returned event definition to be resolved (`true`) or unresolved (`false`).

The returned object is the immediate parent event definition of the specified event definition. If the specified event definition has no parent (which is true only of the root definition), this method returns null. If no event definition in the catalog matches the specified name, an EventDefinitionNotFoundException exception is thrown.

Related concepts

"Inheritance" on page 70

Querying the ancestors of an event definition

To query the ancestors of a specified event definition (all of the event definitions from which it inherits, either directly or indirectly), use the getAncestors method:

```
EventDefinition[] definitions =
  eventCatalog.getAncestors(name, resolve);
```

The parameters of this method are as follows:

name
A string that specifies the name of the event definition for which you want to query the ancestors.

resolve
A boolean value that indicates whether you want the returned event definitions to be resolved (`true`) or unresolved (`false`).

The returned object is an array that contains all of the ancestors of the specified event definition. If the specified event definition has no ancestors (which is true only of the root definition), this method returns an empty array. If no event definition in the catalog matches the specified name, an EventDefinitionNotFoundException exception is thrown.

Related concepts

"Inheritance" on page 70

Querying the children of an event definition

To query the immediate children of a specified event definition, use the getChildren method:

```
EventDefinition[] definitions =
  eventCatalog.getChildren(name, resolve);
```

The parameters of this method are as follows:

name
A string that specifies the name of the event definition for which you want to query the children.

resolve
A boolean value that indicates whether you want the returned event definitions to be resolved (`true`) or unresolved (`false`).

The returned object is an array that contains all of the immediate children of the specified event definition. If the specified event definition has no children, the returned array is empty. If no event definition in the catalog matches the specified name, an EventDefinitionNotFoundException exception is thrown.

Related concepts

"Inheritance" on page 70

Querying the descendants of an event definition

To query the descendants of a specified event definition (all of the event definitions that inherit from it, either directly or indirectly), use the getDescendants method:

```
EventDefinition[] definitions =
  eventCatalog.getDescendants(name, resolve);
```

The parameters of this method are as follows:

name
A string that specifies the name of the event definition whose descendants you want to query.

resolve
A boolean value that indicates whether you want the returned event definitions to be resolved (`true`) or unresolved (`false`).

The returned object is an array that contains all of the descendants of the specified event definition. If the specified event definition has no descendants, this method returns an empty array. If no event definition in the catalog matches the specified name, an EventDefinitionNotFoundException exception is thrown.

Related concepts

"Inheritance" on page 70

Querying the root event definition

To query the root event definition, use the getRoot method:

```
EventDefinition definition = eventCatalog.getRoot();
```

The returned object is the root event definition, which by default is `event`. If the event catalog is empty, this method returns null.

Working with event classes and source categories

The event catalog supports binding event definitions to source categories. This binding associates an event class (identified by extension name) to the name of an arbitrarily defined source category. Applications can use these categories to manage event classes in logical groups. These categories are entirely distinct from event groups, which are used to categorize event instances according to their content.

The event catalog does not parse or interpret the source category. The mapping between event classes and categories can be anything that is meaningful to applications that use the event catalog. An event class can belong to multiple source categories.

You can write event catalog applications to manage event classes in the following ways.

- Create new bindings
- Remove an existing binding

- Query source category bindings

Creating a source category binding

To bind an event class to a source category, use the EventCatalog.bindEventExtensionToSourceCategory method:

```
eventCatalog.bindEventExtensionToSourceCategory(extensionName, sourceCategory);
```

The parameters of this method are as follows:

extensionName
A string that specifies the extension name of an event class (the value of the extensionName attribute). This must be a legal extension name as defined by the CommonBaseEvent specification.

sourceCategory
A string that specifies a source category. This can be any string, provided it is no longer than 64 characters.

Removing a source category binding

To remove a binding between an event class and a source category, use the EventCatalog.unbindEventExtensionFromSourceCategory method:

```
eventCatalog.unbindEventExtensionFromSourceCategory(extensionName, sourceCategory);
```

The parameters of this method are as follows:

extensionName
A string that specifies the extension name of an event class (the value of the *extensionName* attribute). This must be a legal extension name as defined by the CommonBaseEvent specification.

sourceCategory
A string that specifies a source category. This can be any string, provided it is no longer than 64 characters.

Querying source category bindings

You can use the methods of the event catalog bean to query source category bindings. Queries exist for retrieving the event classes that belong to a source category, the source categories associated with an event class, or a set of source category bindings that match a specified pattern.
To query source category bindings, use the appropriate method of the com.ibm.events.catalog.EventCatalog class.

getEventExtensionNamesForSourceCategory method
Queries the event classes that belong to a specified source category:

```
String[] evClasses =
  eventCatalog.getEventExtensionNamesForSourceCategory(name);
```

The parameters of this method are as follows:

name
A string that specifies the name of a source category. This value cannot be longer than 64 characters.

The returned object is an array of strings, each string specifies an event class (the value of the extensionName attribute of events belonging to the class). If no event classes are bound to the specified source category, the returned array is empty.

getSourceCategoriesForEventExtension method

Queries the source categories associated with a specified event class:

```
String[] categories =
  eventCatalog.getSourceCategoriesForEventExtension(name);
```

The parameters of this method are as follows:

name

A string that specifies the name of an event class. This is the value of the *extensionName* attribute of events that belong to the class.

The returned object is an array of strings, each string specifies the name of a source category to which the specified event class belongs. If the specified event class is a member of any source categories, the returned array is empty.

getEventExtensionToSourceCategoryBindings method

Queries source category bindings for those that match a specified pattern:

```
java.util.Collection bindings =
  eventCatalog.getEventExtensionToSourceCategoryBindings
     (eventClassPattern, categoryPattern);
```

The parameters of this method are as follows:

eventClassPattern

A string that specifies the pattern you want to compare to the event class names. Only bindings with an event class name that matches the specified pattern are returned. In this string, a percent character (%) matches any sequence of zero or more characters, and an underscore (_) matches any single character. All other characters are treated literally.

categoryPattern

A string that specifies the pattern you want to compare to the source category names. Only bindings with a source category name that matches the specified pattern are returned. In this string, a percent character (%) matches any sequence of zero or more characters, and an underscore (_) matches any single character. All other characters are treated literally.

The returned object is a collection of two-element arrays, each representing a source category binding. In each array, the first element is a string that specifies the name of an event class. The second element is a string that specifies the name of a source category. These arrays are sorted in ascending order, first by the source category name and then by the event class name. If no bindings exist, or no existing bindings match the specified patterns, the collection is empty.

For example, the following code fragment queries all bindings of event classes for names beginning with `insurance_claim`:

```
Collection bindings =
  catalog.getEventExtensionToSourceCategoryBindings("insurance_claim%",
                                                     "%");
```

Source category bindings only associate event class names with source category names. The existence of a source category binding for a particular class name does not guarantee that an event definition exists for that event class. To retrieve the event definition associated with an event class, you must use the EventCatalog methods for querying event definitions.

Related tasks

"Querying event definitions" on page 77

Chapter 9. Viewing events with the event browser

Use the event browser to select and review common base events in the event database.

The event browser uses the event access interface to query event data. The results of the query are shown in the browser.

1. Start the event browser.
   a. Click **Enterprise applications** in the navigation pane of the administrative console.
   b. Select the **CBEViewer** check box, and click **Start**.
   c. Check that the status icon is green. Start the CBEViewer in your Web browser, using the server name and port 9080. The Web address has the format `http://<localhost>:9080/cbeviewer/`
2. Specify the events you want to view. See "Specifying the events to view."
3. Select the view of the returned events. See "Working with the returned events" on page 86.

Related tasks

"Querying events from the event server" on page 55

Specifying the events to view

This task describes how to use the event browser to specify search criteria for querying events in the event database.

This task assumes that you have already started the event browser.

1. Optional: Specify the calendar period for the report. Enter the start and end dates.
2. Enter the maximum number of records that you want to search with the specified criteria. The default number of records is `200`.
3. Required: Specify the event group to search. The default is `All events`. For more information on the event group profile, see "Default configuration" on page 9.
4. Required: Specify the data store to search. The field is a Java Naming and Directory Interface (JNDI) name, an Enterprise JavaBeans (EJB) reference that can be configured in the administrative console. The WebSphere Application Server default is `java:/comp/env/events/access`. For more information on the data store profile, see "Default configuration" on page 9.
5. Optional: Enter your other search criteria in the appropriate fields.
6. Click **Get data**.

The number of common base events that match the search criteria is displayed.

To view the returned events, select a view from the navigation bar. When you view event data, you can change your search criteria at any time by clicking **Get data**.

Related tasks

5.1.1+ Chapter 9, "Viewing events with the event browser"
Use the event browser to select and review common base events in the event database.

Working with the returned events

This task describes how to use the event browser to view the events returned from a query.

This task acts on data that is returned by a submitted query, as described in "Specifying the events to view" on page 85.

The query returns all the events that meet your criteria.

1. Select a view from the navigation bar. The navigation bar contains the following views of the previous query:

   All All the events returned.

   Processes
   Process choreographer events for a specific process instance.

   Data Events with the extension name `ECS:UserDataEvent`. This event type is created by the addUserDataEvent method of the ECSEmitter class.

   Servers
   Events for a specific server.
2. Perform one of the following actions.
   - If you selected **Processes** in step 1, select a process template, and then a process instance.
   - If you selected **Servers** in step 1, select a server.
3. Click an event to display the event data in the pane at the bottom of the browser window.

Related tasks

"Creating and populating an event using the ECSEmitter class" on page 34

Appendix. Command reference

Command-line scripts are available to access some functions of the Common Event Infrastructure. These scripts are implemented as Jacl scripts, which must be run using the WebSphere wsadmin tool (located in the *install_root*/bin directory). For more information about the wsadmin tool, see the WebSphere Application Server documentation.

Use the following syntax to run the scripts:

```
wsadmin -f scriptname.jacl
```

You can shorten parameter names, as long as you provide enough of the name to distinguish it from other parameters. For example, when you use the eventquery.jacl script, you can type -ex instead of -extensionname. However, -e is not valid, because it can represent either -extensionname or -end.

To get help with the syntax and usage for a command, type the command followed by the word help:

```
wsadmin -f scriptname.jacl help
```

If you are using the wsadmin tool with the SOAP protocol, a command might time out before the operation can complete. For example, this might happen if you query or purge a large number of events from the event server. If this happens, the wsadmin tool displays an error message indicating a failed SOAP RPC call:

```
Failed to make the SOAP RPC call: invoke
```

If you get this error message, try the command again, specifying RMI as the connection type and 2809 as the destination port. For example, the following command purges events from the event server using an RMI connection:

```
wsadmin -conntype rmi -port 2809 eventpurge.jacl -seconds 0
```

For more information about the -conntype parameter of the wsadmin tool, refer to the WebSphere Application Server documentation.

emitevent.jacl

Purpose

Sends an event to the event server.

wsadmin -f emitevent.jacl [**-xml** *url*] [**-msg** *message*] [**-severity** *severity*] [**-extensionname** *extension_name*] [**-emitter** *profile_name*] [**-synchronous** | **-asynchronous**]

Description

The emitevent.jacl script provides a command-line interface for submitting events to the event server. You can provide the event content by providing a source XML file or by specifying property values on the command line.

Events generated by this script have the following default content:

```
<CommonBaseEvent creationTime=current_system_time version="1.0.1">
 <sourceComponentId component="emitevent.jacl" componentIdType="Application"
  location=local_hostname locationType="Hostname"
  subComponent="com.ibm.events.cli.util.EmitEventCliHelper"
  componentType="http://www.ibm.com/namespaces/autonomic/Tivoli/EventInfrastructure"/>
 <situation categoryName="ReportSituation">
  <situationType xsi:type="ReportSituation" reasoningScope="EXTERNAL"
   reportCategory="CLI"/>
 </situation>
</CommonBaseEvent>
```

The current_system_time parameter is the system time at which the event is generated, specified as an XML dateTime string.

Parameters

-xml *url*
A uniform resource locator (URL) that specifies the location of an XML document that contains the event to be submitted. This XML document must conform to the Common Base Event version 1.0.1 XSD schema. If no URL scheme (such as `http://`) is specified, a local file is assumed. This parameter is optional.

Two sample XML files, `eventsample1.xml` and `eventsample2.xml`, are available in the *install_root*`/events/samples` directory.

-msg *message*
The value to use for the message property of the event. If the message contains spaces, enclose this value in quotation marks. This parameter is optional. If you specify this parameter in addition to an XML file, the value of the **-msg** parameter overrides any value specified in the XML file for the msg property.

-severity *severity*
The value to use for the severity property of the event. This parameter is optional. If you specify this parameter in addition to an XML file, the value of the **-severity** parameter overrides any value specified in the XML file for the severity property.

-extensionname *extension_name*
The value to use for the extensionName property of the event. If the extension name contains spaces, enclose this value in quotation marks. This parameter is optional. If you specify this parameter in addition to an XML file, the value of the **-extensionname** parameter overrides any value specified in the XML file for the extensionName property.

-emitter *profile_name*
The Java Naming and Directory Interface (JNDI) name of the emitter factory profile to use when obtaining an emitter. This parameter is optional. If this parameter is not specified, the default emitter factory profile (`/com/ibm/events/configuration/emitter/Default`) is used.

-synchronous | -asynchronous
The synchronization mode to use for event transmission. This parameter is optional. If it is not specified, the preferred synchronization mode configured for the emitter is used.

Examples

The following example sends an event to the event server with a severity of 30 and the extension name `test_event` (all other properties have the default values):

```
wsadmin -f emitevent.jacl -severity 30 -extensionname test_event
```

The following example sends an event using the properties specified in `eventsample1.xml`:

```
wsadmin -f emitevent.jacl -xml ../samples/eventsample1.xml
```

eventquery.jacl

Purpose

Generates a report listing events in the event database.

wsadmin -f eventquery.jacl [**-globalinstanceid** *global_instance_id* | **-group** *event_group*] [**-severity** *severity*] [**-extensionname** *extension_name*] [**-start** *start_time*] [**-end** *end_time*] [**-number** *number*] [**-ascending**| **-descending**]

Description

The `eventquery.jacl` script queries the event database and generates a report that lists the result. You can query events based on the event group, the severity, or the extension name. You can also query events that were created during a specified period of time.

Parameters

-globalinstanceid *global_instance_id*
The global instance identifier of the event to query. Either this parameter or **-group** (but not both) is required.

-group *event_group*
The event group from which to query events. The *event_group* value must be the name of an event group defined in the Common Event Infrastructure configuration. Either this parameter or **-globalinstanceid** (but not both) is required.

-severity *severity*
The severity of the events that you want to include in the report. The *severity* value must be an integer. Only events with a severity equal to the value that you specify are included in the report. This parameter is optional.

-extensionname *extension_name*
The extension name of events that you want include in the report. Use this parameter to restrict the query to events of a specific type. Only events with the extensionName property equal to *extensionName* are included in the report. This parameter is optional.

-start *start_time*
The earliest time of the events that you want to include in the report. Use this parameter to restrict the query to events that were generated after a specified date and time. This parameter must be a date and time that is specified according to the XML dateTime data type. The basic format is `CCYY-MM-DDThh:mm:ss`, optionally followed by a time zone indicator. For example, noon on 1 January 2004 in Eastern Standard Time is `2004-01-01T12:00:00-05:00`. For more information about the dateTime data type, refer to the XML schema at www.w3.org.

-end *end_time*
The latest time of the events that you want to include in the report. Use this parameter to restrict the query to events that were generated before a specified

date and time. This parameter must be a date and time that is specified according to the XML dateTime data type. For more information, see the description of the **-start** parameter.

-number *number*
The maximum number of events that you want to include in the report. This parameter must be an integer. If the number of matching events in the database exceeds the specified value, the report is truncated. If the report is sorted in ascending order, this means that the most recent matching events are omitted. If the report is sorted in descending order, the oldest matching events are omitted.

-ascending | -descending
The chronological order in which the events in the report are sorted. This must be one of the following values:
ascending
Ascending (chronological) order, with the oldest events first. This is the default value.
descending
Descending (reverse chronological) order, with the most recent events first.

Example

The following example lists all of the events from the database that belong to the **All events** event group and were generated on 17 February 2004. The report is sorted in reverse chronological order:

```
eventquery.jacl -group "All events" -start "2004-02-17T00:00:00-05:00"
  -end "2004-02-17T23:59:59-05:00" -order DESC
```

eventpurge.jacl

Purpose

Purges events from the event database.

wsadmin -f eventpurge.jacl -group *event_group* [**-severity** *severity*] [**-extensionname** *extension_name*] **-seconds** *seconds* [**-size** *size*]

Description

The `eventpurge.jacl` script purges events from the event database. You can purge all events from the event database, or you can limit the purge to events meeting certain criteria.

Parameters

-group *eventGroup*
The event group from which to purge the events. The *event_group* value must be the name of an event group that is defined in the Common Event Infrastructure configuration. This parameter is required.

-severity *severity*
The severity of the events that you want to purge. The *severity* value must be an integer. Only events with a severity equal to the value that you specify are purged. This parameter is optional.

-name *extension_name*
The extension name of the events that you want to purge. Use this parameter to restrict the purge to events of a specific type. Only events with an extensionName property equal to *extensionName* are purged. This parameter is optional.

-seconds *seconds*
The minimum age of the events that you want to purge. The *seconds* value must be an integer. Only events older than the specified number of seconds are purged. This parameter is required.

-size *size*
The number of events to purge in a single transaction. The *size* value must be an integer. After this number of events is purged, the command commits the transaction before continuing in a new transaction. This parameter is optional.

Example

The following example purges all of the events from the database with a severity of 20 (harmless) that were generated earlier than 10 minutes ago.

```
eventpurge.jacl -group "All events" -severity 20 -seconds 600
```

eventcatalog.jacl

Purpose

Lists event definitions or source categories in the event catalog and imports and exports event definitions.

wsadmin -f eventcatalog.jacl [**-listdefinitions** | **-listcategories** | **-exportdefinitions** | **-importdefinitions**] [**-file** *filename*] [**-name** *event_def_name*] [**-pattern**] [**-resolve**] [**-replace**]

Description

The `eventcatalog.jacl` script provides command-line access to the contents of the event catalog. It also provides support for importing and exporting event definitions.

Parameters

-listdefinitions
Lists the specified event definitions in a readable format, sorted by name in ascending order. The listing is written to the file specified by the **-file** parameter. If this parameter is not specified, the listing is written to the standard output.

-listcategories
Lists all of the defined event source categories and the event classes they contain, sorted by source category in ascending order. The listing is written to the file specified by the **-file** parameter. If this parameter is not specified, the listing is written to the standard output.

-exportdefinitions
Lists the specified event definitions in a format that is suitable for importing. The listing is written as an XML document conforming to the `eventdefinition5_0_1.xsd` XSD schema, which is packaged in the

`events-client.jar` file. The listing is written to the file specified by the **-file** parameter. If this parameter is not specified, the listing is written to the standard output.

-importdefinitions
Reads a listing of event definitions from a file and adds the event definitions to the event catalog. The listing of event definitions to import must be written as an XML document that conforms to the `eventdefinition.xsd` XSD schema.

-file *filename*
For a list or export operation, the name of the file to which the output is written. For an import operation, the file that contains the event definitions to be imported. This parameter is required for import operations and optional for list and export operations. If this parameter is not specified for a list or export operation, the output is written to the standard output.

-name *event_def_name*
A name that identifies the event definitions to be listed or exported. If the **-pattern** parameter is not specified, the **-name** parameter identifies a single specific event definition. If **-pattern** is specified, **-name** specifies a pattern against which event definition names are compared. In this pattern, a percent character (`%`) matches any sequence of zero or more characters, and an underscore (_) matches any single character. All other characters are treated literally.

This parameter is valid only with the **-listdefinitions** and **-exportdefinitions** options. It is not valid with the **-listcategories** or **-importdefinitions** options.

-pattern
Specifies that the value specified with the **-name** parameter is to be treated as a pattern. This parameter is valid only with the **-listdefinitions** and **-exportdefinitions** options. It is not valid with the **-listcategories** or **-importdefinitions** options.

-resolve
Specifies that the event definitions to be listed or exported are resolved. A resolved event definition includes the property and extended data element descriptions that are inherited from its ancestors in the inheritance hierarchy. If this parameter is not specified, the event definition listing contains only the raw event definitions.

This parameter is valid only with the **-listdefinitions** and **-exportdefinitions** options. It is not valid with the **-listcategories** or **-importdefinitions** options.

-replace
Specifies that the event definitions to be imported replace existing event definitions with the same names. If this parameter is not specified, a name collision between an existing event definition and an imported event definition results in an error, and no event definitions are imported.

This parameter is valid only with the **-importdefinitions** option. It is not valid with the **-listdefinitions**, **-listcategories**, or **-exportdefinitions** options.

Examples

This example displays the contents of a single, resolved event definition named `insurance_claim_start` and writes the result to standard output:

```
wsadmin -f eventcatalog.jacl -listdefinitions -name insurance_claim_start -resolve
```

This example exports a set of event definitions, the names of which begin with the string `insurance_claim_start` and writes the result to an XML file:

```
wsadmin -f eventcatalog.jacl -exportdefinitions -file d:\myexport.xml
  -name insurance_claim_start% -pattern
```

This example imports a set of event definitions from the file `myimport.xml`and replaces existing definitions with the same names:

```
wsadmin -f eventcatalog.jacl -importdefinitions -file d:\myimport.xml -replace
```

This example displays a listing of all defined event source categories and the events they contain. The result is written to standard output:

```
wsadmin -f eventcatalog.jacl -listcategories
```

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, New York 10594 USA

Trademarks and service marks

The following terms are trademarks of IBM Corporation in the United States, other countries, or both:

- AIX
- AS/400
- CICS
- Cloudscape
- DB2
- DFSMS
- Domino
- Everyplace
- iSeries
- IBM
- IMS
- Informix
- iSeries
- Language Environment
- Lotus
- MQSeries
- MVS
- OS/390
- RACF
- Redbooks
- RMF
- SecureWay
- SupportPac
- Tivoli
- ViaVoice
- VisualAge
- VTAM
- WebSphere
- z/OS
- zSeries

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

Microsoft, Windows, Windows NT, and the Windows logo 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.

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