Deploying and Managing Apama Applications

5.1.0

November 2013

This document applies to Apama 5.1.0 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: -Deploying_And_Managing_Apama_Applications-5.1-20131107

Table of Contents

Preface

About this documentation

This documentation discusses deploying and managing Apama applications and covers the following topics:

- Starting and managing Apama components with the Enterprise Management and Monitoring console (EMM)
- Starting and managing correlators with Apama command line utilities
- Providing and managing access to Apama applications with dashboards
- Collecting and managing event data with Apama Database Connector (ADBC)
- Tuning Apama applications for optimum performance

Preface

How this book is organized

The information in this book is organized as follows:

- "Overview of Deploying Apama Applications" on page 17 — A general description of issues involved in deploying the components that make up an Apama application.
- "Using the Management and Monitoring Console" on page 21 — How to use the Enterprise Management and Monitoring console (EMM) to configure, deploy, and monitor Apama components.
- "Deploying and Configuring Correlators" on page 51 — How to deploy correlators using the Enterprise Management and Monitoring console.
- "Deploying and Configuring Adapters" on page 71 — How to deploy IAF adapters using the Enterprise Management and Monitoring console.
- "Dashboard Deployment Concepts" on page 83 — Describes fundamental dashboard deployment and administration concepts.
- "Deploying Dashboards" on page 91 — How to install and configure dashboard deployment packages generated by the Dashboard Deployment Wizard.
- "Event Correlator Utilities Reference" on page 103 — How to deploy and manage correlators using the Apama command line utilities.
- "Tuning Correlator Performance" on page 167 — How to scale up performance by using multiple correlators and how to preserve a correlator’s runtime state.
- "Using the Apama Database Connector" on page 179 — How to use the Apama Database Connector to collect and store event data.
- "Managing the Dashboard Data Server and Display Server" on page 327 — How to start, manage, and stop the Data Server and the Display Server.
- "Administering Dashboard Security" on page 347 — How to use Apama and your application server to provide a secure environment for deployed dashboards.

- "Using the Apama Component Extended Configuration File" on page 363 — How to set up an optional configuration file that can be used to launch Apama correlators.

Preface

Documentation roadmap

On Windows platforms, the specific set of documentation provided with Apama depends on whether you choose the Developer, Server, or User installation option. On UNIX platforms, only the Server option is available.

Apama provides documentation in three formats:

- HTML viewable in a Web browser
- PDF
- Eclipse Help (if you select the Apama Developer installation option)

On Windows, to access the documentation, select Start > All Programs > Software AG > Apama 5.1 > Documentation . On UNIX, display the `index.html` file, which is in the `doc` directory of your Apama installation directory.

The following table describes the PDF documents that are available when you install the Apama Developer option. A subset of these documents is provided with the Server and User options.

Title	Contents
What's New in Apama	Describes new features and changes since the previous release.
Installing Apama	Instructions for installing the Developer, Server, or User Apama installation options.
Introduction to Apama	Introduction to developing Apama applications, discussions of Apama architecture and concepts, and pointers to sources of information outside the documentation set.
Using Apama Studio	Instructions for using Apama Studio to create and test Apama projects; write, profile, and debug EPL programs; write Java programs; develop custom blocks; and store, retrieve and playback data.
Developing Apama Applications in Event Modeler	Instructions for using Apama Studio's Event Modeler editor to develop scenarios. Includes information about using standard functions, standard blocks, and blocks generated from scenarios.
Developing Apama Applications in EPL	Introduces Apama's Event Processing Language (EPL) and provides user guide type information for how to write EPL programs. EPL is the native interface to the correlator. This document also provides information for using the standard correlator plug-ins.
Apama EPL Reference	Reference information for EPL: lexical elements, syntax, types, variables, event definitions, expressions, statements.

Title	Contents
Developing Apama Applications in Java	Introduces the Apama Java in-process API and provides user guide type information for how to write Java programs that run on the correlator. Reference information in Javadoc format is also available.
Building Dashboards	Describes how to create dashboards, which are the end-user interfaces to running scenario instances and data view items.
Dashboard Property Reference	Reference information on the properties of the visualization objects that you can include in your dashboards.
Dashboard Function Reference	Reference information on dashboard functions, which allow you to operate on correlator data before you attach it to visualization objects.
Developing Adapters	Describes how to create adapters, which are components that translate events from non-Apama format to Apama format.
Developing Clients	Describes how to develop C, C++, Java, or .NET clients that can communicate with and interact with the correlator.
Writing Correlator Plug-ins	Describes how to develop formatted libraries of C, C++ or Java functions that can be called from EPL.
Deploying and Managing Apama Applications	Describes how to: ● Use the Management & Monitoring console to configure, start, stop, and monitor the correlator and adapters across multiple hosts. ● Deploy dashboards over wide area networks, including the internet, and provide dashboards with effective authorization and authentication. ● Improve Apama application performance by using multiple correlators, and saving and reusing a snapshot of a correlator's state. ● Use the Apama ADBC adapter to store and retrieve data in JDBC, ODBC, and Apama Sim databases. ● Use the Apama Web Services Client adapter to invoke Web Services. ● Use correlator-integrated JMS messaging to reliably send and receive JMS messages in Apama applications.
Using the Dashboard Viewer	Describes how to view and interact with dashboards that are receiving run-time data from the correlator.

Preface

Contacting customer support

You may open Apama Support Incidents online via the eService section of Empower at http://empower.softwareag.com. If you are new to Empower, send an email to `empower@softwareag.com` with your name, company, and company email address to request an account.

If you have any questions, you can find a local or toll-free number for your country in our Global Support Directory at https://empower.softwareag.com/public_directory.asp and give us a call.

Preface

1 Overview of Deploying Apama Applications

Deploying and managing Apama applications involves the following topics:

- Starting and managing Apama components with the Enterprise Management and Monitoring console (EMM)
- Starting and managing correlators with Apama command line utilities
- Providing and managing access to Apama applications with dashboards
- Collecting and managing event data with the Apama Database Connector (ADBC)
- Tuning Apama applications for optimum performance

Deploying components with EMM

Apama's Enterprise Management and Monitoring (EMM) console provides a graphical interface for configuring, deploying, and monitoring various Apama components across multiple hosts.

- For a general overview of EMM, see "Using the Management and Monitoring Console" on page 21.
- For information on using EMM to start and manage correlators, see "Deploying and Configuring Correlators" on page 51.
- For information on using EMM to start and manage adapters, see "Deploying and Configuring Adapters" on page 71.

Overview of Deploying Apama Applications

Using Apama command line utilities

Apama provides a variety of command line tools for managing and monitoring Apama correlators. For information and instructions on using these tools to monitor and manage event correlators, see "Event Correlator Utilities Reference" on page 103.

Overview of Deploying Apama Applications

Deploying dashboards

Dashboard deployment and administration involves the following activities:

- Deployment package installation and configuration—see "Deploying Dashboards" on page 91.
- Data Server and Display Server management—see "Managing the Dashboard Data Server and Display Server" on page 327.
- Security administration—see "Administering Dashboard Security" on page 347.

Before you perform these tasks, you should familiarize yourself with the deployment and administration concepts introduced in this section. Each concept described here is covered in more detail in "Dashboard Deployment Concepts" on page 83.

Deployment options

Dashboards can be deployed as simple, thin-client Web pages, as Java applets, as Java Web Start applications, or as files that can be loaded into a locally-installed, desktop application, the Dashboard Viewer. These deployment options are described and compared in "Deployment options" on page 84.

Data server and display server

Scalability and security of dashboard deployment are supported by the use of the *Data Server* and *Display Server*, which mediate dashboard access to running Scenarios and DataViews. The Data Server and Display Server are introduced in "Data server and display server" on page 86.

Process architecture

Applet and Web Start dashboards communicate with the Data Server via servlets running on an application server. Simple, thin-client, Web-page dashboards communicate with the Display Server via servlets running on your application server. Locally-deployed dashboards communicate directly with the Data Server. The structure of deployed configurations is detailed in "Process architecture" on page 86.

Builders and administrators

Dashboard deployment involves the use of a dashboard deployment package that was generated by Apama Studio's Dashboard Deployment Configuration Editor. In some cases the user that generated the deployment package is different from the user that installs and configures the deployment and administers the Data Server. The information that must be transferred between these two types of users is discussed in "Builders and administrators" on page 89.

Overview of Deploying Apama Applications

Tuning applications for performance

The performance of Apama applications can be enhanced by employing multiple correlators. For information about strategies for deploying multiple correlators and the Apama tools available for accomplishing this, see "Tuning Correlator Performance" on page 167. The section also contains information about preserving a correlator's runtime state.

Overview of Deploying Apama Applications

2 Using the Management and Monitoring Console

This section describes how to use Apama's Enterprise Management and Monitoring (EMM) console to configure, deploy, and monitor Apama components.

EMM allows centralized, graphical management and monitoring of an Apama installation. From it, you can configure, start, stop and monitor Apama software components (correlators and IAF integration adapters) across multiple hosts. You can also centrally install and upgrade licenses.

Installation

EMM can be run from any computer that has network access to the hosts where Apama components are installed. That is, it does not have to be installed on the machine on which the other Apama components are to be run, although doing so is not detrimental as long as the host is powerful enough to cope with the performance requirements of the application being run on Apama.

Note: In this release, EMM is supported on only Windows platforms. However, it can manage components running on all supported hosts. For a complete list of supported platforms go to Software AG's Knowledge Center in Empower at https://empower.softwareag.com.

Once installed on the host of choice, you can run EMM by selecting the Program Filess > SoftwareAG > Apama 5.1 > Administration > Management and Monitoring menu item from the Windows Start menu.

Alternatively, you can run it from the command line as follows:

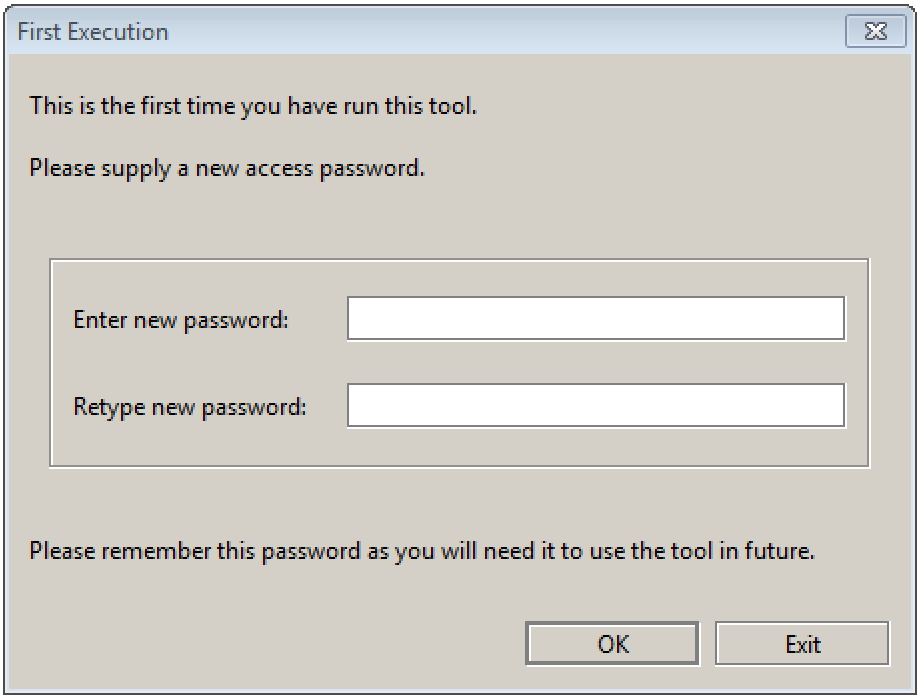
```
emm
```

from an Apama Command Prompt window

When EMM is run for the first time you will be asked to configure an access password, as shown in the illustration, below.

- If you enter a password here, the password will be required whenever EMM is launched. It can be changed subsequently through the Options menu.
- If you do not enter a password here (i.e., leave the field blank), EMM will not prompt for a password when it starts up. This can be useful if you want to start EMM from a script.

Note: It is important that you keep a record of the access password, as you will be asked to enter it on all subsequent uses of the tool. If you forget the password you will have to reinstall the tool from the original installation distribution.


First Execution

This is the first time you have run this tool.

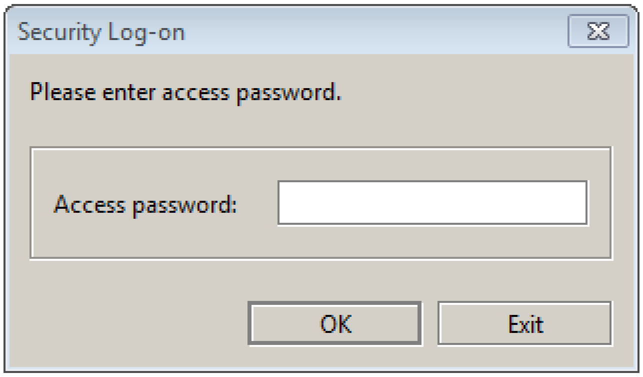
Please supply a new access password.

Enter new password:

Retype new password:

Please remember this password as you will need it to use the tool in future.

If you are not running the tool for the first time, you will be asked to enter the access password:


Security Log-on

Please enter access password.

Access password:

Once you have entered the correct password, you then have access to all the capabilities of the tool.

Using the Management and Monitoring Console

Concepts

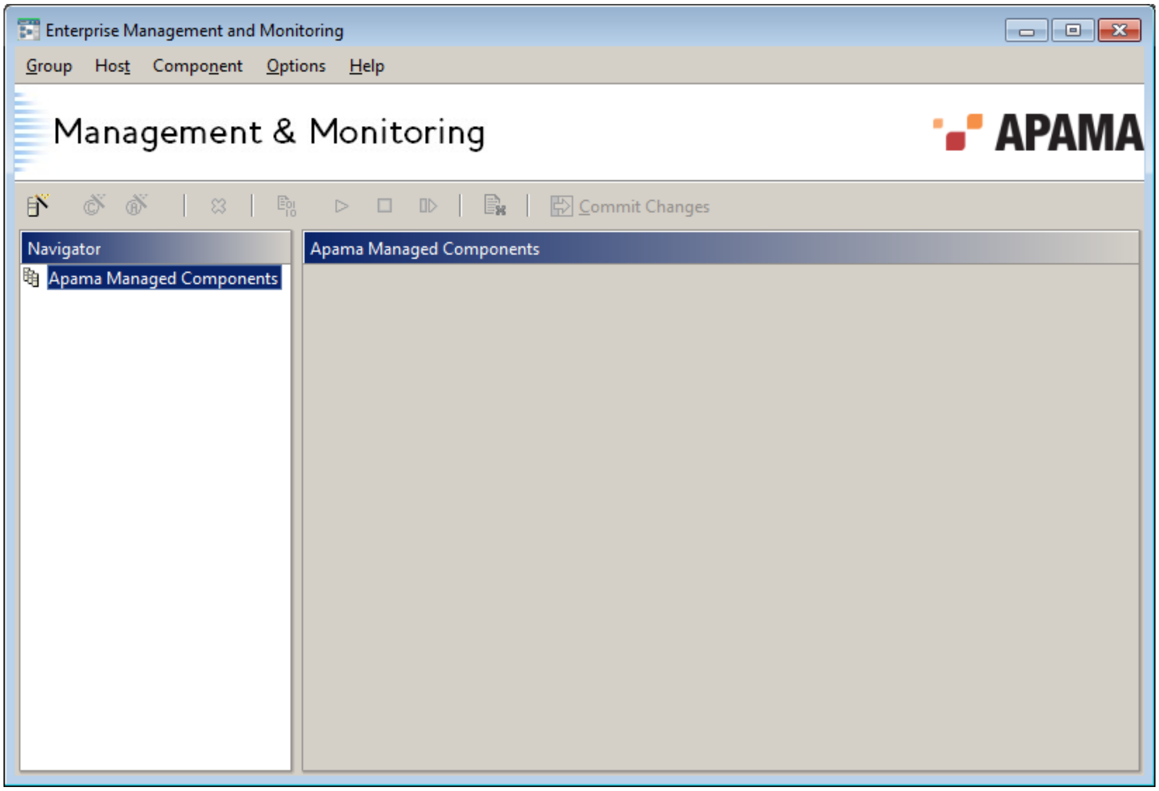
Two basic entities can be configured and manipulated in EMM: *hosts* and *components*. *Hosts* refer to personal computers, workstations or servers, running Solaris, Linux or Windows, which have Apama installations. In addition these machines must have had the *Sentinel Agent* (see "Sentinel Agents" on page 28) installed on them and it must be running. "Managing hosts" on page 28 describes the facilities provided by EMM for dealing with hosts.

Components refer to Apama event correlators and IAF adapters. Apama components can be run and manipulated by EMM on any host that is running the Sentinel Agent. "Managing components" on page 38 describes the facilities provided by EMM for dealing with components.

Using the Management and Monitoring Console

The EMM console window

The following illustration shows the key features of EMM's main window.



Menubar

The Menubar is the main way of interacting with the tool and carrying out operations on hosts and components.

Toolbar

The Toolbar contains a set of icons that provide a convenient graphical shortcut to the most common operations that can be carried out on hosts and components. These operations can also all be carried out from the Menubar.

Navigation Pane

The Navigation Pane lists the set of hosts that EMM is aware of and the Apama components that are running on them. It also provides visual cues as to the current operating status of the hosts and components. Right clicking on an entity displays a context-sensitive menu that lists the operations that can be carried out on that entity. These operations can also all be carried out from the Menubar and from the Toolbar.

Details Pane

The Details Pane displays information that is relevant to the currently selected host or component. Its contents change accordingly, and can be used to both configure as well as to monitor and inspect the status of the selected item.

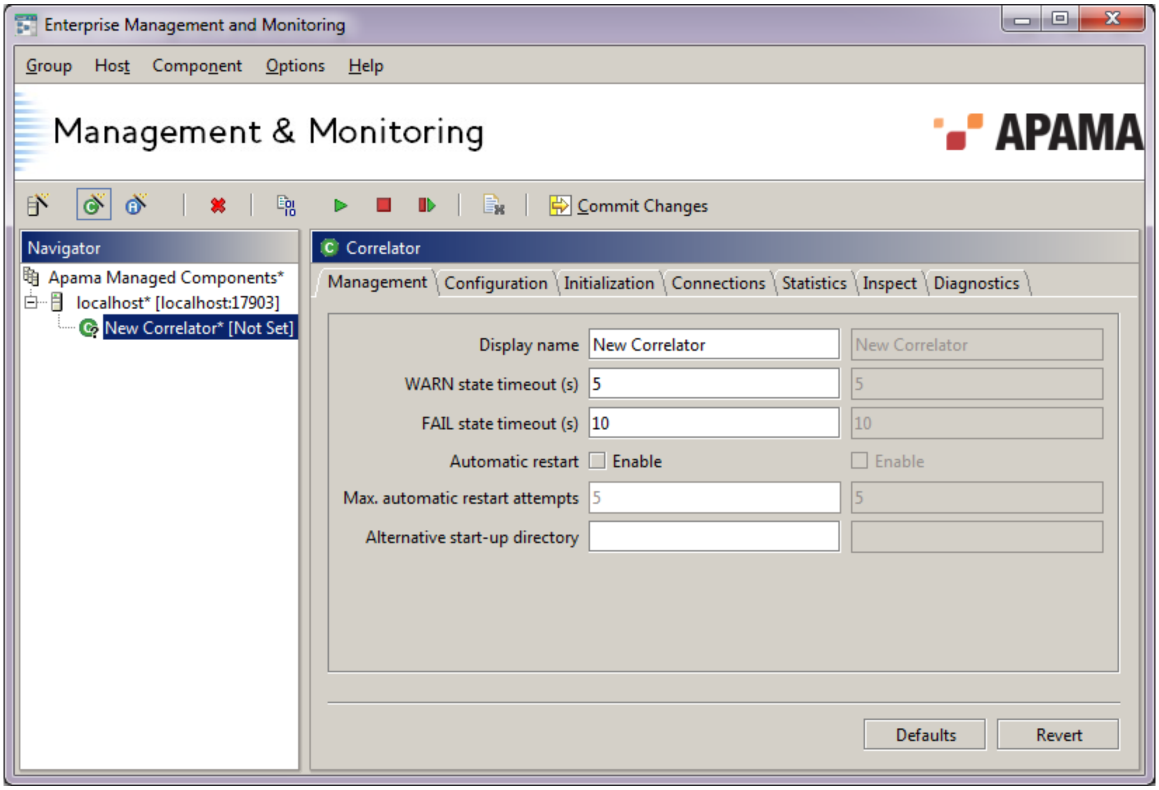
Concepts

Committed values and outstanding changes

EMM can configure and manipulate hosts and Apama components, and to do this it needs to maintain extensive configuration information.

Throughout EMM there are concepts of the *current, committed* configuration data and of *outstanding changes*, or *working copy* data.

On each Details Pane for a host or a specific component where you can supply configuration data, you will see two columns of values, as shown in the following illustration. The rightmost, grayed out values indicate the actual *committed* data – the values currently in use. The leftmost column is where you can provide the new values you would like to use. As long as they are just shown in the leftmost columns, these *outstanding changes* are just that – temporary working data. A component or host with unsaved changes will have a "*" suffix in its entry in the Navigation Pane. For changes to take effect you need to commit them, using the Commit Changes button ( ) on the Menubar, or the relevant menu items on the Group, Host and Component menus. Note that some component options cannot be changed once the component has started, and these will not take effect until the component is restarted, even after being committed.



Display name	New Correlator	New Correlator
WARN state timeout (s)	5	5
FAIL state timeout (s)	10	10
Automatic restart	□ Enable	□ Enable
Max. automatic restart attempts	5	5
Alternative start-up directory		

The Commit Changes action scope varies according to what is selected when it is pressed:

- if a component is selected in the Navigation Pane, then all outstanding changes in that component, and only in that component, are committed (the Commit changes item on the Component menu is enabled in this case),
- if a host is selected in the Navigation Pane, then all outstanding changes in that host, and in all its components, are committed (the Commit changes item on the Host menu is enabled in this case),
- if Apama Managed Components is selected in the Navigation Pane, than all outstanding changes throughout all of EMM's hosts and components are committed (the Commit changes item on the Group menu is enabled in this case).

Please refer to "Managing hosts" on page 28 and "Working with hosts" on page 31 for information on the Details Pane for hosts and for the specific components supported by the EMM console.

Concepts

State persistence

All of EMM's management state, including the list of hosts and components it is managing and the committed settings for each (including the diagnostic messages), are persisted by EMM to disk.

When EMM is stopped and restarted no settings are lost, although please note that in the current version EMM's status monitoring and auto-restart capabilities are only active while the console is running.

The state is stored in a file in the Apama Work directory called `etc\emm_state.dat` and a backup is created in `etc\emm_state.dat.bak`. EMM must have the necessary permissions to read/ write these files. If there is a problem locating or reading the state file EMM will offer to load from the backup file instead. If neither file can be located or read, EMM will revert back to its default installation stage configuration. Should EMM fail to save its state file correctly for whatever reason then temporary files of the form `serialXXXXX.tmp` will be created in the `etc` directory; these files can be safely deleted.

Using the Management and Monitoring Console

Managing licenses

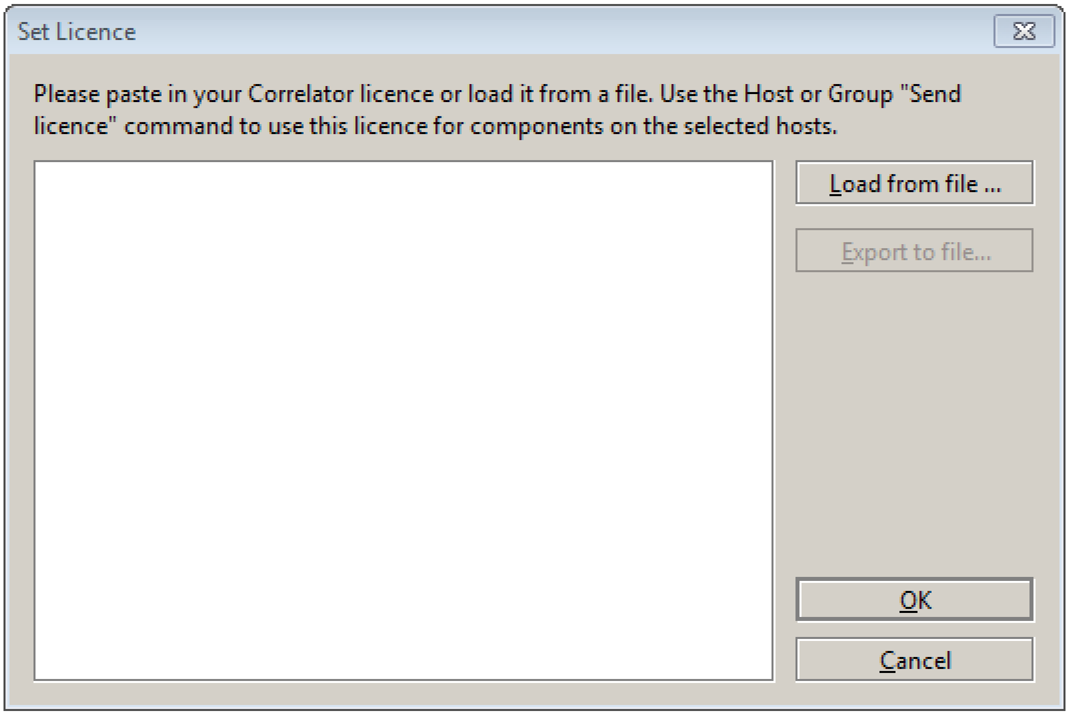
An Apama event correlator requires a valid Apama license. As well as allowing the correlator to start, a license specifies restrictions in its functionality and performance. A file containing a valid license must be available to each correlator all the time.

EMM aids license deployment and maintenance by its facility to push out licenses to any host where Apama correlators are to be started (or are already running). This facility also allows updating of existing installed licenses.

To load a license for use with EMM:

1. Select Set License... from the Options menu option on the Menubar.

   This displays the Set License dialog box shown in the following illustration.



Set Licence

Please paste in your Correlator licence or load it from a file. Use the Host or Group "Send licence" command to use this licence for components on the selected hosts.

In theory you can type in the license but as it is quite a long string and must be entered exactly as it was sent to you by Apama Technical Support, it is greatly preferable to read it in from a text file.

2. Select Load from file ..., locate the license file from where you saved it, and Open it.
3. Click OK to set this as the current active license to be used by EMM.

However, note that this license is only actually sent out to hosts when the Send license option is applied to the host entity. Alternatively, the licenses on all known hosts can be updated at the same time by selecting Apama Managed Components in the Navigation Pane and choosing the Send License option from the Group menu.

Note: EMM will not attempt to validate your license; therefore it is important that you enter it correctly. If you attempt to start a correlator and the host has no license file then the correlator will fail to start unless it is running on the local host. If the correlator is on the host named “localhost” or “127.0.0.1” then it will start, but in a restricted mode that prevents connections to other hosts and the correlator will automatically terminate after 30 minutes of operation; it will be flagged with a warning icon to indicate that it has started in this state (look at the component’s Diagnostics tab for more information). If the license file exists but is expired or invalid then the correlator will fail to start.

Using the Management and Monitoring Console

Managing security

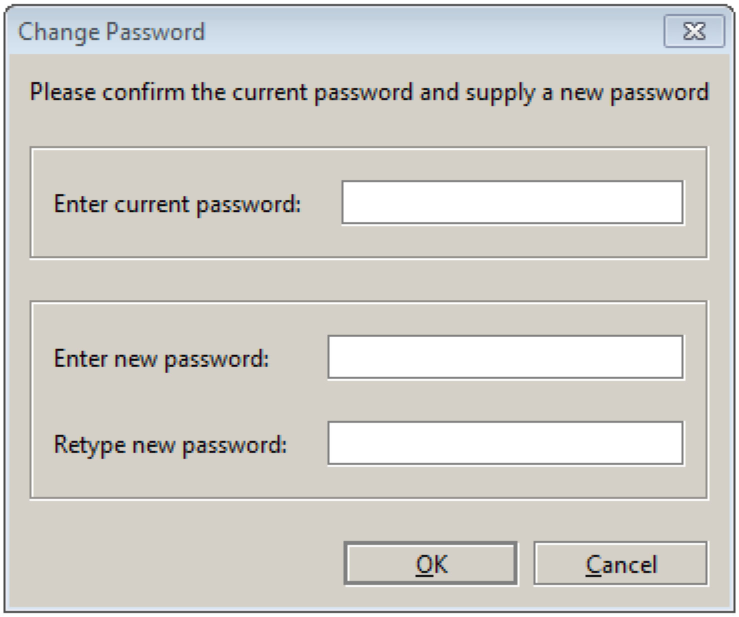
When you run EMM, if you have configured a password as described in "Installation" on page 22, the start up process will always request an access password. If you configured a password when you first ran EMM, this step cannot be bypassed. Should you forget the access password, you will need to reinstall the tool from the Apama installation media.

If you left the password field empty as described in "Installation" on page 22, you are not prompted to enter a password.

To change the password:

1. Select Change Password... from the Options menu option on the Menubar.

   This displays the **Change Password** dialog, shown in the following illustration.



Change Password

Please confirm the current password and supply a new password

Enter current password:

Enter new password:

Retype new password:

Using the Management and Monitoring Console

Managing hosts

EMM can start and stop components on any Windows XP/2003, Sun Solaris 10, RedHat Enterprise Linux 4.0, and SUSE Linux Enterprise Server 10.0 host within your network, as long as that host has had the relevant Apama components installed on it, and as long as it is running a Sentinel Agent.

Using the Management and Monitoring Console

Sentinel Agents

A Sentinel Agent is a small process that can start and stop Apama correlators and IAF adapters on the host where it is running. EMM starts and stops components on a particular host in a platform independent way by interacting with the Sentinel Agent running on that host. Therefore a Sentinel Agent must be running on any machine where you intend to manipulate Apama components using EMM.

On Windows, the Sentinel Agent can be started in one of these ways:

- If you checked the sentinel agent check box during the Apama installation process, sentinel agent is automatically installed with `startup type =automatic`.

- By starting the Windows Service with the apama_services command apama_services -start -s sentinel. The service can be stopped with apama_services -stop -s sentinel. If the service was not added during installation it can be added with apama_services -add -s sentinel.
- By starting the Sentinel Agent executable manually but not as a Windows Service. This can be done either from an Apama command prompt or command shell where you have run apama_env.bat to set the necessary Apama environment variables. Run sentinel_agent.exe and pass command line arguments to it.

On UNIX, Sentinel Agent can be started in one of these ways:

- By adding it as a service (daemon) through the installer. In this case the Sentinel Agent will automatically start up when the machine boots.
- By starting the service with the apama_services command apama_services -start -s sentinel. The service can be stopped with apama_services -stop -s sentinel. If the service was not added during installation it can be added with apama_services -add -s sentinel.
- By starting the Sentinel Agent either from a script or command shell where you have sourced apama_env. Run sentinel_agent and pass command line arguments to it.

Note: Any information written to stdout or stderr by components started with a Sentinel Agent will be logged in the Agent's log directory in files called *<component-type>*-*<PID>*-std{out,err}.log on UNIX and *<component-type>*-*<num>*-std{out,err}.log on Windows, where *<PID>* = Process Id and *<num>* is an integer number. Valid component types are correlator and adapter. The stdout / stderr logs will be deleted by Sentinel when the relevant component is stopped if they are empty (which should normally be the case).

Managing hosts

Options

The Sentinel Agent's startup and usage information is as follows:

```
Usage: sentinel_agent [ options ]
Where options may include any of:
   -V | --version             Print program version info
   -h | --help                This message
   -p | --port <port>         Listen on alternate port (default is 17903)
   -f | --logfile <file>      Log to named file (default is stderr)
   -v | --loglevel <level>    Set logging verbosity.  Available levels
                              are TRACE, DEBUG, INFO, WARN, ERROR CRIT FATAL
                              (default logging level is WARN)
   -t | --truncate            Truncate log file at startup
   -N | --name <name>         Set the component name
   -l | --license <file>      Apama license filename
   -H | --apamahome <dir>     Root of Apama installation
   -W | --apamawork <dir>     Root of Apama work directory
 -Xconfig | --configFile <file> Use service configuration file <file>
```

The license is assumed to be in APAMA_WORK\license\license.txt unless overridden on the command line. The Apama work directory is set from the -W argument or from the APAMA_WORK environment variable if -W is not specified.

These options can be specified to the Sentinel Agent executable whether it is run as a normal Windows or UNIX executable.

Options

Table 1. Sentinel Agent options

Option	Description
`-h`	Display the above usage information. Does not start the Sentinel Agent.
`-p`	Name of the host on which the Sentinel Agent will listen for connections from EMM (default is `17903`).
`-l`	A reference to an Apama license file. If provided, the Sentinel Agent will be able to pass this license file's location on to the Apama components that it starts. Alternatively a license file can be sent to the Sentinel Agent dynamically via EMM.
`-H`	The root directory where the Apama executable files are installed, normally `C:\Program Files\Software AG\Apama` *<rel_num>*`\bin` on Windows or `/opt/SoftwareAG/apama_`*<rel_num>*`/bin` on UNIX.
`-W`	The root directory where the Apama work files are located, normally `C:\Users\`*`username`*`\Software AG\ApamaWork_`*`version`* on Windows or *$HOME*`/apama-work` on UNIX.
`-f`	The file where to output logging information from the Sentinel Agent. If a log file is not specified and the Sentinel Agent was run as a normal executable from a Command Prompt or a Terminal window, it will log to the system configured `stderr`. If the Sentinel Agent was run as a Windows service, a log file must be specified; otherwise the Agent will not start.
`-N` *name*	Sets the name of the component to *name*.
`-t`	If set, the log file will be emptied at start up, otherwise it will be appended to.
`-v`	Set the logging level. The log levels possible, in increasing order, are `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, `CRIT`, and `FATAL`, where each level includes all subsequent ones. For example, `WARN` includes all `WARN` level, all `ERROR` level, all `CRIT` level,and all `FATAL` level log messages. If not specified the default log level is `WARN`.
`-V`	Print out the Sentinel Agent's version information. Does not start the Sentinel Agent.
`-Xconfig`*file*	Use the specified service configuration file. For more information about this configuration file, see "Using the Apama Component Extended Configuration File" on page 363.

Managing hosts

Working directory

The Sentinel Agent's current working directory/folder becomes the current working directory of all components started by the Agent, unless an `Alternative start-up directory` was configured for the component. The working directory is important because most of the files and paths specified when configuring components in EMM are assumed to be relative to the Sentinel Agent's working directory if a full absolute path is not provided.

Determining the working directory

If the Sentinel Agent was started manually then its current working directory is simply the current directory when it was run.

If it was started automatically – either as a service on Windows or as a daemon on UNIX – then the current working directory is determined as follows.

On Windows:

1. Normally, the Sentinel Agent tries to use the `logs` subdirectory of the Apama work directory.
2. Otherwise the Sentinel Agent tries to use the `%WinDir%\System32` directory.
3. If the Sentinel Agent cannot write to any of these directories, it will fail to start.

On UNIX:

1. The Sentinel Agent tries to use the `/logs` subdirectory of the Apama work directory.
2. If the Sentinel Agent cannot write to this directory, it will fail to start.

Managing hosts

Working with hosts

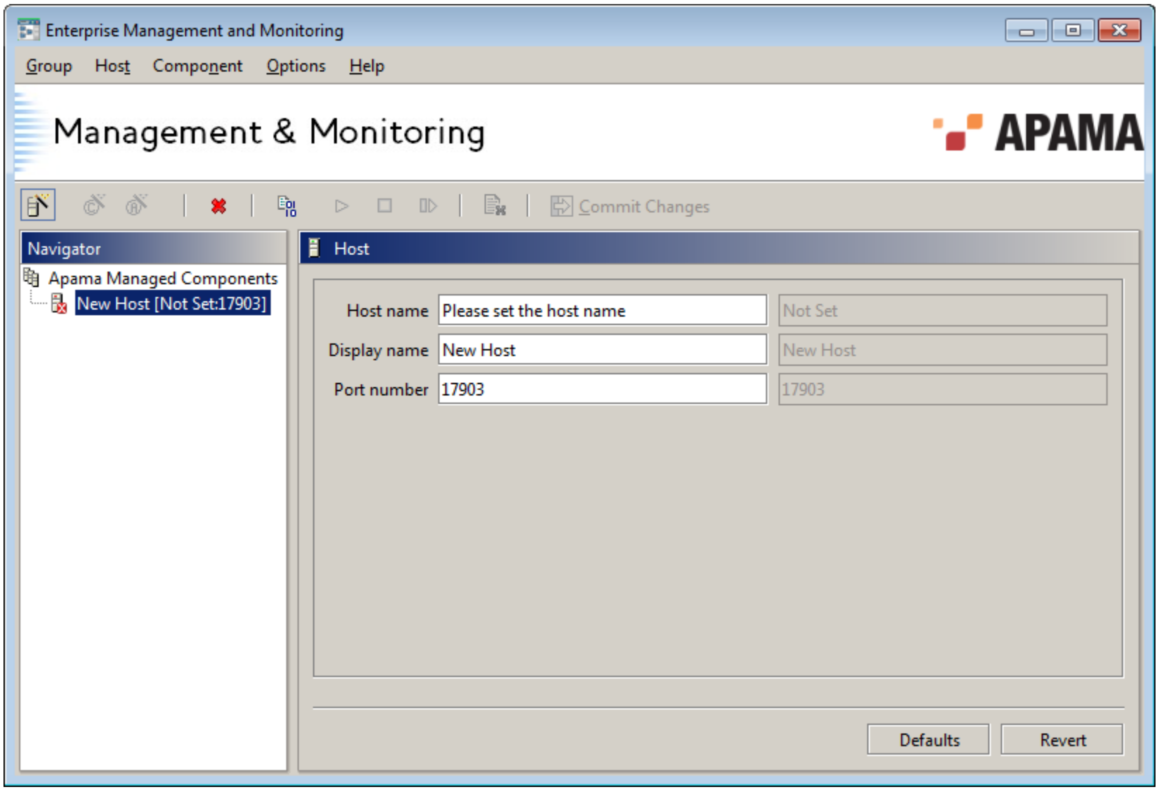
This documentation refers to the items on the EMM menu as the primary means of carrying out operations on hosts. However, these operations can also be carried out from the context menus in the Navigation Pane, and the most useful are also available on the toolbar.

The Host menu provides the following options for interacting with hosts:

Add host

This is equivalent to using the button on the toolbar.

Add host allows you to add a host to the `Apama Managed Components` group, the root node illustrated in the Navigation Pane. This adds a placeholder node in the Navigation Pane and provides options for configuring EMM to work with the host in the Details Pane, as illustrated in the following.



Host name: Please set the host name — Not Set

Display name: New Host — New Host

Port number: 17903 — 17903

You need to provide the following information:

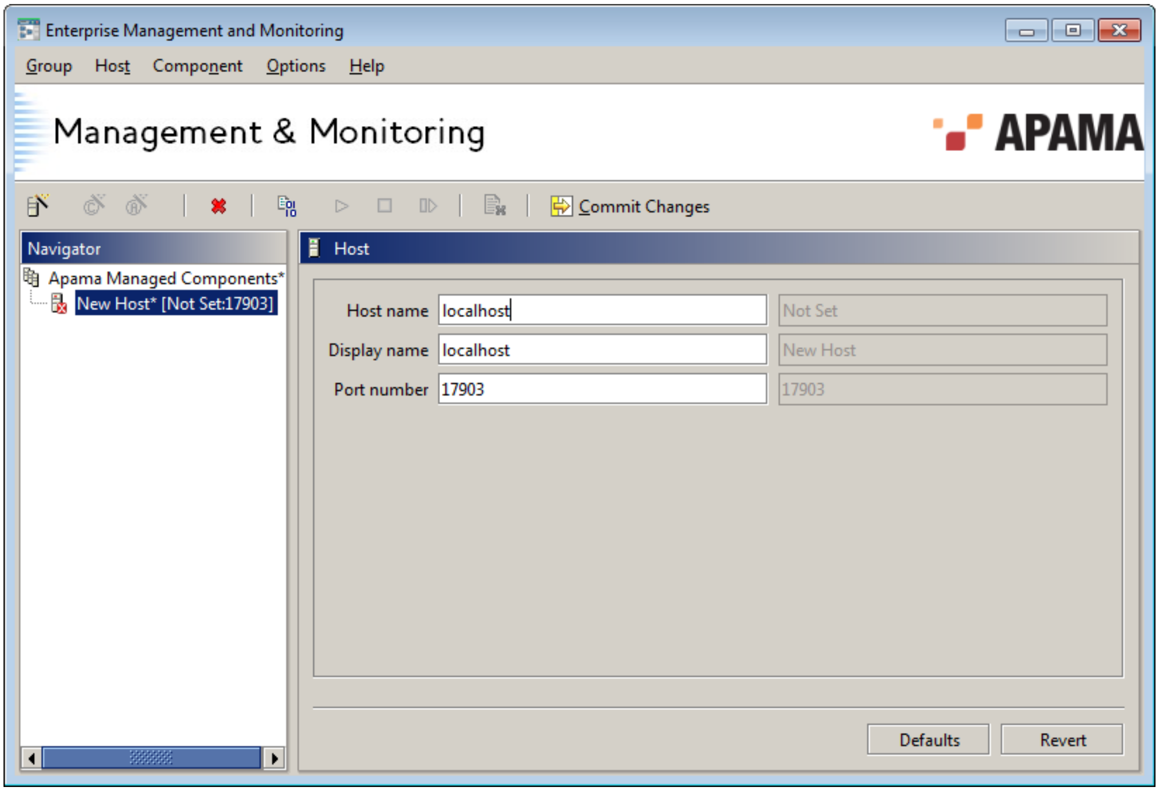
- Display name – A descriptive name for the host. This name is what is displayed in the Navigation Pane.
- Host name – The host's name or IP address, for example `lawrence.apama.com` or `127.0.0.1`. Whether you need to provide the full name as in this example, or whether an abbreviation (e.g. `lawrence`) will do depends on your network configuration and DNS/WINS settings, but in most circumstances it is advisable to use the host's full network name.

You can only use ASCII characters in a host name.

Note that it is not possible to change the Host name once components have been added to the host.

- Port number – The port that the Sentinel Agent (on that host) is listening on. By default this is `17903`, although it might have been changed by whoever installed and configured the Sentinel Agent on that host.

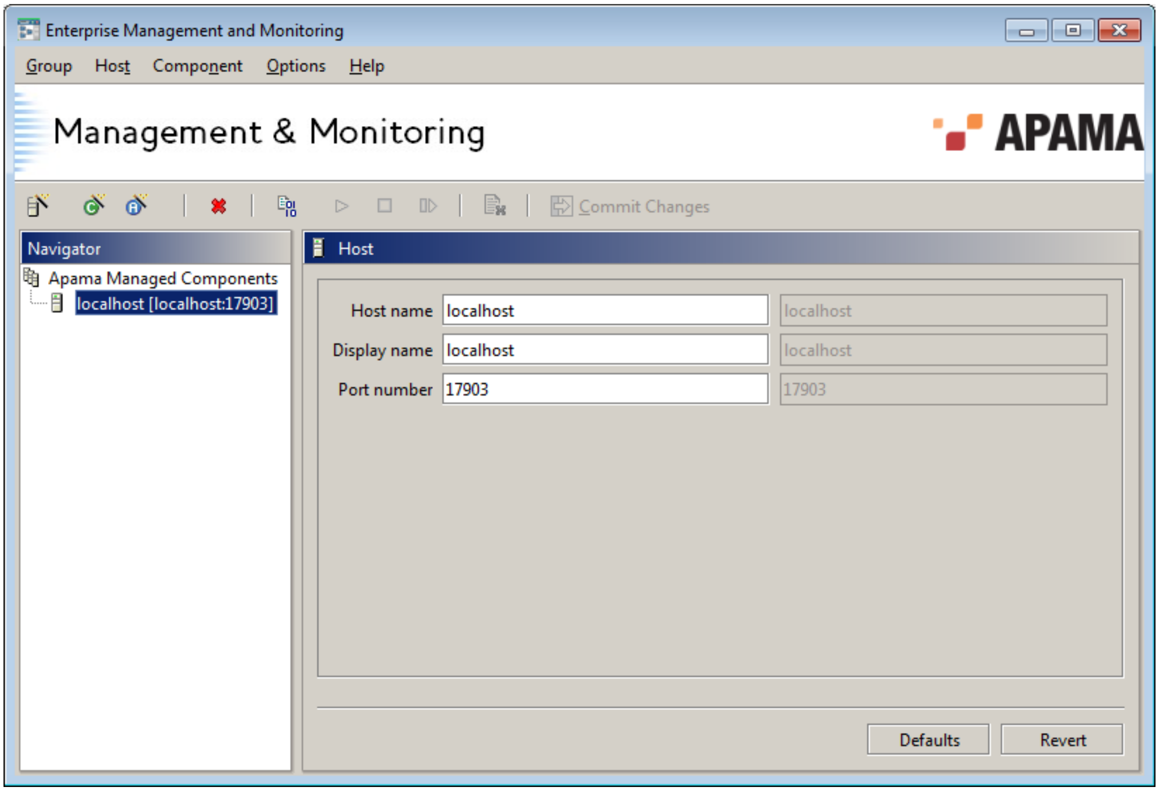
In following illustration, we have created a reference to the host `localhost` and specified port `17903` (the default port) as the port to use for communicating with the Sentinel Agent running on this machine.


Enterprise Management and Monitoring
Group Host Component Options Help
Management & Monitoring
APAMA
Commit Changes
Navigator
Apama Managed Components*
New Host* [Not Set:17903]
Host
Defaults
Revert

Host name: localhost (Not Set)
Display name: localhost (New Host)
Port number: 17903 (17903)

Once the host information has been set, click on the Commit Changes button ( ) on the toolbar. Note that no components can be added to a host until it is committed.

The Commit Changes button will be disabled if the Host name has not yet been set or if the host has already been setup and committed, and there are no outstanding changes to commit on the host or any associated component.


Enterprise Management and Monitoring
Group Host Component Options Help
Management & Monitoring
APAMA
Commit Changes
Navigator
Apama Managed Components
localhost [localhost:17903]
Host
Defaults
Revert

Host name: localhost (localhost)
Display name: localhost (localhost)
Port number: 17903 (17903)

Notice how the icons next to the new node in the Navigation Pane are different in the illustrations above. When you first define a new host the icon alongside it in the Navigation Pane will be ☒. This will change to ▯ once EMM successfully manages to contact the Sentinel Agent running on that host.

EMM will continue verifying communication with the Sentinel Agent on that host at regular intervals, for as long as that host is included in the Apama Managed Components group.

If EMM cannot communicate with a host's Sentinel Agent, or connects and subsequently loses connection, the icon against that host will change to ☒ and stay that way until the problem is resolved. This might be the case if the host is no longer available on the network, the network has failed, the remote Sentinel Agent has been shut down or if a firewall (such as the one installed by default with Microsoft Windows XP Service Pack 2) is blocking the port that the EMM console uses to communicate with the Sentinel Agent.

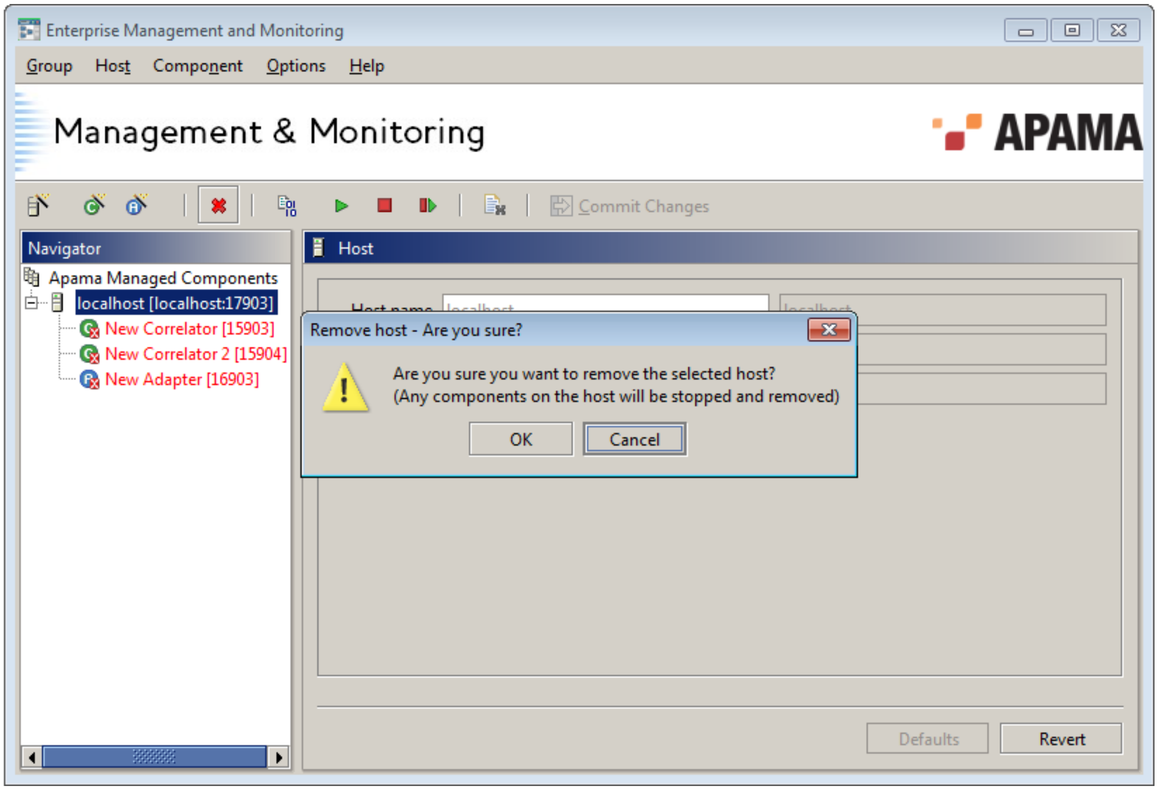
Remove host

This is equivalent to using the ✖ button on the toolbar, and is only available if a host is currently selected in the Navigation Pane.

Choose it to remove all components on that host from the Apama Managed Components group.

Note that this operation does not shut down the remote Sentinel Agent on the host in question.

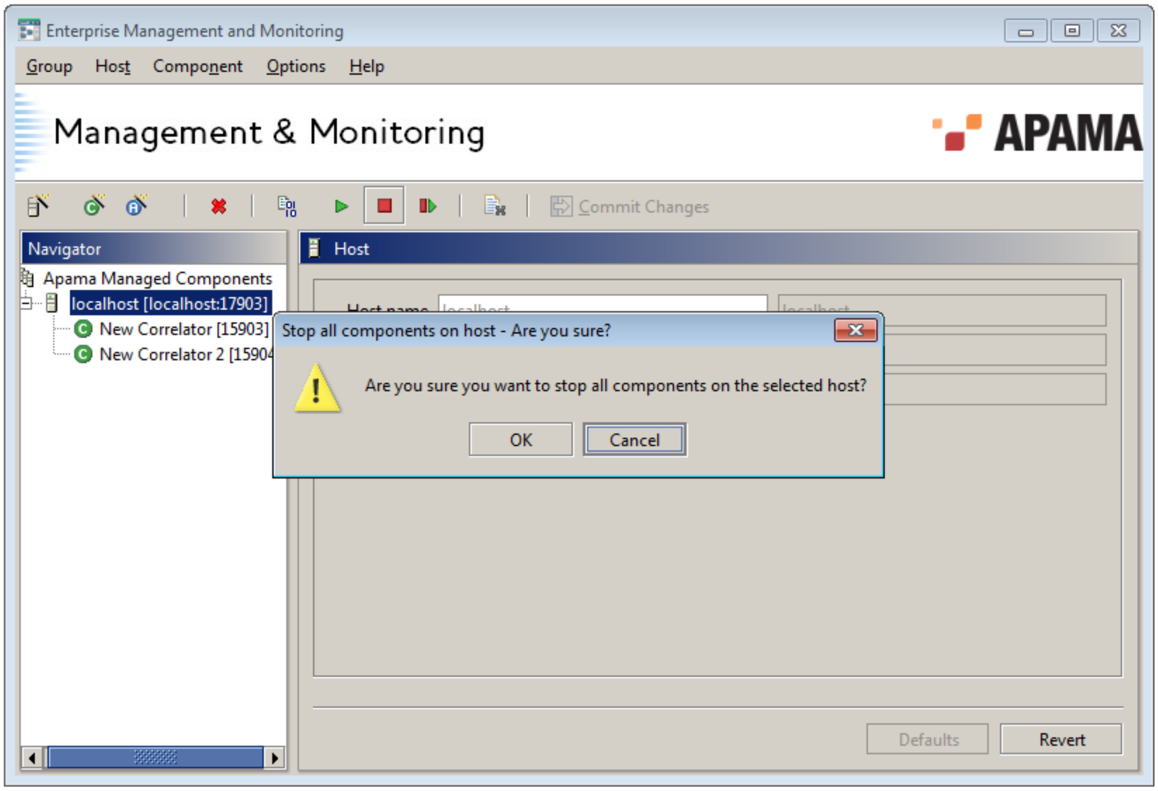
By default, EMM displays a confirmation dialog providing the option to cancel the removal of the host, as shown in the following illustration.



Remove host - Are you sure?

Are you sure you want to remove the selected host?
(Any components on the host will be stopped and removed)

If the host contains one or more components that are known to be running, as well as providing the option to cancel the removal the dialog asks whether these component(s) should be stopped before being removed; see the following illustration.



Stop all components on host - Are you sure?

Are you sure you want to stop all components on the selected host?

Note: See "Preferences" on page 44 for information about how these confirmation dialogs can be turned on or off using the EMM Preferences dialog.

Start component(s)

This is equivalent to using the ▶ button on the toolbar. This option is available if a host is currently selected in the Navigation Pane.

Choose it to start all Apama components configured on that host. Please see Chapter 3 for details on how to add and configure Apama components.

Please refer to "Start component" on page 41 for information on why components might fail to start.

Stop component(s)

This is equivalent to using the ■ button on the toolbar. This option is available if a host is currently selected in the Navigation Pane.

Choose it to stop all Apama components configured on that host.

Restart component(s)

This is equivalent to using the ❚▶ button on the toolbar. This option is available if a host is currently selected in the Navigation Pane.

Choose it to re-start (stop and start again) all Apama components configured on that host.

Send license

This is equivalent to using the button on the toolbar. This option is available if a host is currently selected in the Navigation Pane and a license has been configured within EMM (through the Options / Set license menu option).

Choose it to push out the currently configured license to the Sentinel Agent running on that host, which the Sentinel Agent will use to create a local license file on that host. The license file will be written to the location that was specified in the Sentinel Agent's command line startup parameters, replacing any existing license file.

A dialog will be displayed indicating if the license push was successful or whether the license was rejected as invalid or some other error occurred.

Components do not have to be restarted to become aware of the new license file.

Managing hosts

Managing all known hosts

All hosts defined within EMM are attached to the Apama Managed Components group item in the Navigation Pane.

The Group menu option allows one to run operations across all hosts in this group; that is all known hosts, without having to select them individually.

The operations available on the Group menu are also available by selecting the Apama Managed Components item in the Navigation Pane and then right-clicking to get its context-sensitive popup menu. While the item is selected you can also use the toolbar buttons to the same effect.

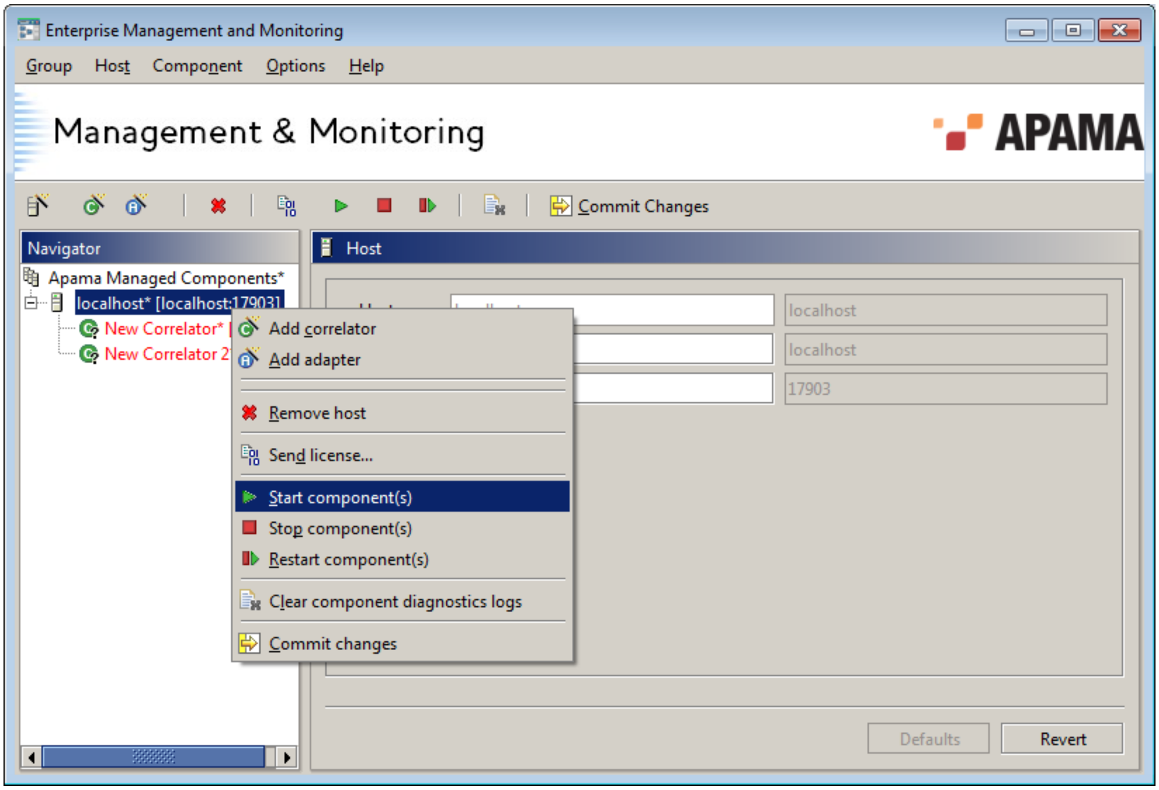
Restart components

Choose this option to restart all components on all hosts. This is equivalent to clicking ▮▶ on the toolbar.

Start components

Choose this option to start all components on all hosts. This is equivalent to clicking ▶ on the Menubar.

Please refer to "Start component" on page 41 for information on why components might fail to start.



Stop components

Choose this option to stop all components on all hosts. This is equivalent to clicking 🟥 on the Menubar.

Send license

Update the license on all hosts. This is equivalent to clicking 📄 on the Menubar.

See "Send license" on page 36 for more information about license sending.

Managing hosts

Managing components

EMM can support various Apama components that can be used together to create a distributed Apama deployment.

The components are:

- Event Correlators – the event correlator is the core Apama event matching and analytic engine. See "Add correlator" on page 40 and "The correlator tabs" on page 53.
- IAF Adapters – the Integration Adapter Framework (IAF) allows rapid generation of integration adapters. These allow Apama to interface with an external source of events like a message bus, an event feed, or a database. See "Add adapter" on page 40 and "The Adapter tabs" on page 74.

For information on how to use Apama command line utilities to run and configure the event correlator, see "Event Correlator Utilities Reference" on page 103.

For information about generating custom adapters with the Apama Integration Adapter Framework, see "The Integration Adapter Framework" in *Developing Adapters*. For information about the standard Apama pre-packaged adapters, see "Standard Apama Adapters" in *Developing Adapters*. The *Developing Adapters* documentation is available if you selected Developer during installation.

Using the Management and Monitoring Console

Component status indicators

EMM can configure, start and stop components. For all the components it manages, EMM can monitor execution and automatically restart in case of failure (re-initializing where applicable).

As components are managed, their icons in the Navigation Pane will change to indicate their status. Although the base icons for a correlator and an IAF adapter are different, being Ⓒ and Ⓐ, they have the same set of symbols overlaid on them to indicate their state. Using the correlator's icons as an example, the indicated states are:

- Ⓒ - UNKNOWN. The state is still unknown, i.e. EMM is trying to communicate with the component.
- Ⓒ - STOPPED. The component is not running and it is not supposed to be, that is, it has not been started, or it was explicitly stopped, in EMM.
- Ⓒ - RUNNING. The component is running normally and it is supposed to be, that is, it was started from EMM.
- Ⓒ - STOPPING, STARTING. A stop or start operation is in progress.
- Ⓒ - WARN. This can mean one of the following:
  - The component is supposed to be running but is not responding – EMM has lost contact with it and is trying to reestablish communication. If this does not succeed the component will progress to the FAIL state.

  - The component itself started correctly but one or more initialization actions failed. Use the component's Diagnostics tab to investigate where the problem occurred. The warning can be cleared by pressing the Clear warning button on the Diagnostics tab.
  - The component has been started but it is not supposed to be running (e.g. it was started from outside EMM). The warning can be cleared by pressing the Clear warning button on the Diagnostics tab.
  - A correlator started on the local machine but there was no license file, so the component will automatically terminate after 30 minutes. The warning can be cleared by pressing the Clear warning button on the Diagnostics tab, however Apama recommends supplying a valid license and restarting the component.
  - The component itself started correctly but one or more upstream connections to other components could not be established. Use the component's Diagnostics tab to investigate where the problem occurred. The warning can be cleared by pressing the Clear warning button on the Diagnostics tab.

- - FAIL. The component has failed. The component stopped responding to EMM for the entire WARN timeout configured for the it plus the FAIL timeout, so EMM now assumes that the component is no longer running.

In case of a FAIL check the Diagnostics tab to discover which of these situations occurred.

Note: The time to wait after a component stops responding before entering the WARN state, and the time between entering the WARN state and progressing to FAIL may both be configured on a per-component basis using the Management tab in the Details Pane.

Using the Management and Monitoring Console

Working with components

This section refers to the items on the EMM menu as the primary means of carrying out operations on Apama components. However, these operations can also be carried out from the context menus in the Navigation Pane, and the most useful are also available on the toolbar.

Before operations can be carried out on these components, a Sentinel Agent must be installed and running on the target host, and the EMM console must have been configured to manage that host, as detailed in "Managing hosts" on page 28.

The following options are available from the Components menu when a host is selected:

- Add correlator
- Add adapter
- Remove component
- Move component to host
- Start component
- Stop component
- Restart component
- Clear all component logs

Using the Management and Monitoring Console

Add correlator

This option is only available if a host (or an existing component on a host) is selected in the Navigation Pane. It is equivalent to pressing the button on the toolbar.

Choose this option to define and configure a new event correlator component on the selected host.

Working with components

Add adapter

This option is only available if a host (or an existing component on a host) is selected in the Navigation Pane. It is equivalent to pressing the button on the toolbar.

Choose this option to define and configure a new IAF Adapter component on the selected host. This will add an Adapter to the host in the Navigation Pane, select it, and display the Management tab in the Details Pane.

Working with components

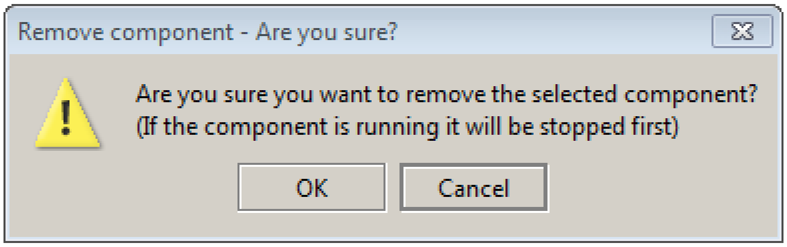
Remove component

This option is only available if a component is selected in the Navigation Pane. It is equivalent to pressing the ✖ button on the Menubar.

Choose this to remove the currently selected component from the Navigation Pane and stop it from being managed by EMM.

You can also remove a component by selecting it and pressing the Delete key.

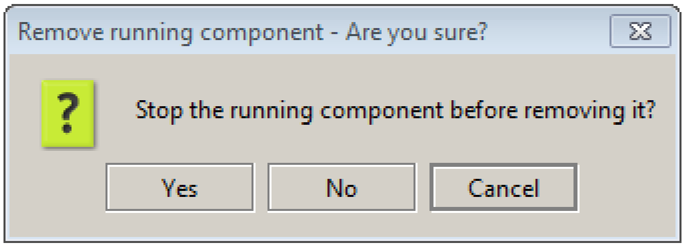
By default, EMM displays a confirmation dialog providing the option to cancel the removal of the component, as shown below.



Remove component - Are you sure?

Are you sure you want to remove the selected component?
(If the component is running it will be stopped first)

If the component is known to be running, as well as providing the option to cancel the removal the dialog asks whether the component should be stopped before being removed, as shown below.



Remove running component - Are you sure?

Stop the running component before removing it?

Note: "Warnings" on page 44 describes how these confirmation dialogs can be turned on or off using the **EMM Preferences** dialog.

Working with components

Move component to host

This option is only available if a component is selected in the Navigation Pane. When a component is selected and you select Move component to host from the Component menu, the **Select New Host for Component** dialog is displayed. Select the new host and click OK.

Any component type may be moved with this command. The moved component will have exactly the same configuration as it did on its previous host, including the same logical ID.

Working with components

Start component

This option is only available if a component is selected in the Navigation Pane. It is equivalent to pressing the ▶ button on the toolbar.

Choose this option to start the currently selected component. The component's icon will change to indicate that the component is starting, and then to show it is operational. Note that this can take a few seconds since the icon is only updated once EMM can satisfactorily communicate with the component.

A component can fail to start for several reasons. The most common are:

- A valid license was not available on that host.
- The Sentinel Agent is not running on the host (check that its icon in the Navigator Pane is ▯ and not ☒)
- The component's executable was corrupt, or inaccessible due to user permissions issues.
- The component's listening port was not available as it is in use by something else.
- There was insufficient disk space on that host for the Sentinel Agent's or the component's logging.
- The component could not write to the specified log file due to user or file permissions issues.
- The component could not locate its essential runtime libraries. Either the component or the libraries it requires were manually moved, or the environment variables are set incorrectly on that host.

- You might have set important management and configuration parameters but forgotten to commit them.
- The component might appear not to have started even though in fact it has, due to a firewall (such as the one installed by default with Microsoft Windows XP Service Pack 2) blocking the port that the EMM console uses to communicate with the component. The firewall must be disabled or reconfigured to allow traffic on the port the component is configured to listen on.
- The Sentinel Agent on that host could not locate the component’s binary. This means it was not configured correctly; normally this is due to it not having had valid component paths specified.

Note: All components started by the Sentinel Agent are started under the same user context (and therefore with the same privileges) as the user the Sentinel Agent itself was started as.

Therefore, if the Sentinel Agent was started as a service as the user `Local System`, all the components it starts will also start as the user `Local System`. Note that `Local System` is not a Windows Domain user, and therefore might not be allowed access to Domain resources. This might be particularly relevant to IAF adapters which link against custom user code and are likely to require access to various network services and resources.

If this situation is encountered, you need to change the user that the Sentinel Agent is started as, which can be done through Control Panel, Administrative Tools, Services, right click on Apama Sentinel Agent, select Properties, then the Log On tab. Choose the This account: radio button and enter the username and password of the user to use instead. You can specify a domain user by preceding the username with the domain as follows: <*domain name*>/<*username*>. You will of course have to be an Administrator on the machine to be able to do any of this.

Working with components

Stop component

This option is only available if a component is selected in the Navigation Pane. It is equivalent to pressing the 🟥 button on the toolbar.

Choose this option to stop the currently selected component. The component’s icon will change to indicate that the component is no longer running or available. Note that as the icon is only updated once EMM can no longer communicate with the component this can take a few seconds.

Working with components

Restart component

This option is only available if a component is selected in the Navigation Pane. It is equivalent to pressing the ▮▶ button on the toolbar.

Choose this option to trigger a restart on the currently selected component. The component’s icon will change to indicate that the component has stopped and then change again to indicate it has started up and is operational again.

Working with components

Clear all component logs

This option is equivalent to clicking the ▤ button. If Apama Managed Components is selected in the Navigation Pane, this option clears the diagnostic logs for all the managed components. If a host is selected in the Navigation Pane, it clears the diagnostic logs for all the components on that host.

Working with components

Configuring components with the details pane

The contents of the Details Pane change according to what is currently selected in the Navigation Pane. When a component is added to a host, it is automatically selected in the Navigation Pane.

> Note: As already described, you need to press Commit Changes (⇨) before any changes you make to any panel on the Details Pane take effect. The values that are currently in effect are shown within the rightmost grayed out labels.

Using the Management and Monitoring Console

Specifying paths and filenames in the Details Pane

On several of the component Details Pane panels you are asked to provide a path or filename, (for example to specify where logging information should be stored, or where an adapter configuration file should be loaded from).

It is recommended that *absolute paths* and filenames be used where possible – an absolute path being one that specifies the whole path, for example:

`c:\Document and Settings\My User\apama-work\logs\` (on Windows)

or

`/home/apama-work` (on UNIX)

In contrast, *relative paths* are incomplete, or just consist of a filename on its own, such as:

`New_Correlator_1.log`

Important notes

- Unless otherwise stated, all paths are located on the file system of the remote host, on which the Sentinel Agent and managed components are running, rather than the local host where EMM is running.
- Filenames and paths should never be enclosed in quotes, even if they contain spaces.
- If a relative path is provided – as is the case by default for component log files – it is assumed to be relative to the component's `Alternative start-up directory` setting if one was configured, or to the current working directory of the Sentinel Agent on that host if not. See "Working directory" on page 31 for details of how the Sentinel Agent's working directory is determined.

Configuring components with the details pane

Preferences

Several of EMM's default operational characteristics such as the warnings it issues, some of its display properties, and how it updates component information can be configured by the user.

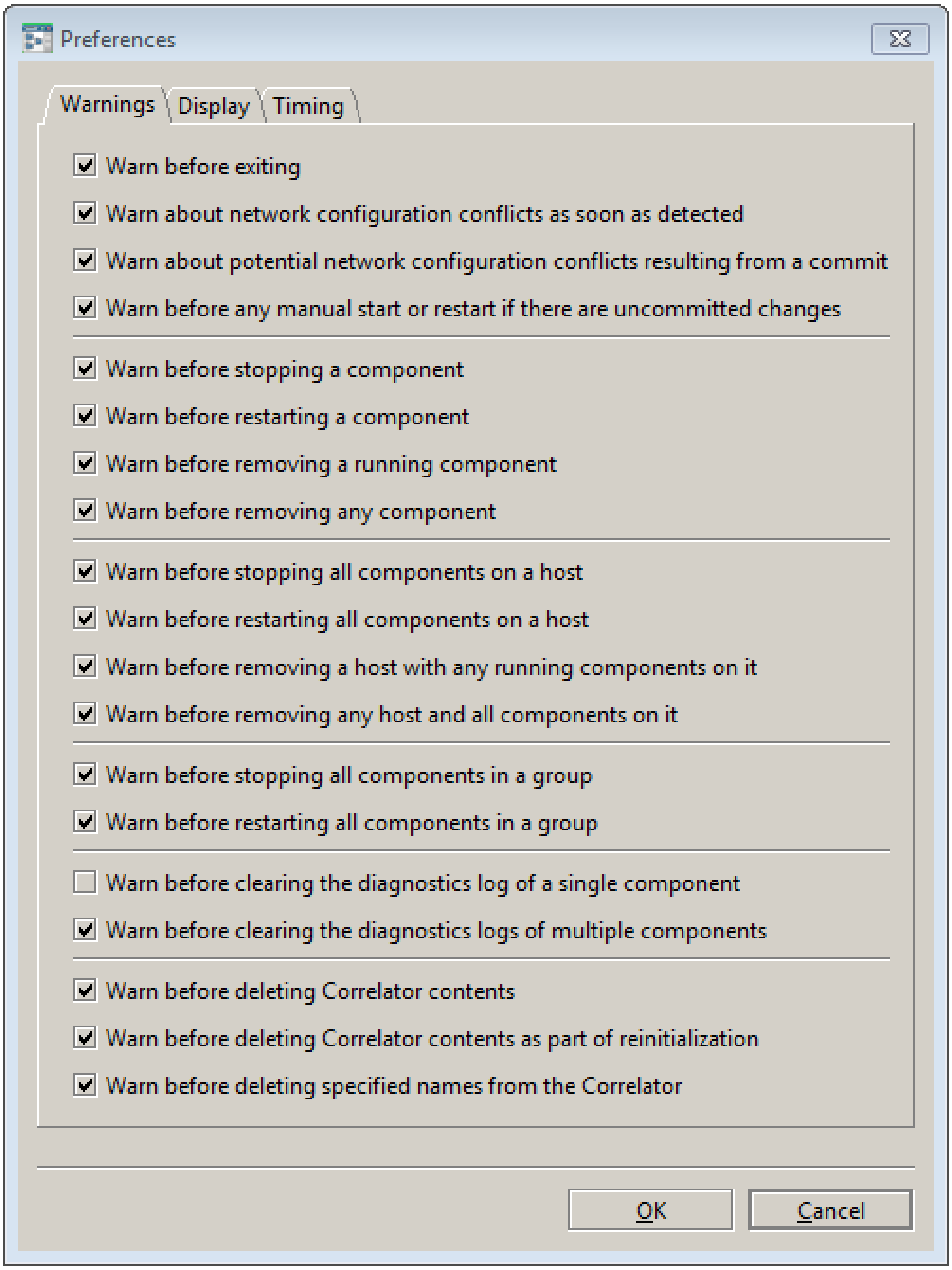
These options are set using the Preferences dialog. This can be accessed by selecting the Preferences... option from the Options menu on the EMM menu.

The dialog has three tabs: Warnings, Display, and Timing.

Using the Management and Monitoring Console

Warnings

The illustration below shows the Warning tab.



☑ Warn before exiting

☑ Warn about network configuration conflicts as soon as detected

☑ Warn about potential network configuration conflicts resulting from a commit

☑ Warn before any manual start or restart if there are uncommitted changes

☑ Warn before stopping a component

☑ Warn before restarting a component

☑ Warn before removing a running component

☑ Warn before removing any component

☑ Warn before stopping all components on a host

☑ Warn before restarting all components on a host

☑ Warn before removing a host with any running components on it

☑ Warn before removing any host and all components on it

☑ Warn before stopping all components in a group

☑ Warn before restarting all components in a group

☐ Warn before clearing the diagnostics log of a single component

☑ Warn before clearing the diagnostics logs of multiple components

☑ Warn before deleting Correlator contents

☑ Warn before deleting Correlator contents as part of reinitialization

☑ Warn before deleting specified names from the Correlator

The Warning tab allows various confirmation dialogs to be turned off if desired.

Miscellaneous warnings:

- `Warn before exiting` – Ask for confirmation when the user asks to shut down EMM. Although all settings in EMM are preserved on shutdown and re-read the next time it is restarted, the background monitoring and automatic restart functionality only works while EMM is running.
- `Warn about network configuration conflicts as soon as detected` – Warn if a component conflict (same host-port combination) is detected.

- Warn about potential network configuration conflicts resulting from a commit – Ask for confirmation if a component conflict (same host-port combination) would result from a Commit Changes operation.
- Warn before any manual start or restart if there are uncommitted changes – Provide a warning when a component is started or restarted from EMM and changes have been made to the component's temporary working configuration that will be ignored when the component is started, because they have not yet been committed.

Component operation warnings:

- Warn before stopping a component – Ask for confirmation when the user tries to stop a component.
- Warn before restarting a component – Ask for confirmation when the user tries to restart a component.
- Warn before removing a running component – Ask for confirmation when the user tries to remove a component that is running, and provide the option of removing without stopping the component.
- Warn before removing any component – Always ask for confirmation when the user tries to remove a component, whether it is running or not.

Host operation warnings:

- Warn before stopping all components on a host – Ask for confirmation when the user tries to stop all components on a specific host.
- Warn before restarting all components on a host – Ask for confirmation when the user tries to restart all components on a specific host.
- Warn before removing a host with any running components on it – Ask for confirmation when the user tries asks to remove a host with any components on it that are known to be running; this provides the option of removing the host without stopping running components.
- Warn before removing any host and all components on it – Always ask for confirmation when the user tries asks to remove a host, whether it contains running components or not.

Group operation warnings:

- Warn before stopping all components in a group – Ask for confirmation when the user tries to stop all components in the Apama Managed Components group.
- Warn before restarting all components in a group – Ask for confirmation when the user tries to restart all components in the Apama Managed Components group.

Clearing log warnings:

- Warn before clearing the diagnostics log of a single component – Ask for confirmation before clearing the diagnostics log when a single component is selected.
- Warn before clearing the diagnostics log of multiple components – Ask for confirmation before clearing the diagnostics logs for all the components associated with a selected host or group.

Deleting correlator contents warnings:

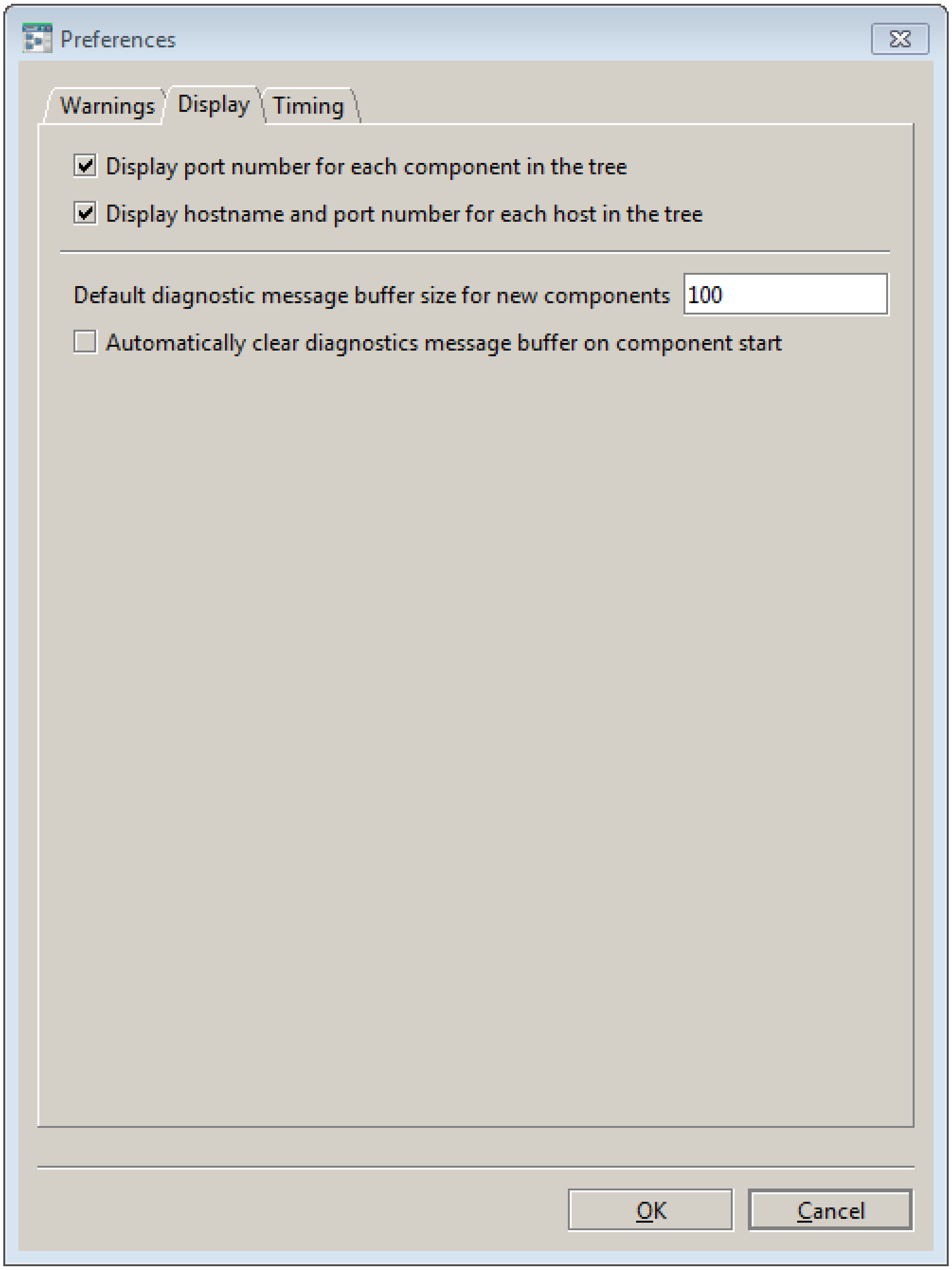
- Warn before deleting Correlator contents – Ask for confirmation before deleting all Apama Event Processing Language (EPL), Correlator Deployment Package (CDP), or Apama Java files in the correlator. The warning will be displayed after clicking the Delete button on the correlator's Inspect tab.

- `Warn before deleting Correlator contents as part of reinitialization` – Ask for confirmation before re-initializing the correlator with the Delete everything loaded in the engine before reinitializing check box is enabled.
- `Warn before deleting specified names from the Correlator` – Ask for confirmation before deleting specific EPL, CDP, or Apama Java files. The warning will be displayed after clicking the Delete button on the correlator's Inspect tab.

Preferences

Display

The illustration below shows the Display tab.



Preferences

Warnings | Display | Timing

☑ Display port number for each component in the tree

☑ Display hostname and port number for each host in the tree

Default diagnostic message buffer size for new components 100

□ Automatically clear diagnostics message buffer on component start

The following settings may be configured from the Display tab of the **Preferences** dialog:

- Display port number for each component in the tree – If enabled, the listening port for each component will be displayed alongside its name in the Navigation Pane. This makes it easier to check for conflicts (since the port number must be unique per host).
- Display hostname and port number for each host in the tree – If enabled, the actual hostname (as opposed to its EMM descriptive name) and the port its Sentinel Agent is listening on are displayed alongside that host's name in the Navigation Pane. This makes it easier to avoid conflicts in host management.
- Default diagnostic message buffer size for new components – Each component has a Diagnostics tab in its Display Pane, containing a list of the most recent status messages for the component. The

maximum number of messages (after which the oldest will be discarded) may be configured on a per-component basis; this Preferences option specifies the default size of the message buffer that is assigned to new components. By default the buffer size is `100` lines. Minimum size is 1 line.

- Automatically clear diagnostics message buffer on component start – If enabled, removes all log messages for a component when it starts. This is useful during development work, but it should not be enabled in a production system.

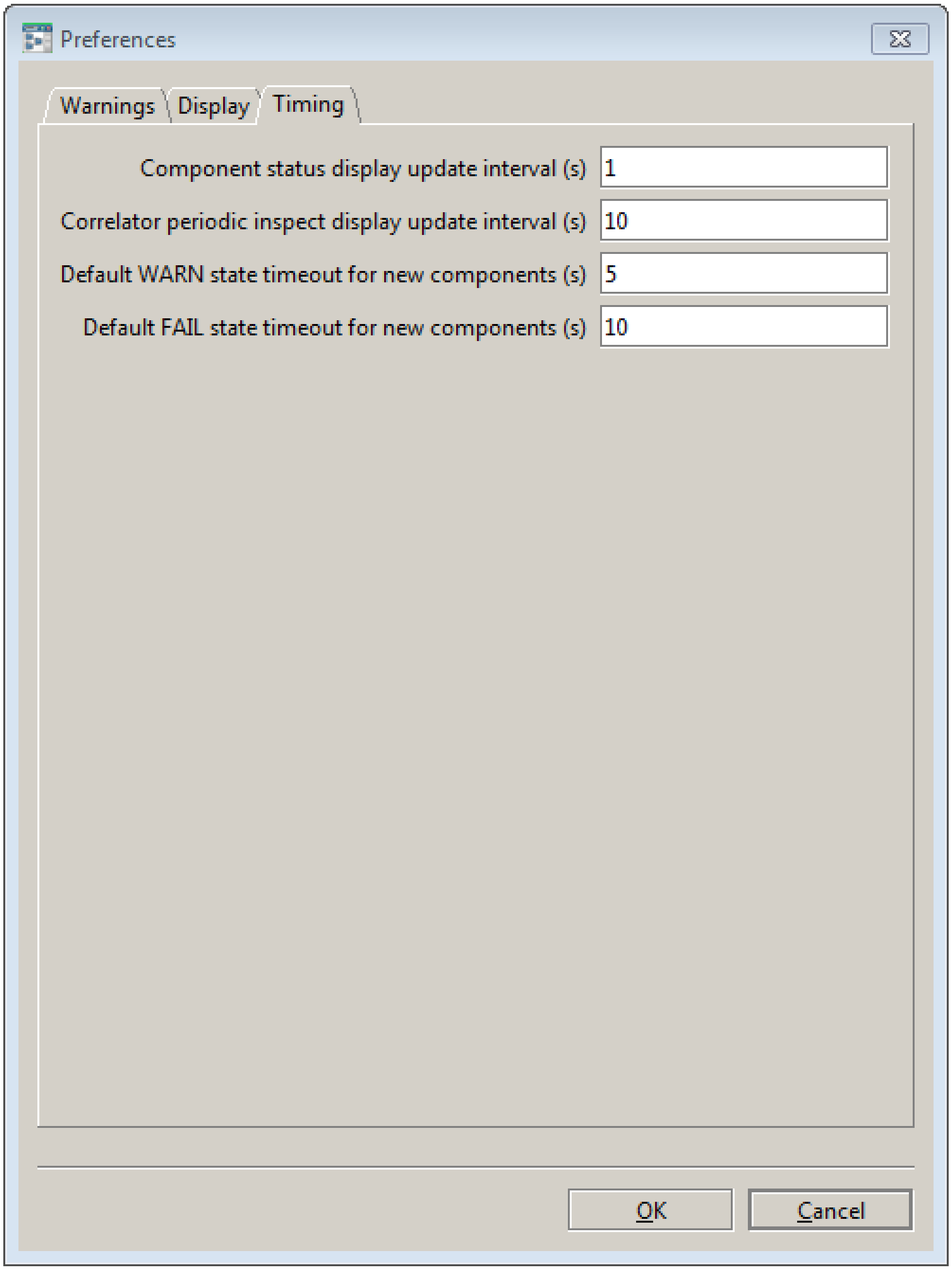
Preferences

Timing

The illustration below shows the Timing tab.

The following settings may be modified from the Timing tab of the **Preferences** dialog:

- Component status display update interval (s) – This parameter specifies how frequently a component's statistics should be updated. By default this is every `1` second.
- Correlator periodic inspect display update interval (s) – This parameter specifies how frequently a correlator's inspect information should be refreshed. By default this is every `10` seconds.
- Default WARN state timeout for new component (s) – This parameter specifies the initial value of `WARN state timeout` used for new components (see "Add correlator" on page 40). By default this is `5` seconds.
- Default FAIL state timeout for new component (s) – This parameter specifies the initial value of `FAIL state timeout` used for new components (see "Add correlator" on page 40). By default this is `10` seconds.



Component status display update interval (s) 1

Correlator periodic inspect display update interval (s) 10

Default WARN state timeout for new components (s) 5

Default FAIL state timeout for new components (s) 10

Preferences

3 Deploying and Configuring Correlators

Apama correlators are the core event processing and correlation engines for Apama applications. This section describes how to start and manage correlators using the EMM console. You can also start and manage correlators using Apama command line utilities; for more information on this, see "Event Correlator Utilities Reference" on page 103.

This topic describes how to use the EMM menu items to carry out operations on Apama correlators. However, all these operations can also be carried out from the context menus in the Navigation Pane, and the most useful are also available on the toolbar.

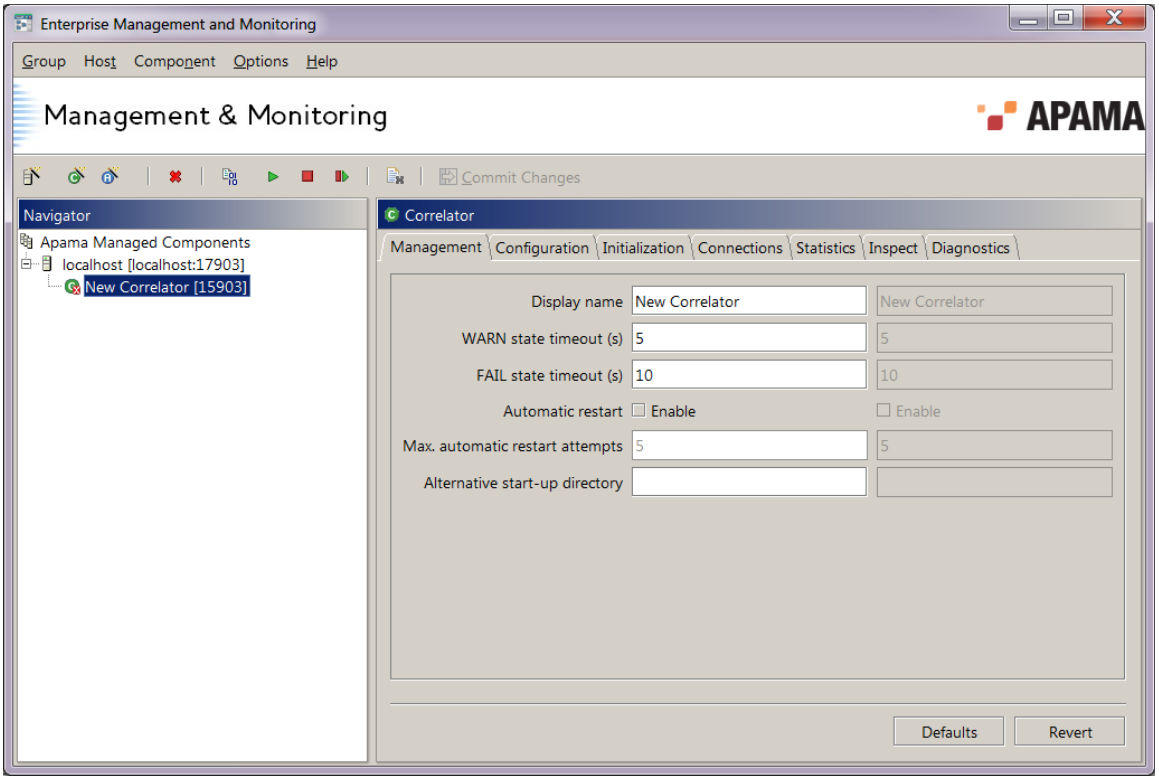
Before operations can be carried out on these components, a Sentinel Agent must be installed and running on the target host, and the EMM console must have been configured to manage that host, as detailed in "Managing hosts" on page 28.

Adding correlators

To define and configure a new correlator:

1. Select the host where you want to add the correlator in the EMM Navigation Pane.
2. Select Component > Add correlator from the EMM menu or click the button on the tool bar

   This adds a correlator to the host in the Navigation Pane, selects it, and displays the Management tab within the Details Pane;



Display name	New Correlator	New Correlator
WARN state timeout (s)	5	5
FAIL state timeout (s)	10	10
Automatic restart	□ Enable	□ Enable
Max. automatic restart attempts	5	5
Alternative start-up directory		

The other tabs available on this Details Pane for a correlator are the Configuration tab, the Initialization tab, the Connections tab, the Statistics tab, the Inspect tab, and the Diagnostics tab. These are discussed in detail in "The correlator tabs" on page 53.

EMM initializes the new correlator with a set of default options, which are normally safe. The default value for the listening port (which has to be unique per host) is automatically selected so as not to conflict with any other known components.

Before the new correlator can be started for this first time you must ‘commit’ its configuration using the Commit Changes (⇨) button. Note that most of the correlator’s configuration options only apply when the component is started up, so changes committed after the component is already running will usually not take effect until it is restarted.

Deploying and Configuring Correlators

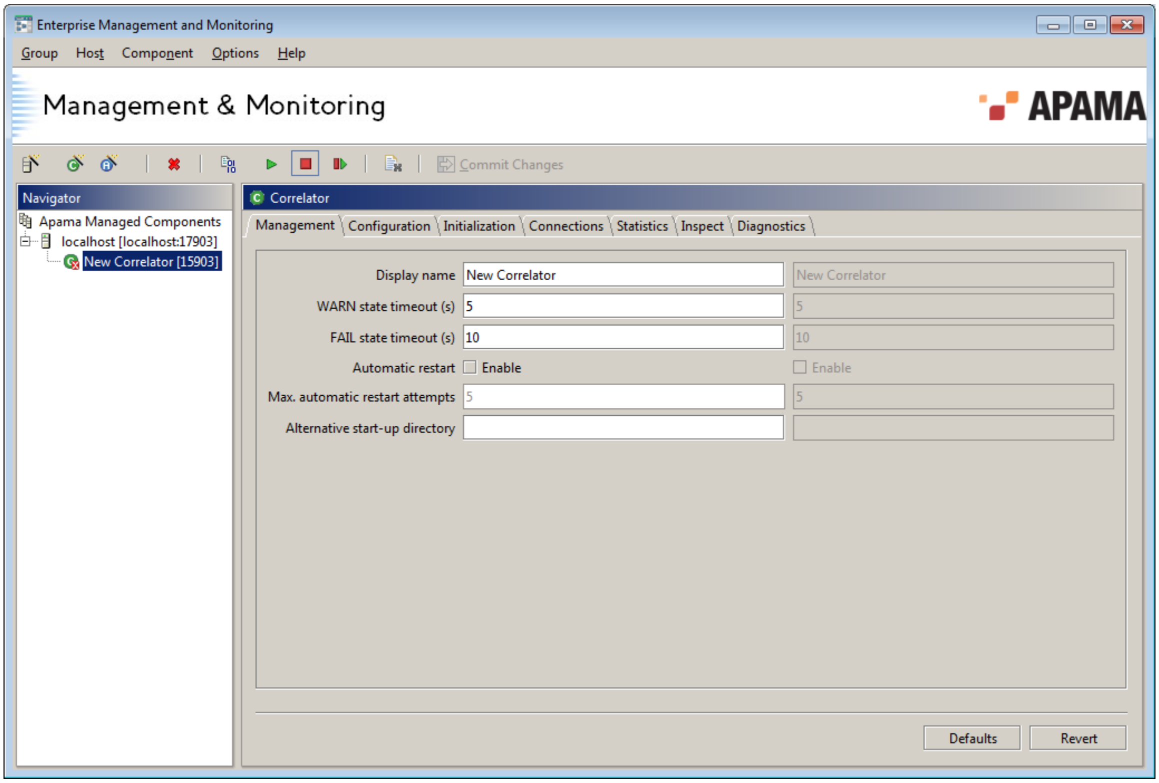
The correlator tabs

When a correlator is selected in the Navigation Pane, the Details Pane displays the following tabs:

- "Management tab" on page 53
- "Configuration tab" on page 55
- "Initialization tab" on page 57
- "Connections tab" on page 60
- "Statistics tab" on page 63
- "Inspect tab" on page 66
- "Diagnostics tab" on page 68

Management tab

This tab contains a number of parameters that are relevant to managing a correlator.



Display name	New Correlator	New Correlator
WARN state timeout (s)	5	5
FAIL state timeout (s)	10	10
Automatic restart	□ Enable	□ Enable
Max. automatic restart attempts	5	5
Alternative start-up directory		

The parameters on the Management tab are:

- `Display name` – This is the name to associate with this correlator in the Navigation Pane. The name can be changed at any time, even if the correlator has already been started.
- `WARN state timeout (s)` – This is the number of seconds to wait before moving the component into the `WARN` state if it is not changing state (stopping/starting/ restarting) as required. This setting can be changed at any time. The default is 5 seconds.
- `FAIL state timeout (s)` – This is the number of seconds to wait after entering the `WARN` state before moving into the `FAIL` state, if the component has still failed to change state as expected. This setting can be changed at any time. The default is 10 seconds.
- `Automatic restart enable` – Tick to configure EMM to monitor the selected component. If it were to stop unexpectedly, EMM would automatically restart it – after waiting the amount of time required for it to enter the `FAIL` state (and subject to the configured limit on the number of restart attempts described below). This setting can be changed at any time.

Note: Component monitoring and auto-restart is carried out by EMM and requires EMM to be running.

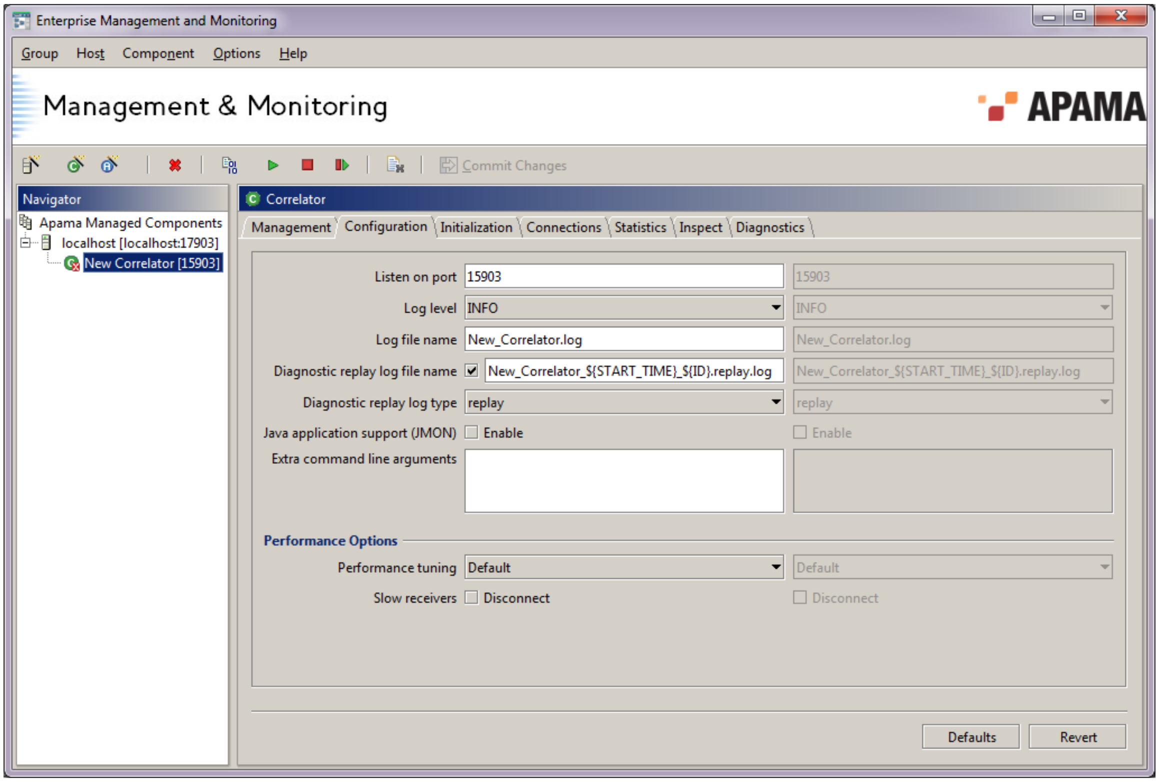
- `Max. automatic restart attempts` – This option is only available if `Automatic restart` is enabled. It specifies the total number of automatic restarts to perform (or attempt) without user intervention. The count is made from when you last enabled `Automatic restart` and committed, or explicitly stopped/started/ restarted the correlator. This setting can be changed at any time.
- `Alternative start-up directory` – By default all components are started from the current working directory of the Sentinel Agent running on the host in question (see "Sentinel Agents" on page 28). If you wish, you can provide an alternative startup folder here.

Click on the Commit Changes button when you are finished customizing the new correlator. This applies these outstanding changes.

The Default button resets all outstanding parameter values to their defaults, while the Revert button reverts the outstanding parameter values to their current committed values.

Configuration tab

This tab contains a number of parameters that configure how a correlator operates. As all of these are startup parameters, you should adjust them before you start the component. If the component is already running they will only take effect the next time it is started.



Listen on port: 15903 | 15903

Log level: INFO | INFO

Log file name: New_Correlator.log | New_Correlator.log

Diagnostic replay log file name: ☑ New_Correlator_${START_TIME}_${ID}.replay.log | New_Correlator_${START_TIME}_${ID}.replay.log

Diagnostic replay log type: replay | replay

Java application support (JMON): □ Enable | □ Enable

Extra command line arguments:

Performance Options

Performance tuning: Default | Default

Slow receivers: □ Disconnect | □ Disconnect

The following parameters are available for a correlator:

- `Listen on port` – Port on which the correlator should listen for monitoring and management requests (default is `15903`). The correlator will fail to start if this port is in use by any other component, or for that matter by any other software that is running on the relevant host.

Note: The highest permitted port number for correlators and other components managed through EMM is 65534.

- `Log level` – Sets the log level the event correlator should log at. This must be one of `OFF`, `CRIT`, `FATAL`, `ERROR`, `WARN`, `INFO`, `DEBUG`, or `TRACE` (in increasing order of verbosity). The default level is `INFO`.
- `Log file name` - Sets the filename that the event correlator should write log messages to (on the file system of the host the correlator is running on). It is recommended that an absolute path be provided. See "Specifying paths and filenames in the Details Pane" on page 43 for more information about setting filename options correctly.

- `Diagnostic replay log file name` - Enables and sets the filename for a diagnostic replay log. A diagnostic replay log collects all messages sent to the correlator and writes then to a file on the file system of the host the correlator is running on. The replay log provides useful input to help Apama technical support diagnose problems with a correlator or an application running on the correlator.

It is recommended that an absolute path be provided. See "Specifying paths and filenames in the Details Pane" on page 43 for more information about setting filename options correctly. For more information on diagnostic replay log files, see "Injecting code into a correlator" on page 119.

- `Diagnostic replay log type` - Specifies whether the correlator will use a full replay log or an input-only replay log. The default is to use a full replay log. The differences are:
  - A full replay log lets you replay the exact same behavior, ordering of emit operations, and so on, as the run that generated the replay log.
  - An input-only replay log contains only the external inputs to the correlator. There is no information about multi-context ordering. Consequently, if there is more than one context, there is no guarantee that you can replay execution in the same order as the original execution.
  - For more information on the replay log types, see "Comparison of full replay logs and input-only replay logs" on page 158.
- `Java application support (JMON)` – Enables support for Apama Java applications. If this is not set, any attempt to inject a Java application either using `engine_inject -j` or by adding a `.jar` file `Inject` initialization action will result in an error. The correlator’s performance is improved when Java application support is disabled.
- `Extra command line arguments` – This option allows additional unspecified command line arguments to be passed to the correlator. For example these might be special options to pass to the embedded JVM used for Apama Java applications, or special settings provided to you by Apama Customer Support to address specific issues.
- `Performance tuning` – Specifies whether the correlator should be configured to favor Low Latency or High Throughput.
  - `Low latency` – The transport waits until an event is on the correlator’s queue. The transport then sends, in one batch, as many events as are available on the queue. Because of thread scheduling effects, this can be more than the event that woke up the transport. This is the default behavior for multiprocessor machines.
  - `High throughput` – The transport waits 10 milliseconds before it tries to send any events. Typically, during that time, the correlator emits enough events to its output queue to form an efficient batch. This is the default behavior for uniprocessor machines.

For more information about this option, see "About the low latency and high throughput options" on page 113.

- `Slow receivers` – Tells the event correlator to disconnect any receiver whose output queue becomes full; that is, a receiver that cannot consume events fast enough. Without this option, the event correlator would eventually stop processing events altogether, (and block) until some output events are consumed by the receiver, freeing space on its output queue.

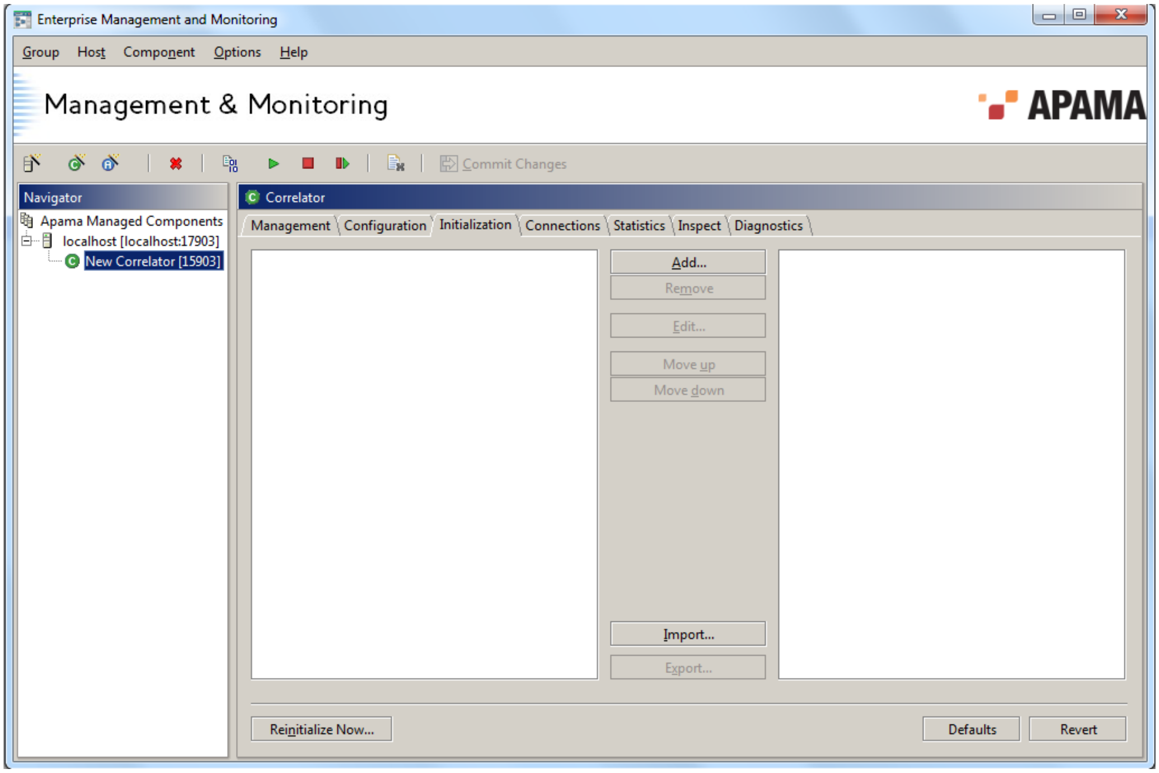
Click on the Commit Changes button when you are finished customizing the new correlator. This applies these outstanding changes for use when the correlator is actually started.

The Default button resets all outstanding parameter values to their defaults, while the Revert button reverts the outstanding parameter values to their current committed values.

Initialization tab

This tab allows you to specify the monitors and events that will be automatically executed whenever the component is started by EMM. The files you specify typically have `.mon`, `.evt`, and `.jar` extensions. Note that if the component is started outside EMM and then added to EMM for management, any initialization actions are only executed the next time it is started through EMM.

Note: By default, EMM uses the default encoding of the system on which EMM is running to read Apama Event Processing Language files (`.mon`) and event (`.evt`) files. If you want to inject UTF-8 encoded files, ensure that you use an editor that adds a Byte Order Mark (BOM) at the start of the file. For example, on Windows, the Notepad editor inserts a BOM when you save a file.

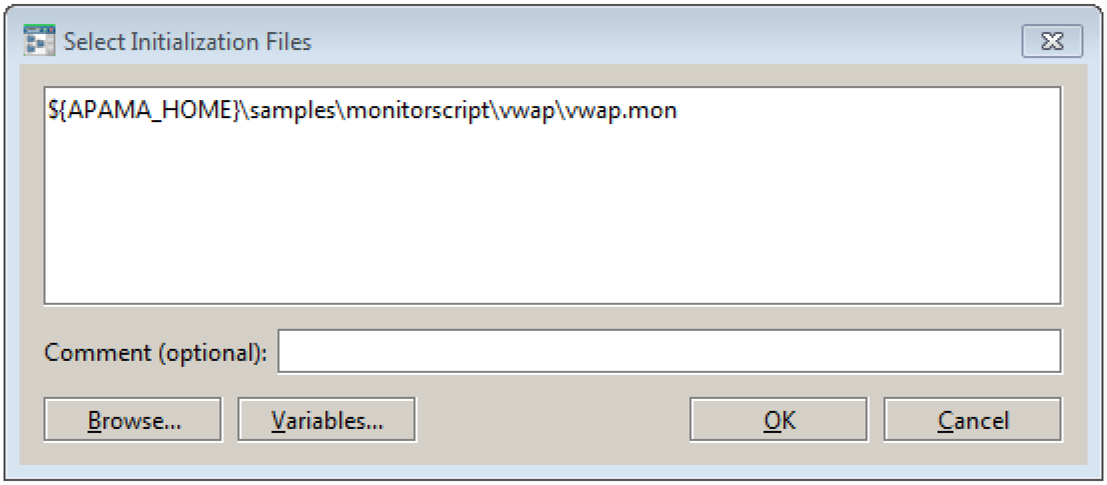


Adding initialization actions

To add initialization actions:

1. On the Initialization tab, click the Add button.

   This displays the **Select Initialization Files** dialog.



${APAMA_HOME}\samples\monitorscript\vwap\vwap.mon

2. Enter the name of the monitor or event file to use for initializing the component. You can also click Browse to navigate to the directory containing the files to use for initializing the component and select the files. Use Ctrl and Shift to select more than one file name. You can add a comment that will be displayed next to the file name on the Initialization tab.

3. Click Variables to select an Apama variable; this alleviates the need to type a full path name. You can also use your own (non-Apama) environment variables by specifying them in the form: `${env_var:foo}`.

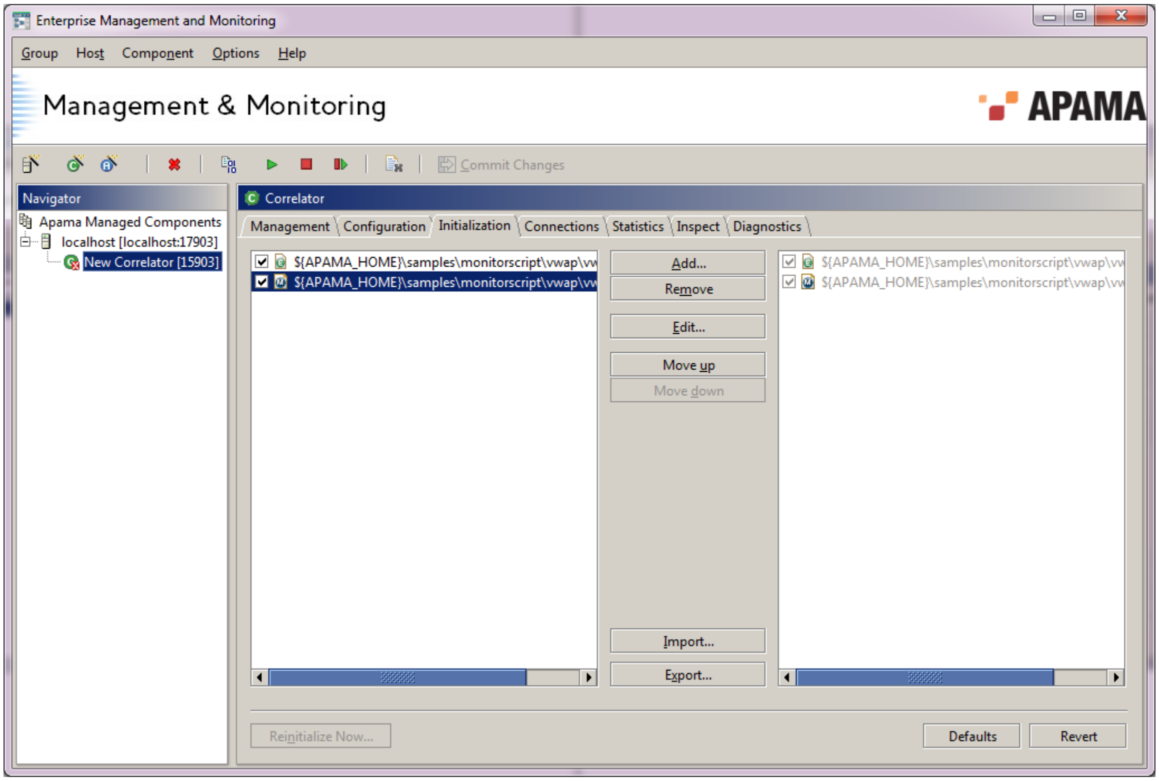
4. When you have specified the name of the file(s) you want to add, click OK.

5. You can also specify a file with a `.txt` extension containing a list of monitors and event files with which to initialize the component. Click Import and enter the name of the file that contains the initialization files. Files for this purpose are formatted with the name of each `.mon`, `.evt`, or `.jar` file on a separate line. Apama Studio exports initialization files in this format.

6. Instead of entering or selecting file names as in Step 2, you can drag and drop files from a Windows Explorer window to the left hand list of the Initialization tab.

   All filenames specified here will be accessed on the local computer on which the EMM console is running – not the machine hosting the correlator (unless of course the correlator is also running on the local machine).

7. After the previous steps, EMM displays the initialization files in the left hand list. The check boxes in the left hand list allow you to select which initialization actions will take effect; actions with unchecked boxes will be ignored. Click the Commit button to apply the changes; this will display the initialization files in the right hand list. The initialization actions will take place the next time the component starts

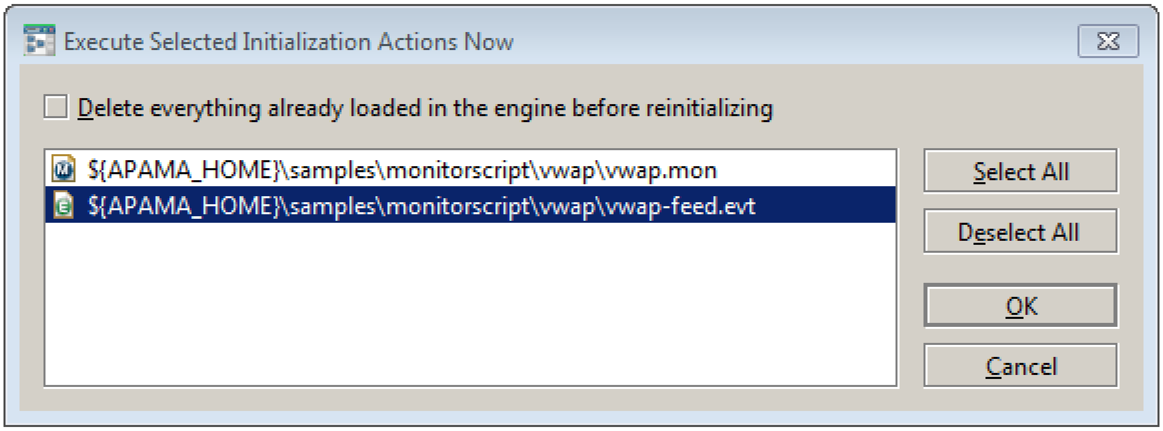


☑ ${APAMA_HOME}\samples\monitorscript\vwap\vw
☑ ${APAMA_HOME}\samples\monitorscript\vwap\vw

☑ ${APAMA_HOME}\samples\monitorscript\vwap\vw
☑ ${APAMA_HOME}\samples\monitorscript\vwap\vw

When an action is selected, the following buttons are also available:

- Remove – Remove the selected initialization action.
- Edit – Allows the comment and/or filename associated with an initialization action to be changed.
- Move up – Move the selected initialization action up in the list. The order is significant because the actions will be executed in the order they appear in the list (from top to bottom).
- Move down – Move the selected initialization action down the list.
- State restore — Restore the entire runtime state of the correlator from a state image file or dump. Due to the nature of Restore operations, only one Restore initialization action may be specified for each correlator, and it will always be executed before any other Initialization action.
- Import – Allows you to specify a text file containing initialization actions.
- Export – Saves the component’s initialization actions to a text file.
- Reinitialize Now... — Initializes the Component with the initialization files specified. When you click Reinitialize Now..., EMM displays the **Execute Selected Initialization Actions Now** dialog.



Execute Selected Initialization Actions Now

☐ Delete everything already loaded in the engine before reinitializing

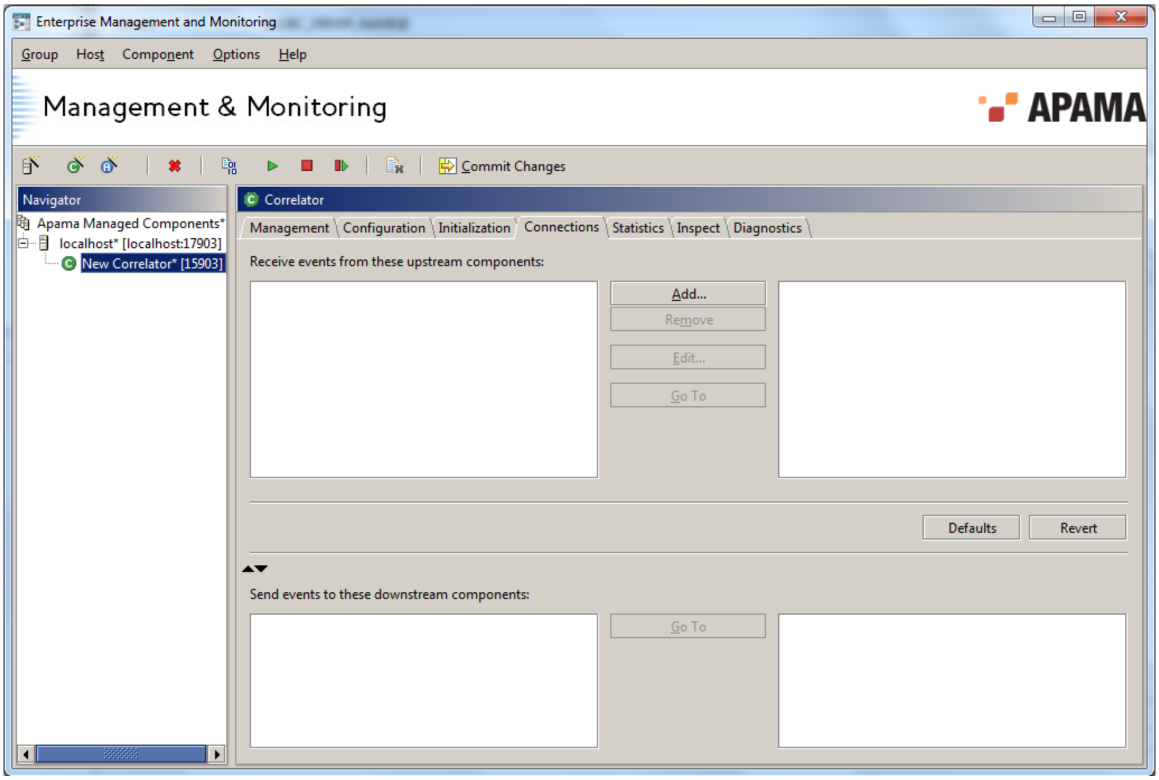
- ${APAMA_HOME}\samples\monitorscript\vwap\vwap.mon
- ${APAMA_HOME}\samples\monitorscript\vwap\vwap-feed.evt

This dialog allows you to specify which actions you want to use and also gives you the opportunity to delete any Apama EPL or Apama Java files loaded in the correlator before you reinitialize it.

- Defaults — Removes any initialization files that have been added to the component.
- Revert — Removes initialization files that have been added to the component since the last time changes were Committed. Initialization files added prior to the last Commit are retained.

Connections tab

This tab displays any other components that are connected to this component in both upstream and downstream directions. “Upstream” components are those components that this component receive events from; “downstream” components are those that this component sends events to. Because the “owner” of the connection is always the downstream component, the list of downstream connections is for display only.

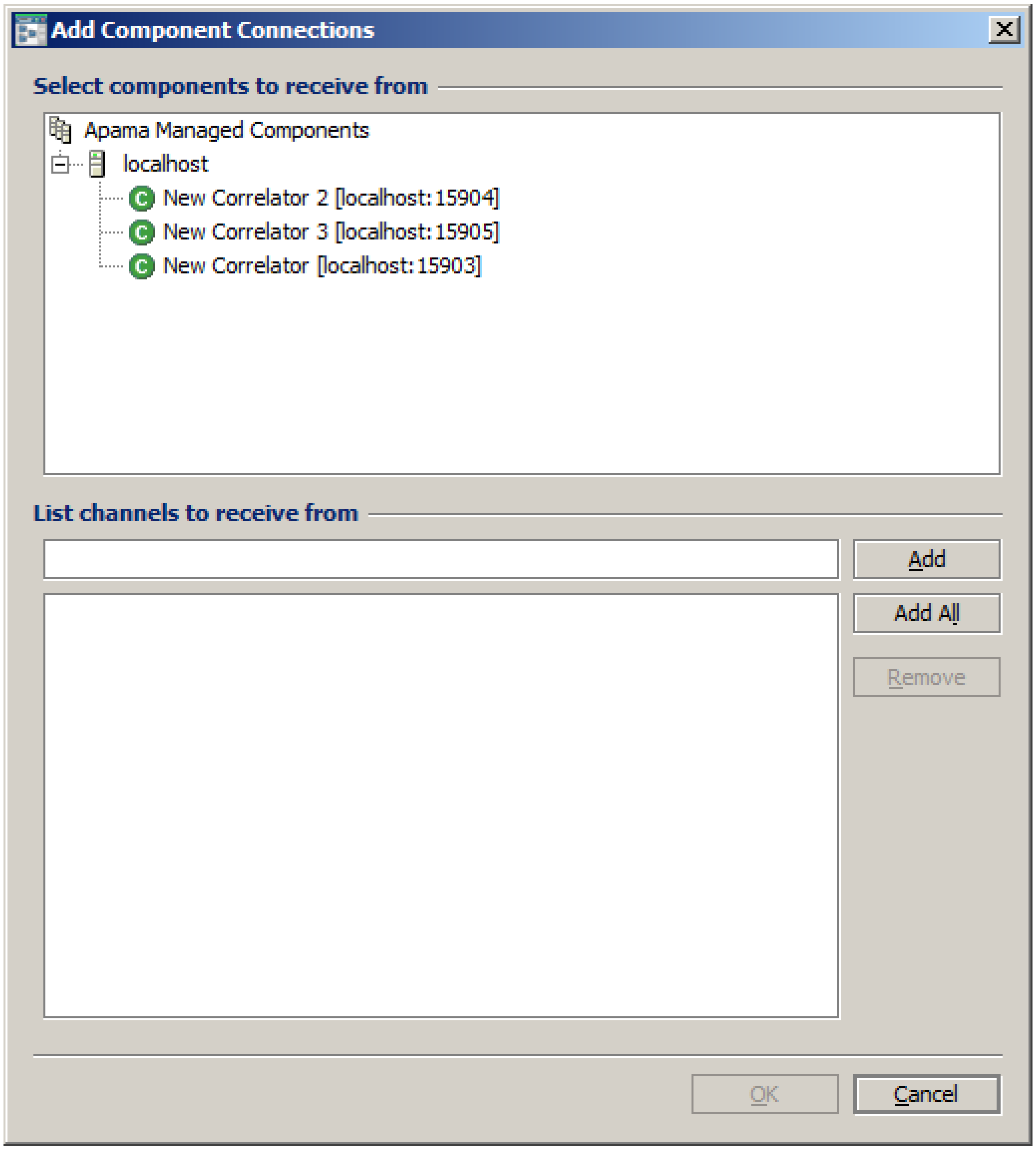


Adding new upstream connections

The Connections tab allows you to add new upstream component connections.

To specify upstream components you want to connect to this component:

1. Click Add to display the **Add Component Connections** dialog.

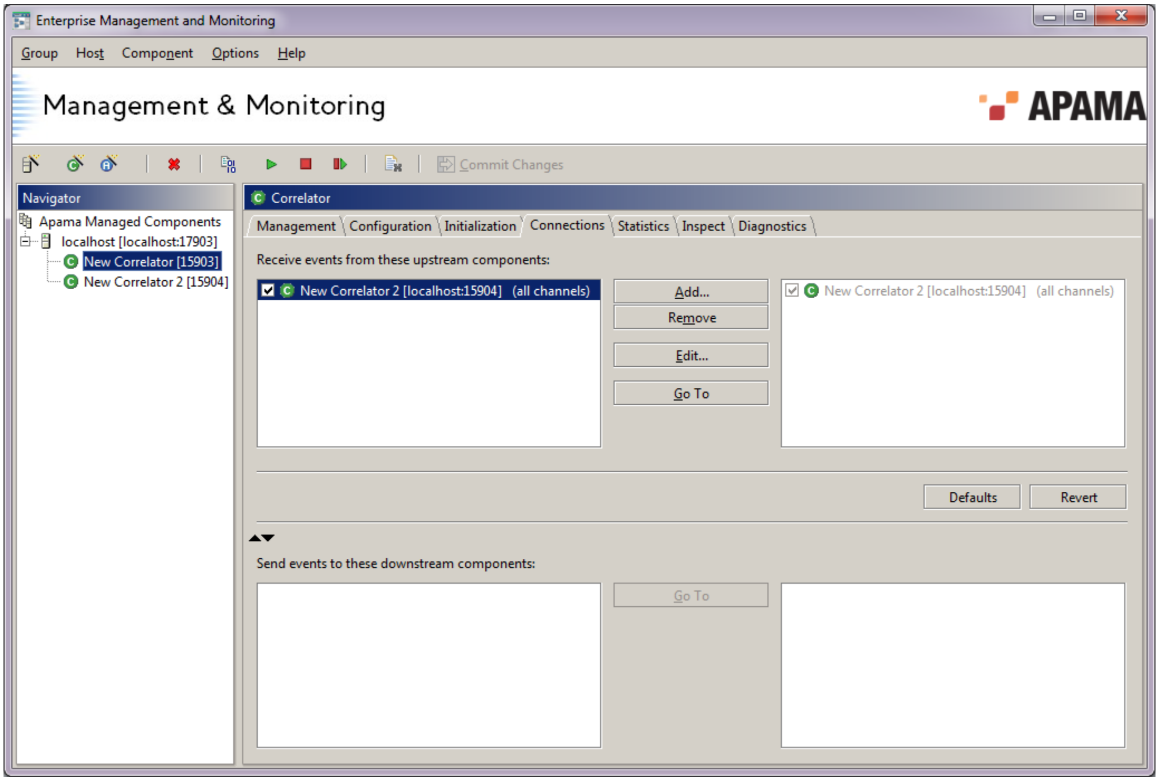

Add Component Connections
Select components to receive from
Apama Managed Components
localhost
New Correlator 2 [localhost:15904]
New Correlator 3 [localhost:15905]
New Correlator [localhost:15903]
List channels to receive from
Add
Add All
Remove
OK
Cancel

2. From the Select components you want to receive from list, select one or more components.
3. In the List channels to receive from field, enter the name of one or more channels associated with the component you selected in Step 2 and click Add, or click Add All to receive all channels associated with a component.
4. Click OK. The Connections tab will display the components you added in the left hand list.

   The check boxes in the left hand list of the upstream connections allow you select which connections will take effect in the next step. Connections with checked boxes will be enabled; those with unchecked boxes will be disabled.

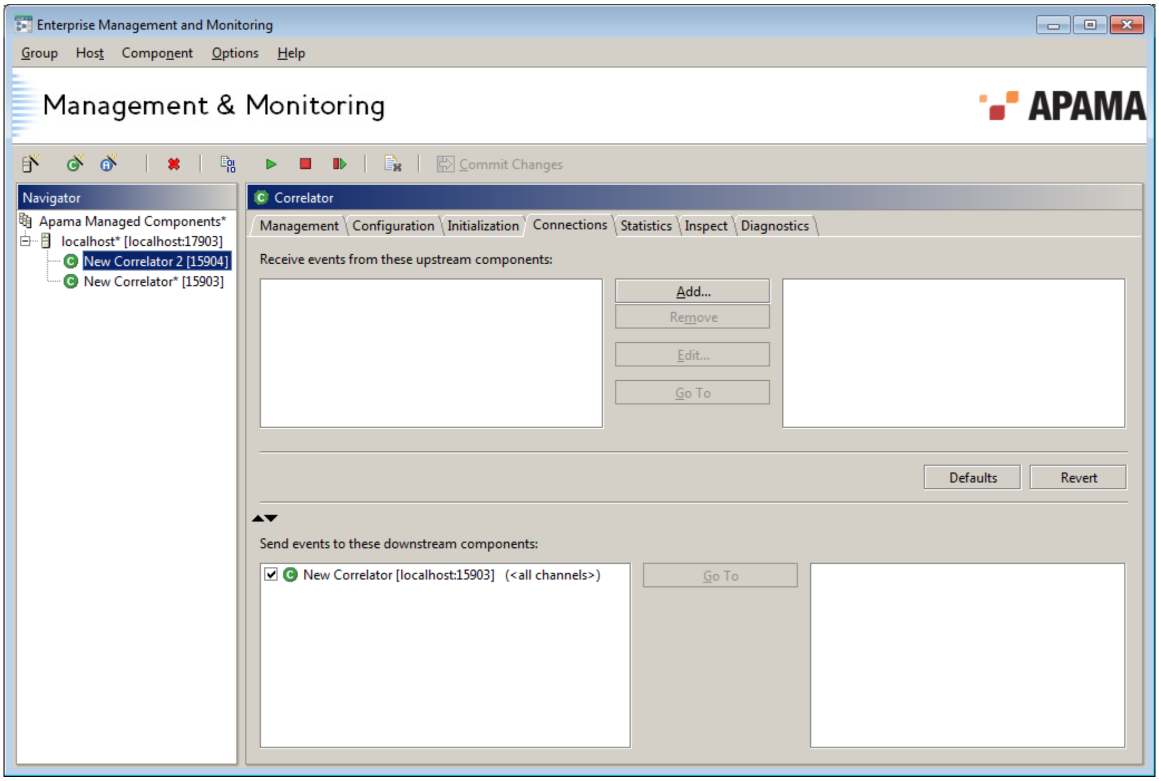
5. Select the Commit changes menu item or the click the Commit Changes ( ) button to make your changes take effect.

   This adds the connected components to the right hand list of the Connections tab.


Enterprise Management and Monitoring
Group Host Component Options Help
Management & Monitoring
APAMA
Commit Changes
Navigator
Apama Managed Components
localhost [localhost:17903]
New Correlator [15903]
New Correlator 2 [15904]
Correlator
Management
Configuration
Initialization
Connections
Statistics
Inspect
Diagnostics
Receive events from these upstream components:
☑ New Correlator 2 [localhost:15904] (all channels)
Add...
Remove
Edit...
Go To
☑ New Correlator 2 [localhost:15904] (all channels)
Defaults
Revert
Send events to these downstream components:
Go To

Downstream connections

Downstream components connected to this component are displayed in the Send events to these downstream components field. The following ilulustration shows a correlator with a downstream connection.



If you select an upstream or downstream component and click the corresponding Go To button, EMM switches the display to show the Connections tab for the connected component.

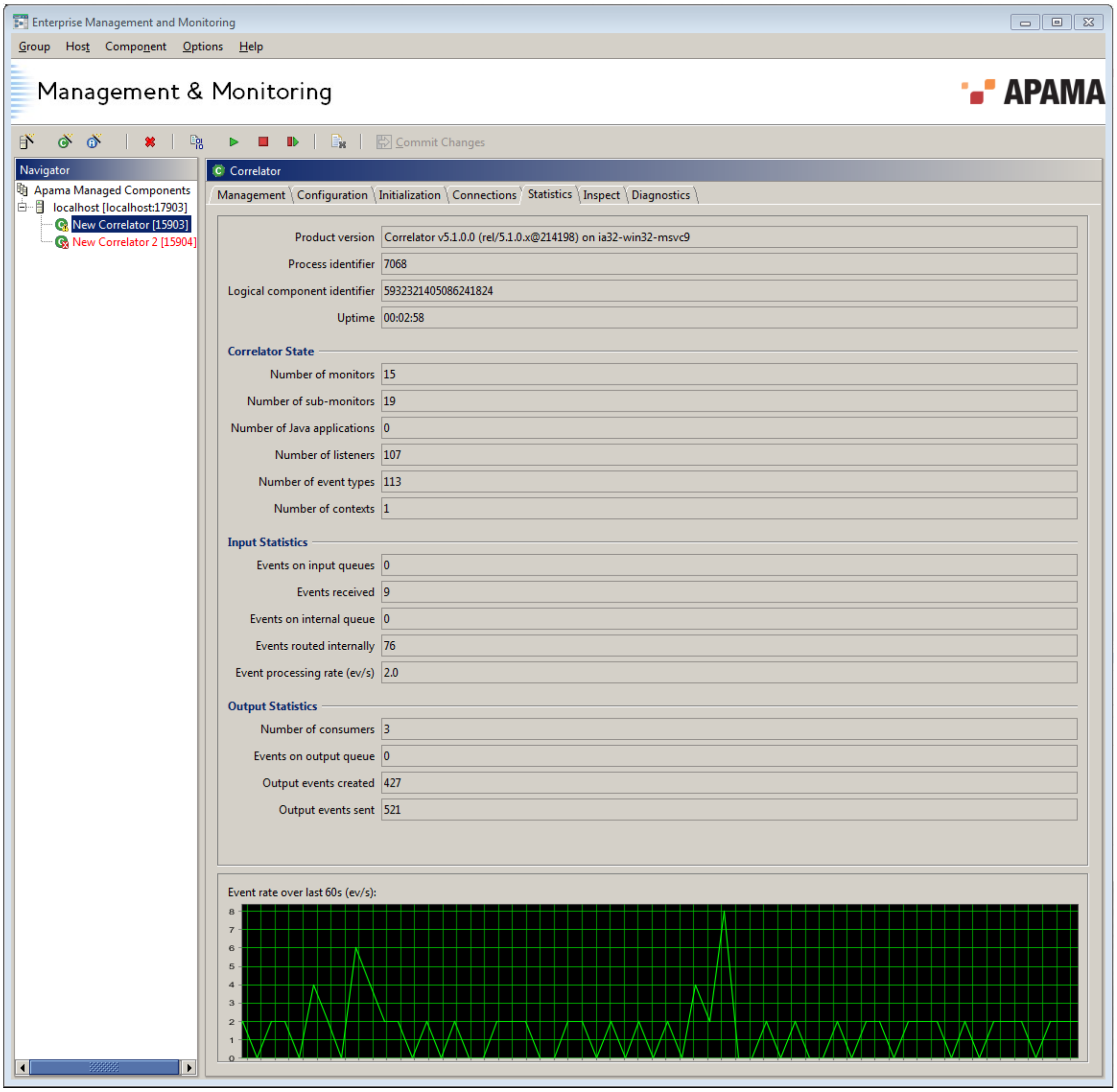
Statistics tab

This tab allows you to monitor of the status of an operational correlator.

Status information is refreshed regularly, by default, once a second. You can change this rate by changing the `Component status display update interval (s)` value on the Timing tab within the **Preferences** dialog. The **Preferences** dialog can be accessed from the Preferences... option from the Options menu on the Menubar.

No status information will be displayed if EMM cannot communicate with the selected correlator. If this is the case it is usually because the correlator has not yet been started or has been stopped. However, it could also be due to network failure. In this situation, statistics will reappear once the connection is restored.

You may need to scroll the display to view all the information on the Statistics tab. The following illustration shows the Statistics tab for an active correlator.



Product version Correlator v5.1.0.0 (rel/5.1.0.x@214198) on ia32-win32-msvc9

Process identifier 7068

Logical component identifier 5932321405086241824

Uptime 00:02:58

Correlator State

Number of monitors 15

Number of sub-monitors 19

Number of Java applications 0

Number of listeners 107

Number of event types 113

Number of contexts 1

Input Statistics

Events on input queues 0

Events received 9

Events on internal queue 0

Events routed internally 76

Event processing rate (ev/s) 2.0

Output Statistics

Number of consumers 3

Events on output queue 0

Output events created 427

Output events sent 521

The Statistics tab displays the following information:

- `Product version` — The Apama release number.
- `Process identifier` — The identifier assigned to this component by the operating system.
- `Logical component identifier` – A numeric identifier for this component. This is useful for matching components with information contained in log files.
- `Uptime(s)` – The time in seconds since this correlator was started. This time is maintained and reported by the component itself, so if the correlator was started independently of EMM and only managed by EMM later, the value would still be accurate.
- `Number of monitors` – The current number of Apama monitors injected and instantiated inside the correlator. This figure changes upwards and downwards as monitors are injected, deleted or just expire.
- `Number of sub-monitors` – The number of Apama sub-monitors. Sub-monitors are created by ‘spawn’ actions within the monitor EPL code. This figure changes upwards and downwards as sub-monitors are spawned, killed or just expire.

- `Number of Java applications` – The number of Apama Java applications currently loaded into the correlator. Apama Java applications do not expire, so this value only decreases when they are explicitly unloaded.
- `Number of listeners` – The number of event listeners created by monitors and sub-monitors in EPL code and by Apama Java applications.
- `Number of event types` – The total number of event types defined within the correlator. This figure decreases when event types are deleted from the correlator.
- `Number of contexts` – The number of contexts in the correlator. This includes the main context plus any created contexts.
- `Events on input queue` – Across all contexts, the total number of events on input queues.
- `Events received` – The total number of events ever received by the correlator. This value is preserved by a checkpoint, so if the state of the correlator is restored from a checkpoint file it will reflect the total number of the events seen by the correlator from which the checkpoint was originally made.
- `Events on internal queue` – Across all contexts, the total number of routed events waiting to be processed. The internal routing queue is a high priority queue that is used when events are internally routed by the `route` instruction in EPL code or in Apama Java applications. For each context, the correlator processes events on the internal queue before processing other events on the input queue.
- `Events routed internally` – Across all contexts, the total number of events that have veen routed since the correlator was started. This value is preserved by a checkpoint, so if the state of the correlator is restored from a checkpoint file it will reflect the total number of the events routed by the correlator from which the checkpoint was originally made.
- `Event processing rate (ev/s)` – The number of events per second currently being processed by the correlator. This value is computed with every status refresh and is only an approximation.
- `Number of consumers` – The number of event consumers registered with the correlator to receive events emitted by it.
- `Events on output queue` – The number of events waiting on the output queue to be dispatched to any registered event consumers.
- `Output events created` – The total number of output events created by the correlator. This value is preserved by a checkpoint, so if the state of the correlator is restored from a checkpoint file it will reflect the total number of output events created by the correlator from which the checkpoint was originally made.
- `Output events sent` – The total number of output events dispatched to event consumers by the correlator. This figure varies from the preceding statistic as an output event might be sent to multiple event consumers. This value is preserved by a checkpoint, so if the state of the correlator is restored from a checkpoint file it will reflect the total number of output events sent out by the correlator from which the checkpoint was originally made.

A graph showing the values taken by the `Event rate` statistic over the last 60 seconds is displayed below the figures.

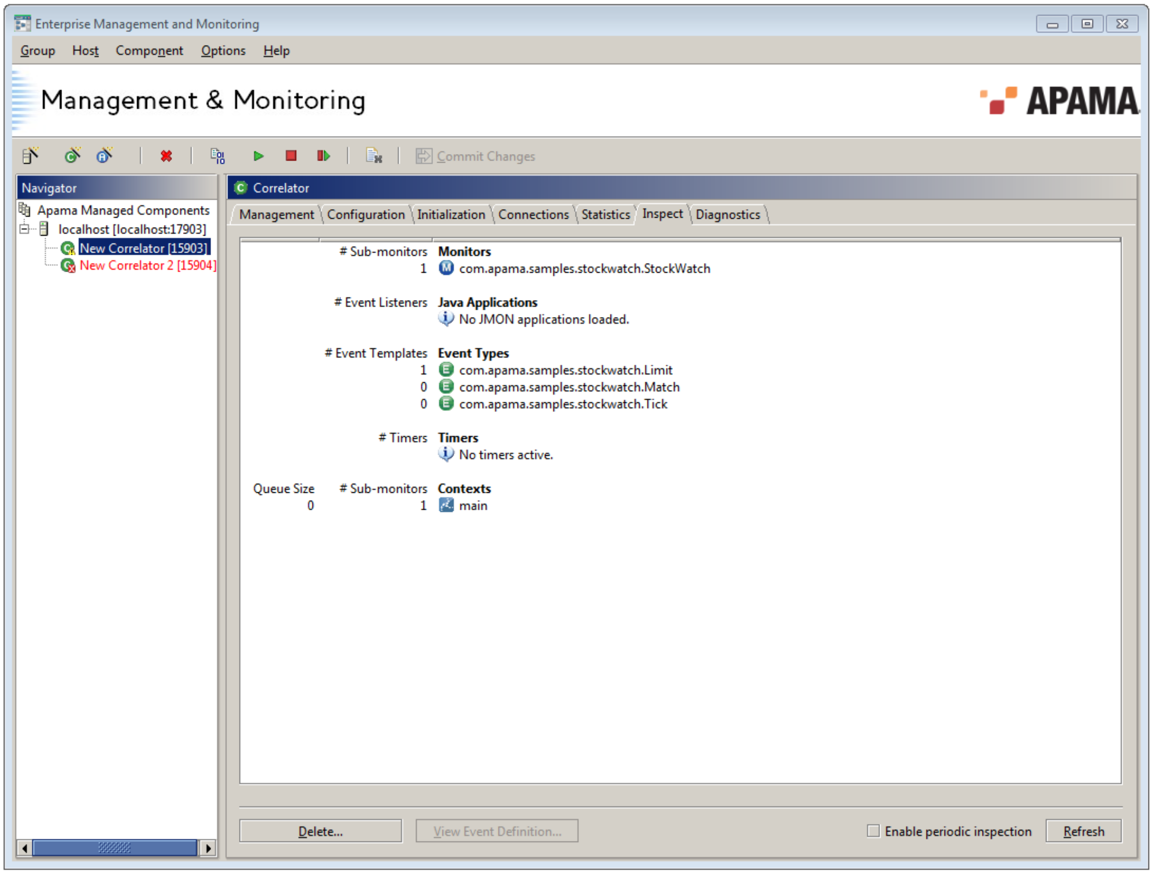
Inspect tab

This tab displays the EPL monitors, Java applications, and Event Types currently active within the correlator.

The information on the Inspect tab is retrieved in real time from the selected correlator when you actually select the Inspect tab, and it might take a few seconds to display.

You can configure the Inspect tab to refresh periodically by ticking the Enable periodic inspection checkbox at the bottom-left corner of the tab. By default the information will be refreshed every 10 seconds. You can change this value by changing the `Correlator periodic inspect display update interval(s)` value on the Timing tab within the **Preferences** dialog. The **Preferences** dialog can be accessed from the Preferences... option from the Options menu on the Menu bar

Alternatively you can refresh manually by clicking the Refresh button at the bottom-right corner. The following illustration depicts the Inspect tab for a correlator showing active definitions



Sub-monitors | Monitors
1 | com.apama.samples.stockwatch.StockWatch

Event Listeners | Java Applications
No JMON applications loaded.

Event Templates | Event Types
1 | com.apama.samples.stockwatch.Limit
0 | com.apama.samples.stockwatch.Match
0 | com.apama.samples.stockwatch.Tick

Timers | Timers
No timers active.

Queue Size | # Sub-monitors | Contexts
0 | 1 | main

□ Enable periodic inspection

The information displayed is organized into the following groups: Monitors, Java Applications, Event Types, and Contexts.

- Monitors — Displays a listing of all the EPL monitors active in the correlator and the number of spawned sub-monitors (if any) for each one.
- Java Applications — Displays a listing of all the Apama Java applications loaded into the correlator. Against each one is also displayed the number of active listeners created by that application.

- Event Types — Displays a listing of all the event types the correlator knows about. Against each event type will also be displayed the number of event templates of that type in use at present in active listeners.
- Timers — Displays the current EPL timers active within the system. The four types of timers which may be displayed here are `wait`, `within`, `at`, and `stream`. The `stream` timers are those set up to support the operation of stream networks.
- Contexts — Display the names of the contexts in the correlator and for each correlator, the number of monitor instances (sub-monitors) and the queue size (number of events on the input queue) of each context.

The Inspect tab contains two other buttons:

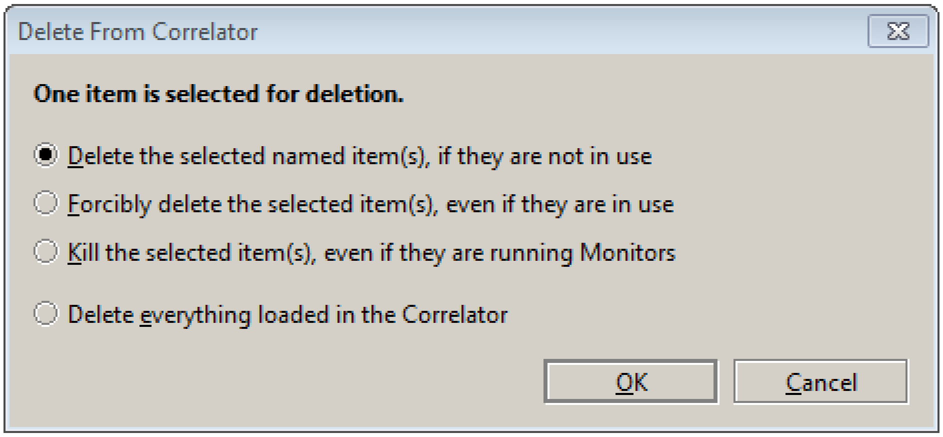
- Delete — Delete an item or items from the correlator.
- View Event Definition — View the code for the definition of a selected Event Type.

Deleting Monitors, Event Types, or Java applications from a correlator

To delete a Monitor, Event Type, or Java application from the correlator:

1. In the Inspect tab, select the item or items you want to delete.
2. Click Delete.

   This displays the **Delete from Correlator** confirmation dialog.
3. Select one of the following choices and click OK.
   - Delete the selected named item(s), if they are not in use
   - Forcibly delete the selected item(s), even if they are in use
   - Kill the selected item(s), even if they are running Monitors
   - Delete everything loaded in the correlator



One item is selected for deletion.

● Delete the selected named item(s), if they are not in use

○ Forcibly delete the selected item(s), even if they are in use

○ Kill the selected item(s), even if they are running Monitors

○ Delete everything loaded in the Correlator

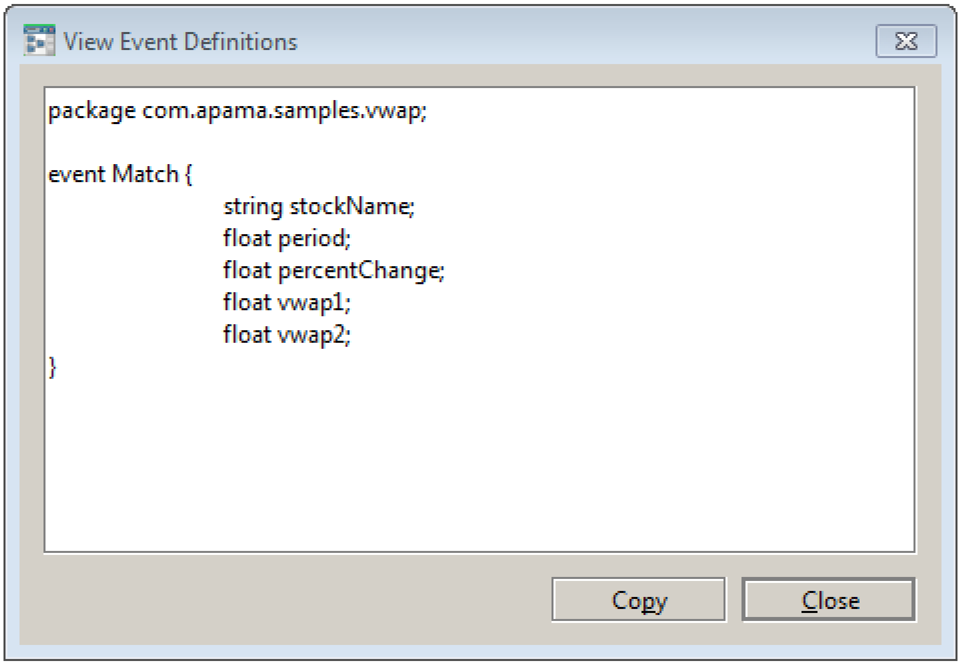
4. Select your choice and click OK.

Displaying event type definition for multiple Event Types

To display the event type definition for one or more Event Types:

1. In the Inspect tab, select the Event Type or Event Types you are interested in.
2. Click View Event Definition.

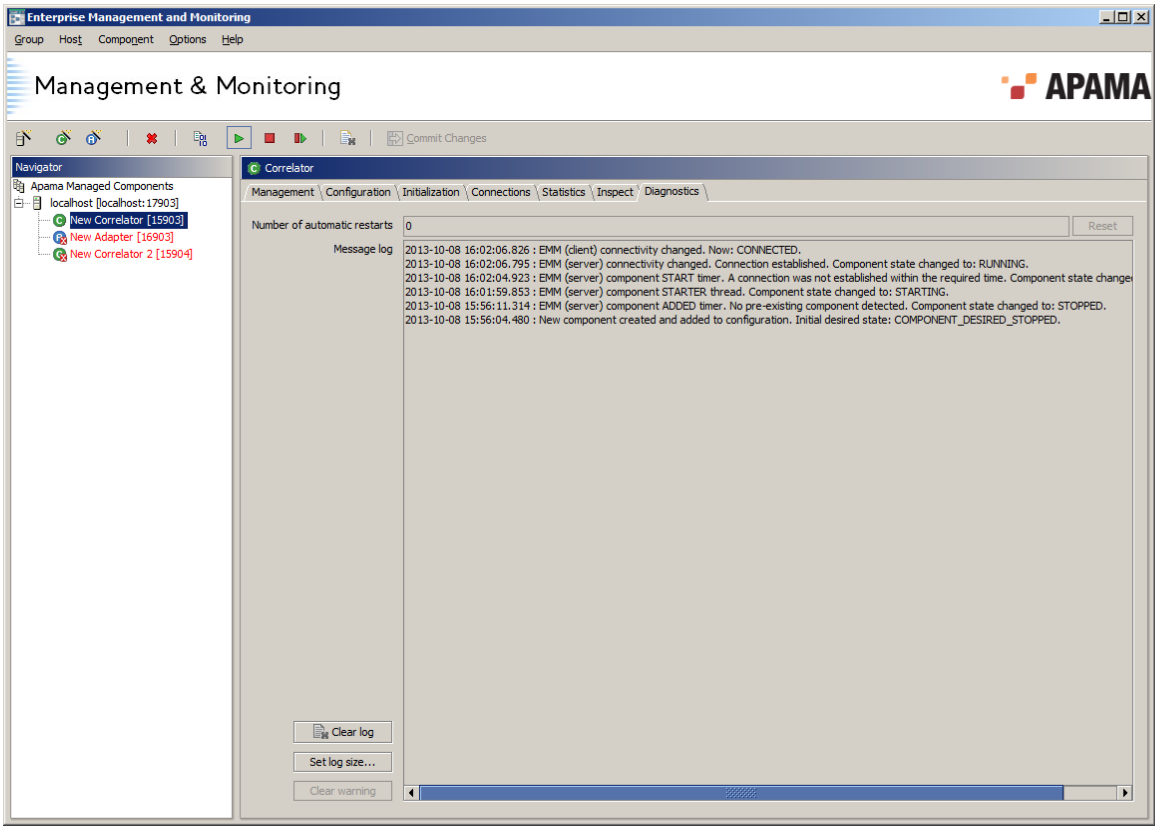
   This displays the **View Event Definitions** window.



```
package com.apama.samples.vwap;

event Match {
                    string stockName;
                    float period;
                    float percentChange;
                    float vwap1;
                    float vwap2;
}
```

Diagnostics tab

The Diagnostics tab displays diagnostics information about the changes in the state of the correlator.



Number of automatic restarts: 0

Message log

```
2013-10-08 16:02:06.826 : EMM (client) connectivity changed. Now: CONNECTED.
2013-10-08 16:02:06.795 : EMM (server) connectivity changed. Connection established. Component state changed to: RUNNING.
2013-10-08 16:02:04.923 : EMM (server) component START timer. A connection was not established within the required time. Component state change
2013-10-08 16:01:59.853 : EMM (server) component STARTER thread. Component state changed to: STARTING.
2013-10-08 15:56:11.314 : EMM (server) component ADDED timer. No pre-existing component detected. Component state changed to: STOPPED.
2013-10-08 15:56:04.480 : New component created and added to configuration. Initial desired state: COMPONENT_DESIRED_STOPPED.
```

It provides two elements of diagnostics information:

- Number of automatic restarts – the total number of automatic restarts since the last time you manually started the component or since you enabled automatic restart. See "Management tab" on page 53 for details of how automatic restart behavior can be configured. The Reset button lets you manually reset the counter and make EMM auto-restart the component again if it is currently in the FAIL state.
- Message log – A log of diagnostic information generated by EMM. Each log entry starts with the date and time of when the event being logged occurred. Note that most entries are tagged as being generated by `EMM (server)` or `EMM (client)`. `EMM (client)` indicates the graphical front-end, whereas `EMM (server)` indicates the backend model.

The Message log has three associated buttons:

- Clear log – Removes all the messages logged for this component.
- Set log size... – Configures the size of the diagnostics log for this component. Note that the default log size assigned to new components can be configured from the Preferences dialog, and is set to `100` entries when Apama is installed. If the size is exceeded the oldest entries are removed to make way for new entries. Minimum size is 1 line.
- Clear warning – Changes the state of the component from `WARN` to `RUNNING`; see "Component status indicators" on page 38 for a full explanation of the meaning of the `WARN` state. This button is only enabled when the component is started and in the `WARN` state.

4 Deploying and Configuring Adapters

Adapters allow Apama to interface with external sources of events like message buses, event feeds, or databases. You use the Integration Adapter Framework (IAF) with any of the standard, pre-packaged Apama adapters installed with the product or with custom adapters that you develop. To deploy an adapter, you specify the appropriate information in the adapter's configuration file and then start the IAF using the configuration file. This section describes how to deploy Apama adapters from the EMM console. You can also use Apama command line utilities to start and manage adapters.

- For more information on using standard Apama adapters, see "Standard Apama Adapters" in *Developing Adapters* (available if you selected Developer during installation).
- For more on developing and using custom adapters, see "The Integration Adapter Framework" in *Developing Adapters* (available if you selected Developer during installation).

This topic describes how to use the EMM menu items to carry out operations on Apama adapters. However, all these operations can also be carried out from the context menus in the Navigation Pane, and the most useful are also available on the toolbar.

Before operations can be carried out on these components, a Sentinel Agent must be installed and running on the target host, and the EMM console must have been configured to manage that host, as detailed in "Managing hosts" on page 28.

Adding adapters

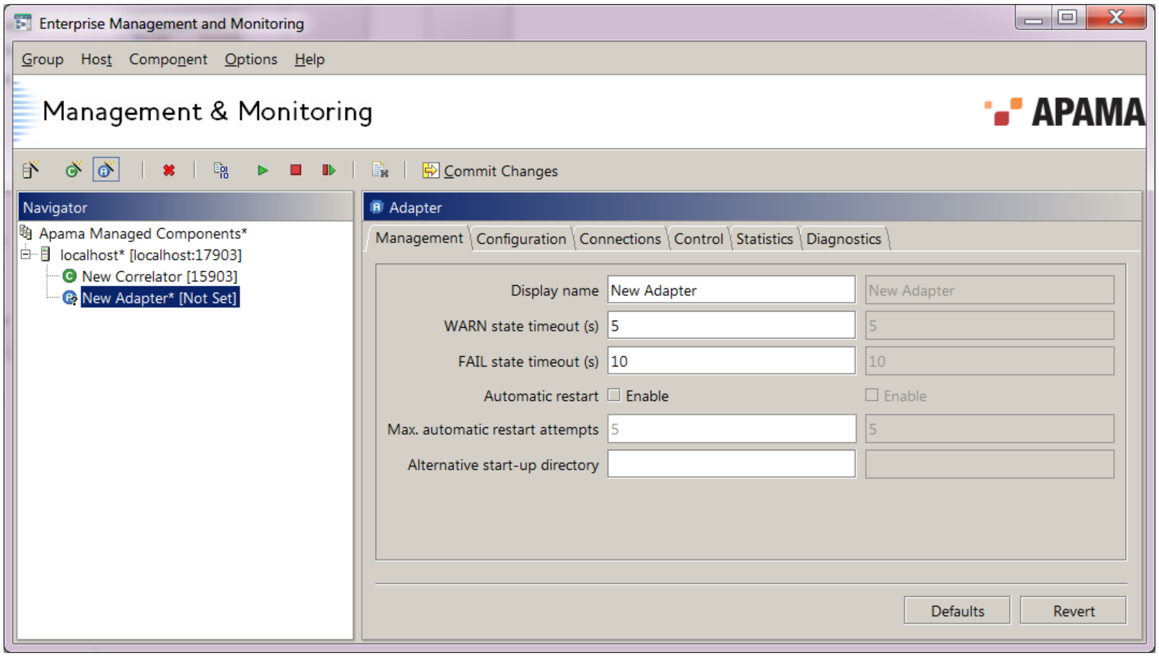
To define and configure a new IAF adapter:

1. Select the host where you want to add the adapter in the EMM Navigation Pane.
2. Select Component > Add adapter from the EMM menu or click the button on the tool bar.

   This adds an adapter to the host in the Navigation Pane, selects it, and displays the Management tab in the Details Pane. The other tabs available on this Details Pane for an adapter are the Configuration tab, the Connections tab, the Statistics tab, the Control tab, and the Diagnostics tab. These are discussed in detail in "The Adapter tabs" on page 74.

EMM initializes the new IAF adapter with a set of default options, which are normally safe. The default value for the listening port (which has to be unique per host) is automatically selected so as not to conflict with any other known components.

Before the new adapter can be started for this first time you must 'commit' its configuration using the Commit Changes ( ) button. Note that most of the adapter's configuration options only apply when the component is started up, so changes committed after the component is already running will usually not take effect until it is restarted.



Display name	New Adapter	New Adapter
WARN state timeout (s)	5	5
FAIL state timeout (s)	10	10
Automatic restart	□ Enable	□ Enable
Max. automatic restart attempts	5	5
Alternative start-up directory		

Deploying and Configuring Adapters

Configuring adapters

The contents of the Details Pane change according to what is currently selected in the Navigation Pane. When a component is added to a host, it is automatically selected in the Navigation Pane.

Note: As already described, you need to press Commit Changes ( ) before any changes you make to any panel on the Details Pane take effect. The values that are currently in effect are shown within the rightmost grayed out labels.

Deploying and Configuring Adapters

Specifying paths and filenames in the Details Pane

On several of the component Details Pane panels you are asked to provide a path or filename (for example, to specify where logging information should be stored, or where an adapter configuration file should be loaded from).

It is recommended that *absolute paths* and filenames be used where possible – an absolute path being one that specifies the whole path, for example:

`C:\Program Files\SoftwareAG\Apama 5.1\Logging\New_Correlator_1.log` (on Windows)

or

`/opt/SoftwareAG/apama_5.1/configuration/New_Correlator_1.log` (on UNIX)

In contrast, *relative paths* are incomplete, or just consist of a filename on its own, such as:

```
New_Correlator_1.log
```

Important notes:

- Unless otherwise stated, all paths are located on the file system of the *remote host*, on which the Sentinel Agent and managed components are running, rather than the local host where EMM is running.
- Filenames and paths should never be enclosed in quotes, even if they contain spaces.
- If a relative path is provided – as is the case by default for component log files – it is assumed to be relative to the component's `Alternative start-up directory` setting if one was configured, or to the *current working directory* of the Sentinel Agent on that host if not. See "Working directory" on page 31 for details of how the Sentinel Agent's working directory is determined.

Configuring adapters

The Adapter tabs

When an IAF adapter is selected in the Navigation Pane, the Details Pane will display the following tabs:

- "Management tab" on page 74
- "Configuration tab" on page 75
- "Connections tab" on page 77
- "Control tab" on page 77
- "Statistics tab" on page 77
- "Diagnostics tab" on page 80

Management tab

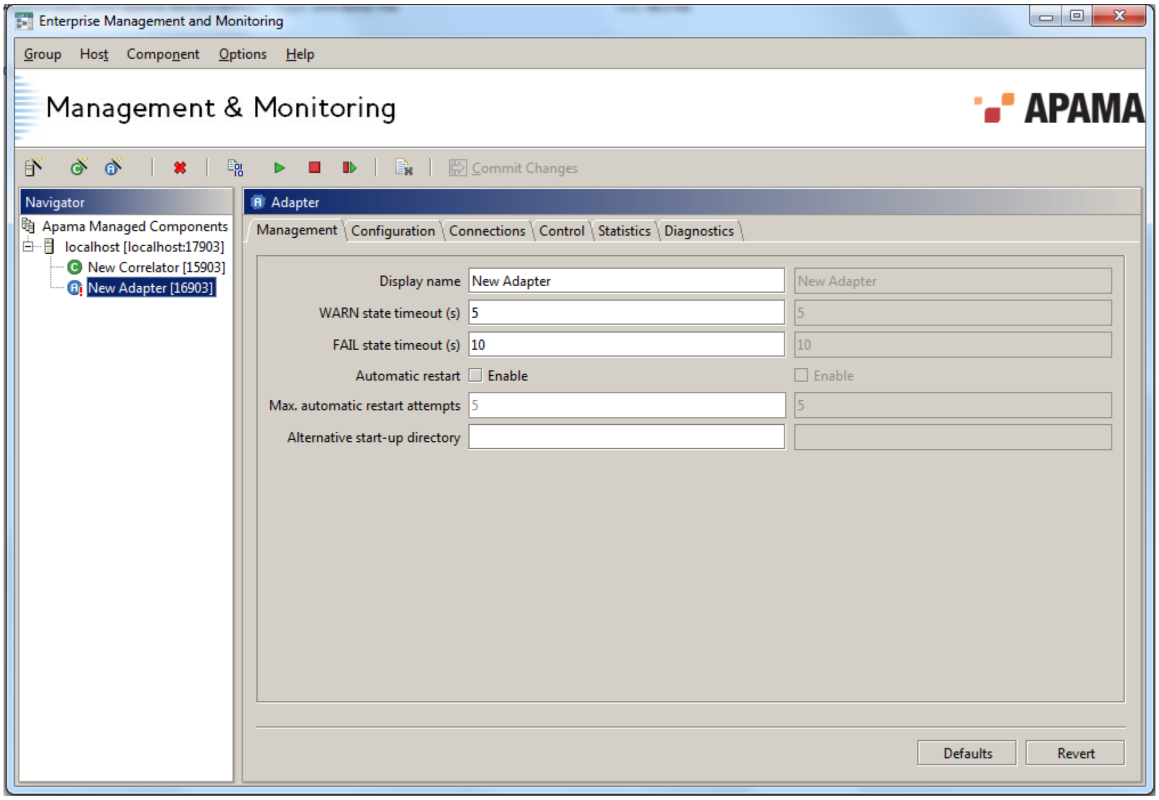
This tab contains a number of parameters that are relevant to managing an IAF adapter, and is identical to that of a correlator.

The available parameters are:

- Display name – This is the name to associate with this adapter in the Navigation Pane. The name can be changed at any time, even if the adapter has already been started.
- WARN state timeout (s) – This is the number of seconds to wait before moving the component into the `WARN` state if it is not changing state (stopping/starting/ restarting) as required. This setting can be changed at any time.
- FAIL state timeout (s) – This is the number of seconds to wait after entering the `WARN` state before moving into the `FAIL` state, if the component has still failed to change state as expected. This setting can be changed at any time.
- Automatic restart enable – Tick to configure EMM to monitor the selected component. If it were to stop unexpectedly, EMM would automatically restart it – after waiting the amount of time required for it to enter the `FAIL` state (and subject to the configured limit on the number of restart attempts described below). This setting can be changed at any time.

Note: Component monitoring and auto-restart is carried out from EMM and requires EMM to be running.

- Max. automatic restart attempts – This option is only available if `Automatic restart` is enabled. It specifies the total number of automatic restarts to perform (or attempt) without user intervention. The count is made from when you last enabled `Automatic restart` and committed, or explicitly stopped/started/ restarted the adapter. This setting can be changed at any time.
- Alternative start-up directory – By default all components are started from the current working directory of the Sentinel Agent running on the host in question (see in "Working directory" on page 31). If you wish, you can provide an alternative startup folder here. Note that this option is only functional if the Sentinel Agent running on the host in question is version 1.1.1 or greater.



Apama Managed Components
localhost [localhost:17903]
New Correlator [15903]
New Adapter [16903]

Display name	New Adapter	New Adapter
WARN state timeout (s)	5	5
FAIL state timeout (s)	10	10
Automatic restart	□ Enable	□ Enable
Max. automatic restart attempts	5	5
Alternative start-up directory		

Click the Commit Changes button when you are finished customizing the new IAF adapter. This applies the outstanding changes.

The Default button resets all outstanding parameter values to their defaults, while the Revert button reverts the outstanding parameter values to their current committed values.

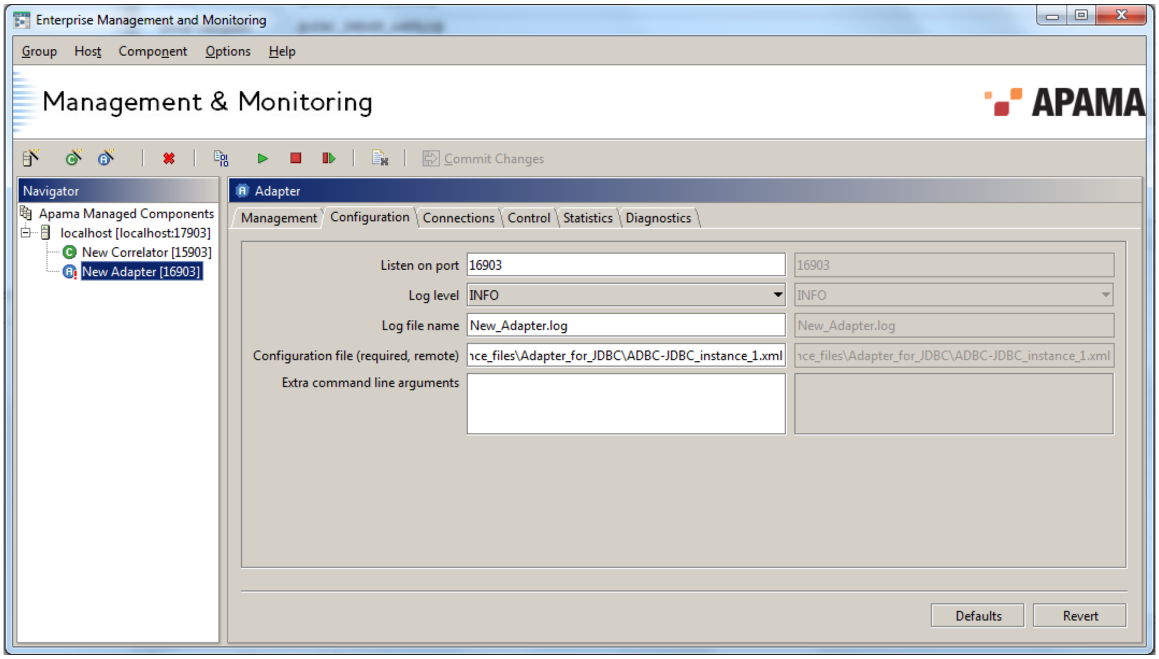
Configuration tab

This tab contains a number of parameters that configure how an IAF adapter operates. As these are all startup parameters, you should adjust them before you start the component. If the component is already running they will only have effect the next time it is started.

The following parameters are available for an IAF adapter:

- Listen on port - Port on which the IAF adapter should listen for monitoring and management requests (default is `16903`). The adapter will fail to start if this port is in use by any other component, or for that matter by any other software that is running on the relevant host.
- Log level - Sets the log level the IAF adapter should log at – must be one of `CRIT`, `ERROR`, `WARN`, `INFO`, `DEBUG` (in increasing order of verbosity). If this value is not set, then the IAF adapter will log at the level specified in the adapter configuration file. If a value is indeed specified, that the EMM value will always override the log level specified in the adapter configuration file.
- Log file name - Sets the filename that the adapter should write log messages to (on the file system of the host the correlator is running on). It is recommended that an absolute path be provided.
- Configuration file (required, remote) – Specifies the path of an IAF adapter configuration file, as described in in *Developing Adapters*. The IAF adapter cannot be committed or started until a configuration file is specified.

  The file's path is looked up on the file system of the host the adapter is running on, and it is recommended that an absolute path be provided. See "Specifying paths and filenames in the Details Pane" on page 73 for more information about setting filename options correctly.
- Extra command line arguments – This option allows additional unspecified command line arguments to be passed to the IAF adapter. For example these might be special settings provided to you by Apama Customer Support to address specific issues.



Listen on port	16903	16903
Log level	INFO	INFO
Log file name	New_Adapter.log	New_Adapter.log
Configuration file (required, remote)	ıce_files\Adapter_for_JDBC\ADBC-JDBC_instance_1.xml	ıce_files\Adapter_for_JDBC\ADBC-JDBC_instance_1.xml
Extra command line arguments		

Click on the Commit Changes button when you are finished customizing the new IAF adapter. This applies these outstanding changes for use when the adapter is actually started.

The Default button resets all outstanding parameter values to their defaults, while the Revert button reverts the outstanding parameter values to their current committed values.

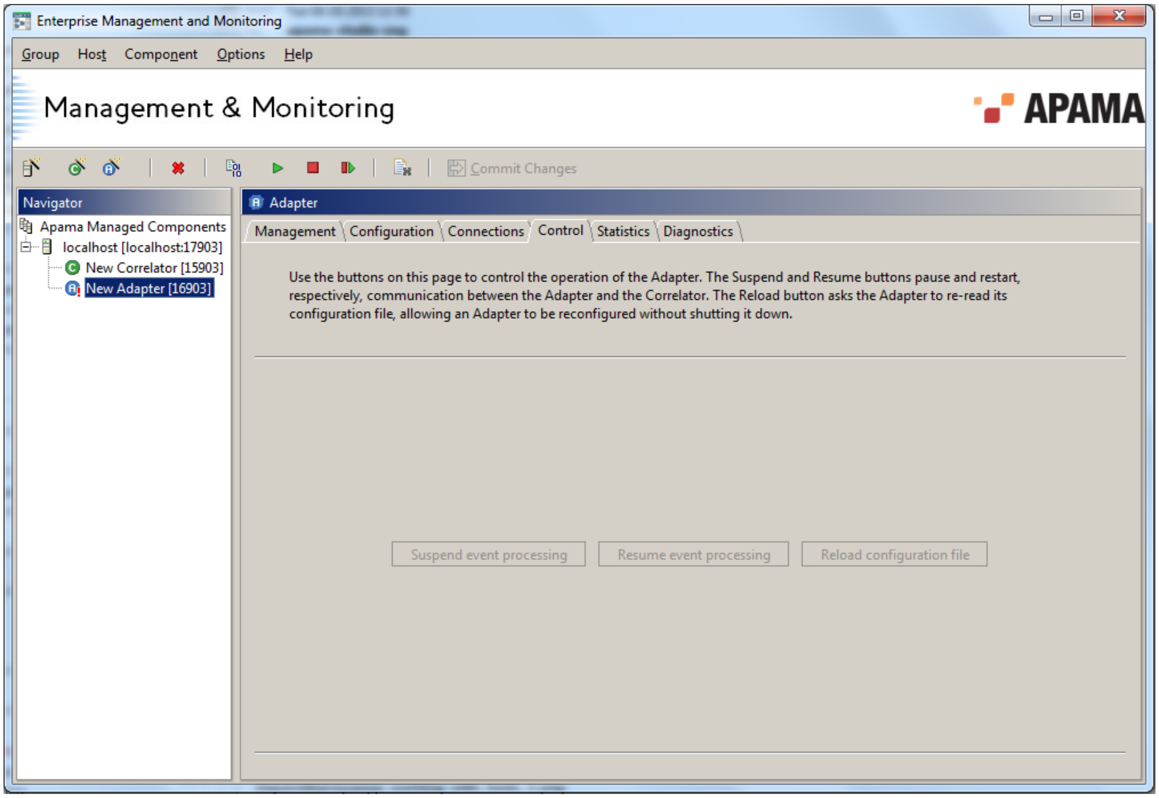
Connections tab

This tab allows you to connect an adapter to upstream components. It also displays downstream components that are connected to the adapter. The tab is similar to the Connections tab for a correlator. For a complete description of the Connections tab, "Connections tab" on page 60.

Control tab

The Control tab provides for control of an IAF adapter. Three operations are supported:

- Suspend event processing – This button will pause a running IAF adapter.
- Resume event processing – This button will resume a running IAF adapter.
- Reload configuration file – This button will ask the IAF adapter to re-read its configuration file, allowing an adapter to be reconfigured without shutting it down.



Use the buttons on this page to control the operation of the Adapter. The Suspend and Resume buttons pause and restart, respectively, communication between the Adapter and the Correlator. The Reload button asks the Adapter to re-read its configuration file, allowing an Adapter to be reconfigured without shutting it down.

Statistics tab

This tab allows monitoring of the status of an operational IAF adapter.

Status information is refreshed regularly, by default, once a second. You can change this rate by changing the Component status display update interval (s) value on the Timing tab within the Preferences dialog. The Preferences dialog is available by selecting Options > Preferences... from the EMM menu.

No status information will be displayed if EMM cannot communicate with the IAF adapter selected. This is usually because the adapter has not yet been started or has been stopped. However, it could also be due to network failure. In this case, statistics will reappear once the connection is restored.

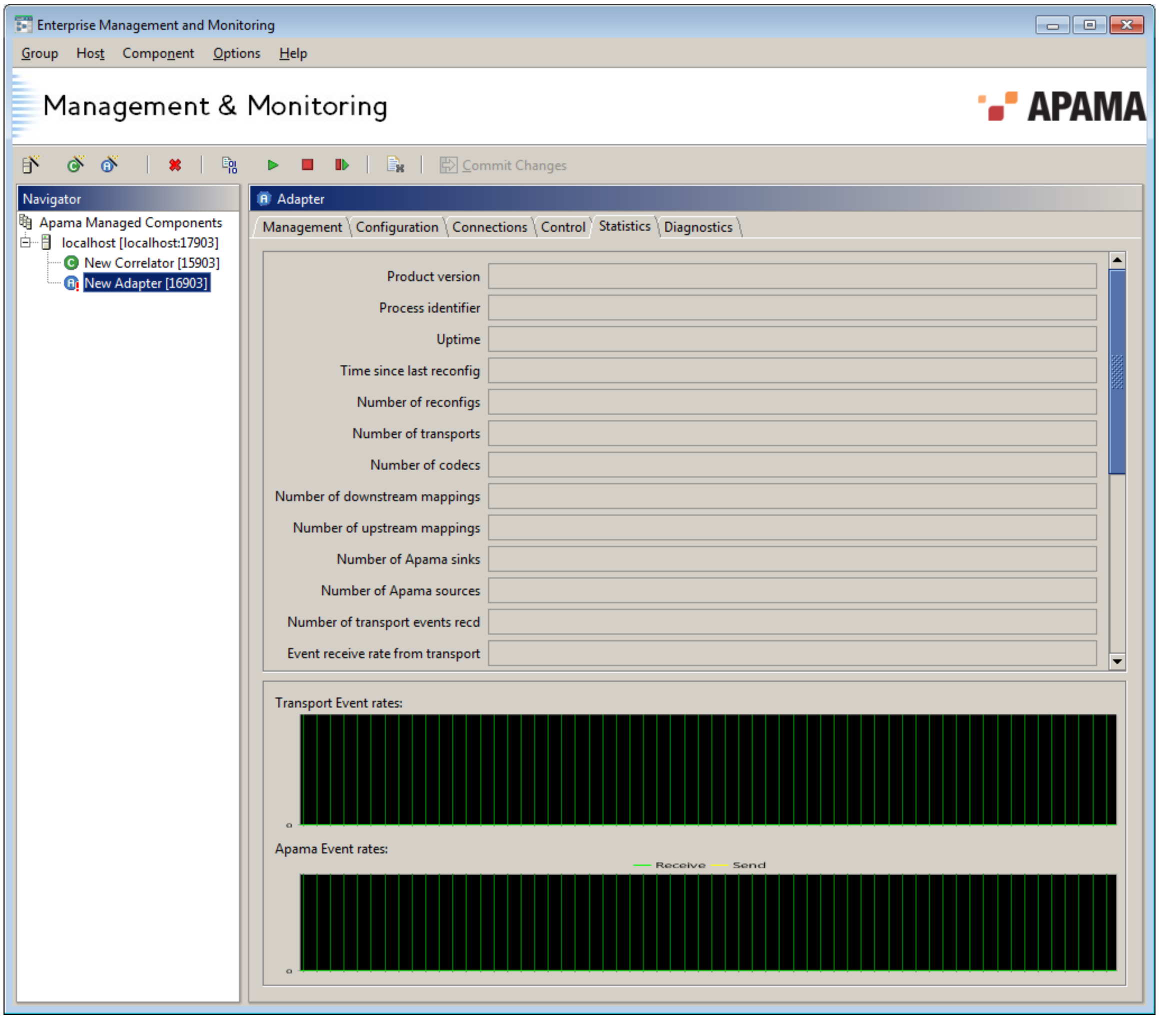
The following statistics are available:

- `Product version` – The Apama release number.
- `Process identifier` – The identifier assigned to this component by the operating system.
- `Uptime` – The time in seconds since this adapter was started. This time is maintained and reported by the component itself, so if the adapter was started independently of EMM and only managed by EMM later, the value would still be accurate.
- `Time since last reconfig` – The time in milliseconds since this adapter was last 'reconfigured', that is, reset with a new configuration file.
- `Number of reconfigs` – The total number of reconfigurations.
- `Number of transports` – The total number of Transport Plugins active in this Adaptor.
- `Number of codecs` – The total number of Codec Plugins active in this Adaptor.
- `Number of downstream mappings` – The total number of mapping rules set up in the adapter's semantic mapper for downstream (from an external message source into Apama events) event mappings.
- `Number of upstream mappings` – The total number of mapping rules set up in the adapter's semantic mapper for upstream (from Apama events into an external message sink) event mappings.
- `Number of Apama sinks` – The number of Apama correlators that the adapter is sending events to.
- `Number of Apama sources` – The number of Apama correlators that the adapter is receiving events from.
- `Number of transport events recd` – The total number of messages received by the adapter's transport Plugins from external message sources (i.e. downstream).
- `Event receive rate from transport` – The number of messages per second currently being received by the adapter's transport plug-ins. This value is computed with every status refresh and is only an approximation.
- `Number of transport events sent` – The total number of messages sent out by the adapter's Transport Plugins to external message sinks (i.e. upstream).
- `Event send rate to transport` – The number of messages per second currently being sent out by the adapter's Transport Plugins. This value is computed with every status refresh and is only an approximation.
- `Number of Apama events recd` – The total number of Apama events received by the adapter from Apama for upstream conversion.
- `Event receive rate from Apama` – The number of Apama events per second currently being received by the adapter from Apama. This value is computed with every status refresh and is only an approximation.
- `Number of Apama events sent` – The total number of Apama events sent to Apama by the adapter because of a downstream conversion.

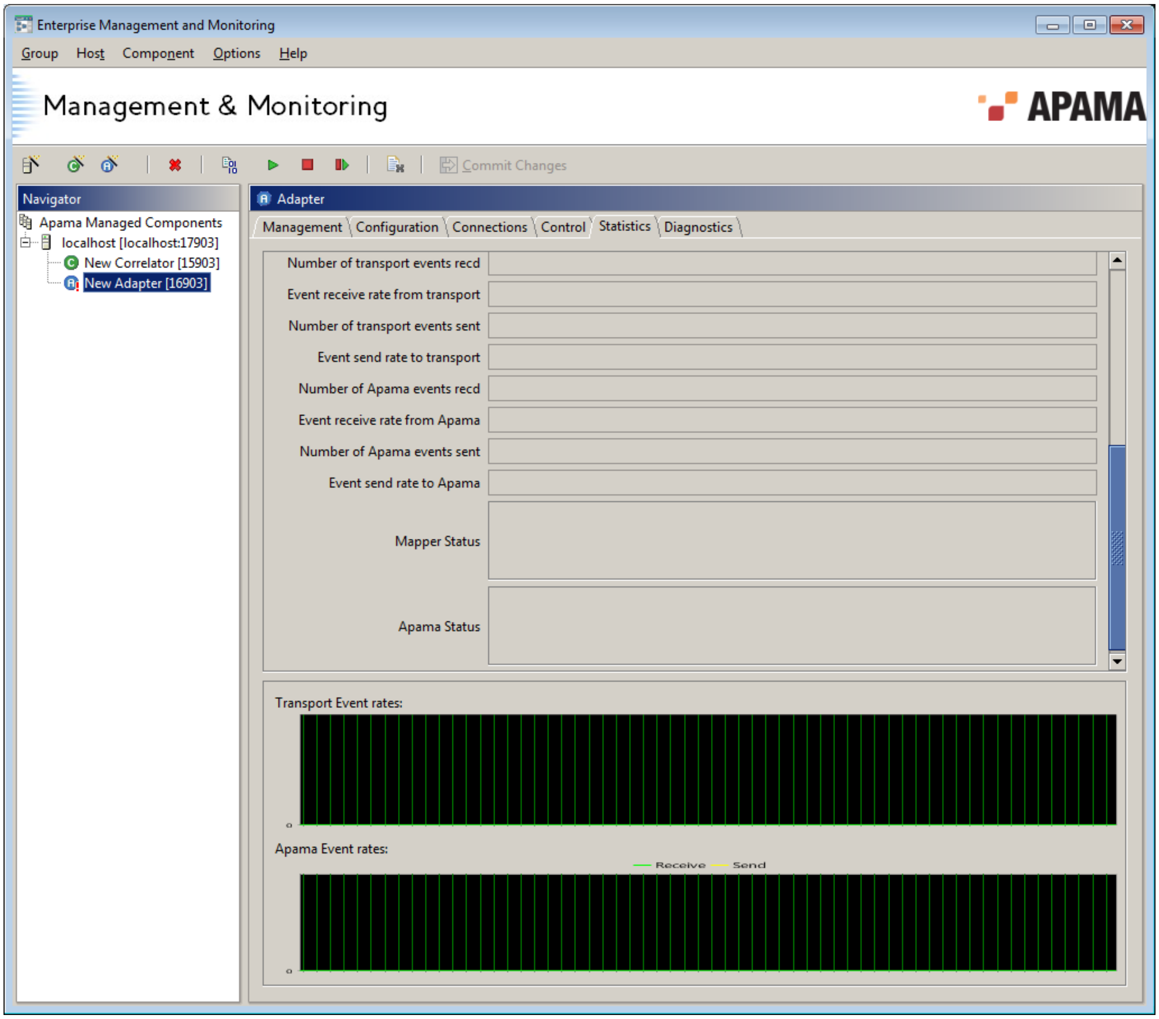
- `Event send rate to Apama` – The number of Apama events per second currently being sent to Apama by the Adaptor. This value is computed with every status refresh and is only an approximation.
- `Mapper Status` – Indicates whether the adapter's semantic mapper has started.
- `Apama Status` – Indicates whether the adapter has contacted the Apama sinks and sources specified in the Configuration File.

The four event rate statistics are also displayed graphically over the last 60 seconds at the bottom of the Statistics tab in the two graphs: `Transport Event rates` and `Apama Event rates`.

The following two illustrations show the top and bottom portions of the Statistics tab for adapters.

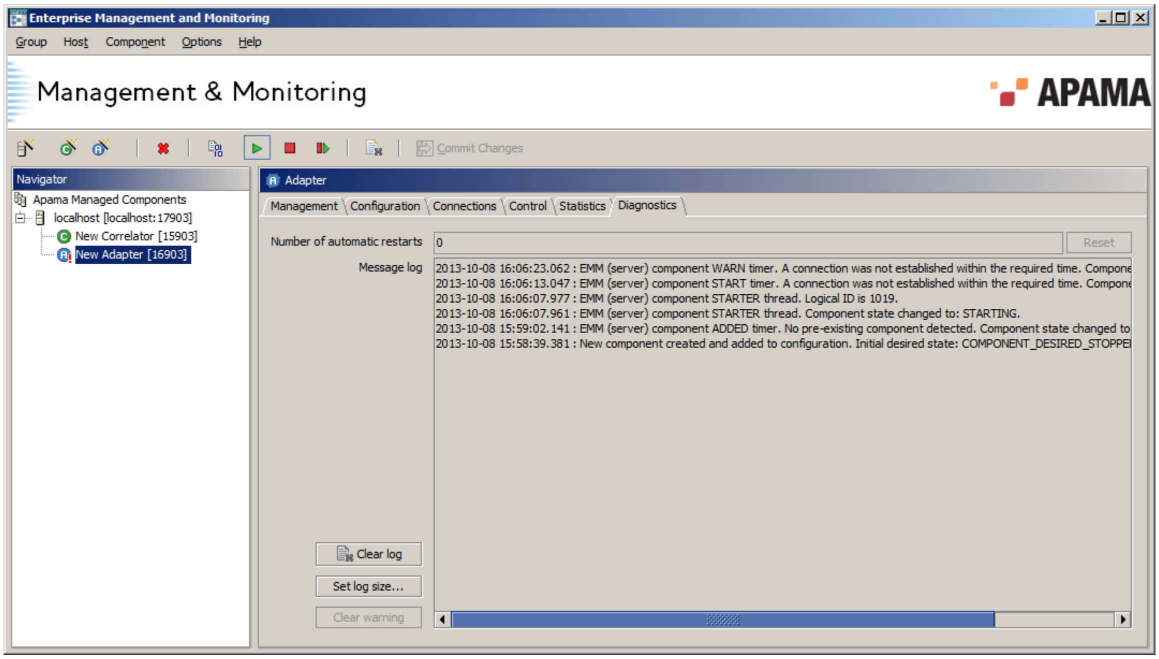


Product version
Process identifier
Uptime
Time since last reconfig
Number of reconfigs
Number of transports
Number of codecs
Number of downstream mappings
Number of upstream mappings
Number of Apama sinks
Number of Apama sources
Number of transport events recd
Event receive rate from transport



Diagnostics tab

The Diagnostics tab for an IAF adapter is identical to that of a correlator and displays diagnostics information about changes in the state of the adapter.



Number of automatic restarts 0

Message log

```
2013-10-08 16:06:23.062 : EMM (server) component WARN timer. A connection was not established within the required time. Compone
2013-10-08 16:06:13.047 : EMM (server) component START timer. A connection was not established within the required time. Compone
2013-10-08 16:06:07.977 : EMM (server) component STARTER thread. Logical ID is 1019.
2013-10-08 16:06:07.961 : EMM (server) component STARTER thread. Component state changed to: STARTING.
2013-10-08 15:59:02.141 : EMM (server) component ADDED timer. No pre-existing component detected. Component state changed to
2013-10-08 15:58:39.381 : New component created and added to configuration. Initial desired state: COMPONENT_DESIRED_STOPPE
```

For complete information on this tab, see "Diagnostics tab" on page 68.

5 Dashboard Deployment Concepts

This section discusses fundamental dashboard deployment and administration concepts.

Deployment options

There are two types of dashboard deployment:

- Web-based: as a simple, thin-client Web page, as an applet, or as a Java Web Start application
- Local: as a locally-installed desktop application (the Dashboard Viewer) together with dashboard-specific files that the application can open

The following sections compare Web-based deployments with local deployments with regard to these factors:

- "Application installation" on page 84
- "Authentication" on page 84
- "Authorization" on page 85
- "Data protection" on page 85
- "Scalability" on page 85

The section "Dashboard support for Apple iPad" on page 85 discusses iPad deployments.

The section on the "Data server and display server" on page 86 discusses some considerations that are relevant to choosing among Web-based deployment options.

Dashboard Deployment Concepts

Application installation

Local deployments require the use of the Dashboard Viewer desktop application (available on Windows platforms only). End users open locally-deployed dashboards in the Dashboard Viewer, which must be pre-installed locally or on a shared file system. See the Apama *Dashboard Viewer* guide for information about using the Dashboard Viewer.

With Web-based deployment, the Dashboard Viewer does not need to be installed locally. Dashboards are invoked through a Web browser, and are installed on demand, as Web pages, applets, or Web Start applications, so they can easily be deployed across a wide area network, including the Internet.

For applet and WebStart deployments, the local Web browser must have the Java plug-in. For simple Web page deployments, no Java plug-in is required. See "Data server and display server" on page 86.

Deployment options

Authentication

Web-based deployments provide Web-based login functionality and use the authentication mechanism provided by your application server. They support authentication customization by allowing you to, for example, configure your application server to use the security realm and authentication service of your choice.

Local deployments include Data Server login functionality, and support authentication by allowing you to supply any JAAS-compliant authentication module as a plugin to the Data Server and to the Dashboard Viewer.

Deployment options

Authorization

Web-based deployments support role-based dashboard access control, which allows you to associate a role with a deployed dashboard, and to authorize use of the dashboard only for application-server users with the dashboard's associated role.

Local deployments support dashboard access control by allowing you to use the system security mechanisms in order to restrict access to the deployed dashboard files.

Both types of deployment support Scenario and DataView access control, which allows you to control who can have which type of access to which Scenarios and DataViews.

Deployment options

Data protection

With Web-based deployments you can secure inter-process communication by enabling HTTPS in the application server. With local deployments you can secure inter-process communication by enabling secure sockets (SSL) in the Data Server or Display Server.

With both types of deployment you can secure inter-process communication through the use of secure channels (SSH) and virtual private networks (VPN).

Deployment options

Refresh latency

The Display Server's minimum refresh latency (5 seconds) is greater than that of the Data Server. Use the Data Server for applications that require high-frequency screen updates.

Deployment options

Scalability

Both types of deployment are highly scalable, since both use the Data Sever or Display Server to mediate access to Correlators.

Deployment options

Dashboard support for Apple iPad

Apama dashboards are now supported on Mobile Safari for iOS 5.0 on the Apple iPad. Any dashboard built with this Apama release is fully functional when viewed on the iPad, provided that the dashboard

- Is a Display Server deployment

- Defines no System Command to Run DOS Command or UNIX Shell

When you develop dashboards targeted for the iPad, use Layout mode for best results.

End users should be aware of the following characteristics associated with using a dashboard on the iPad:

- If mouseover text and drilldown are both enabled on a visualization object, two touches are required for drilldown. The first touch on an element (for example, a bar of a bargraph) displays the mouseover text for that element, and the second touch on the same element performs the drilldown.
- Drilldown results in a new browser tab, rather than a new window.
- You can scroll pages and listboxes by using two-finger scrolling.
- If you minimize Safari by clicking on the iPad's home button, the display will not update until you wake up the iPad or reopen Safari. In some cases it may be necessary for you to refresh the page from Safari's navigation bar.

Deployment options

Data server and display server

The Dashboard Data Server and Display Server can each serve as the gateway through which dashboards can access your applications running on the correlator. Use of these Servers provides scalability by obviating client management on the part of the correlator, and provides security by not exposing the correlator directly to clients.

The Data Server mediates correlator access for applet, WebStart, and local deployments; the Display Server mediates correlator access for simple thin-client, Web-page deployments.

The Data Server delivers raw data from which deployed dashboards construct the visualization objects that they display. The Display Server, in contrast, delivers already-constructed visualization objects in the form of image files and image maps, and therefore no Java plugin is required on clients of the Display Server.

The Display Server's minimum refresh latency (5 seconds) is greater than that of the Data Server. Use the Data Server for applications that require high-frequency screen updates.

Managing the Data Server and Display Server is covered in "Managing the Dashboard Data Server and Display Server" on page 327.

Dashboard Deployment Concepts

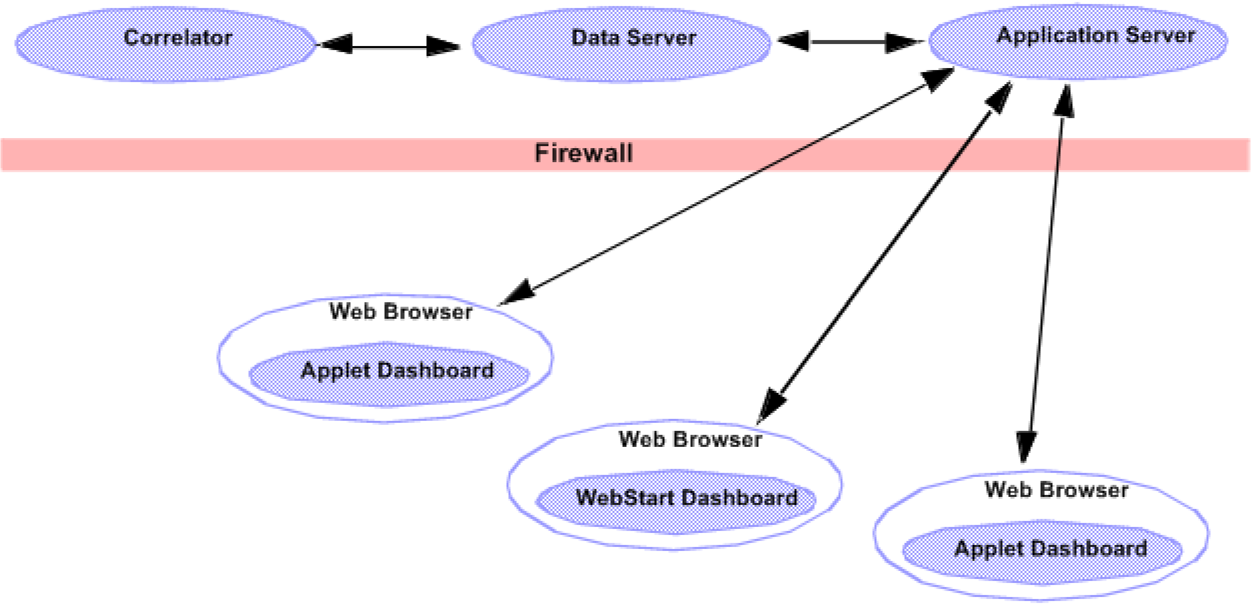
Process architecture

Deployed dashboards connect to one or more Correlators via a Dashboard Data Server. As the Scenarios in a Correlator run and their variables change, update events are sent to all connected dashboards. When a dashboard receives an update event, it updates its display in real time to show the behavior of the Scenarios. User interactions with the dashboard, such as creating an instance of a scenario, result in control events being sent via the Data Server to the Correlator.

Applet and Web Start dashboards communicate with the Data Server via servlets running on an application server.

As illustrated in "Figure 1" on page 87, with applet and WebStart deployments, dashboards communicate with your application server, which communicates with the Dashboard Data Server. The Data Server mediates access to the Correlator.

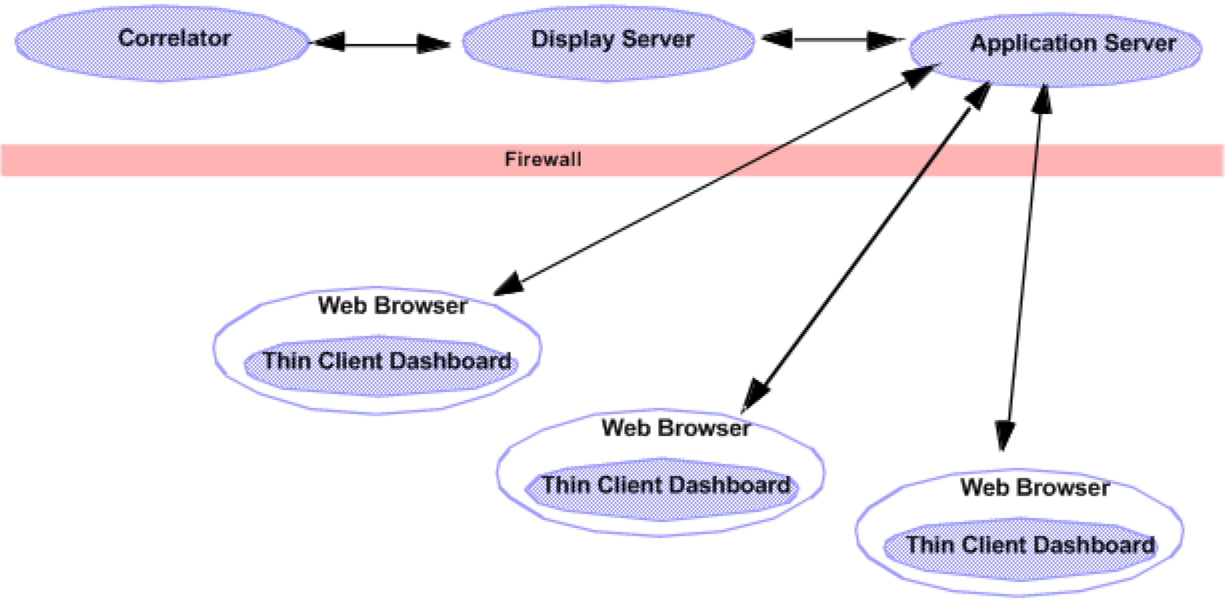
Figure 1. Process architecture: Applet and WebStart deployments



Simple, thin-client, Web-page dashboards communicate with the Display Server via servlets running on your application server bundled with your Apama installation.

As illustrated in "Figure 2" on page 88, with thin-client, Web-page deployments, dashboards communicate with your application server, which communicates with the Dashboard Display Server. The Display Server mediates access to the Correlator.

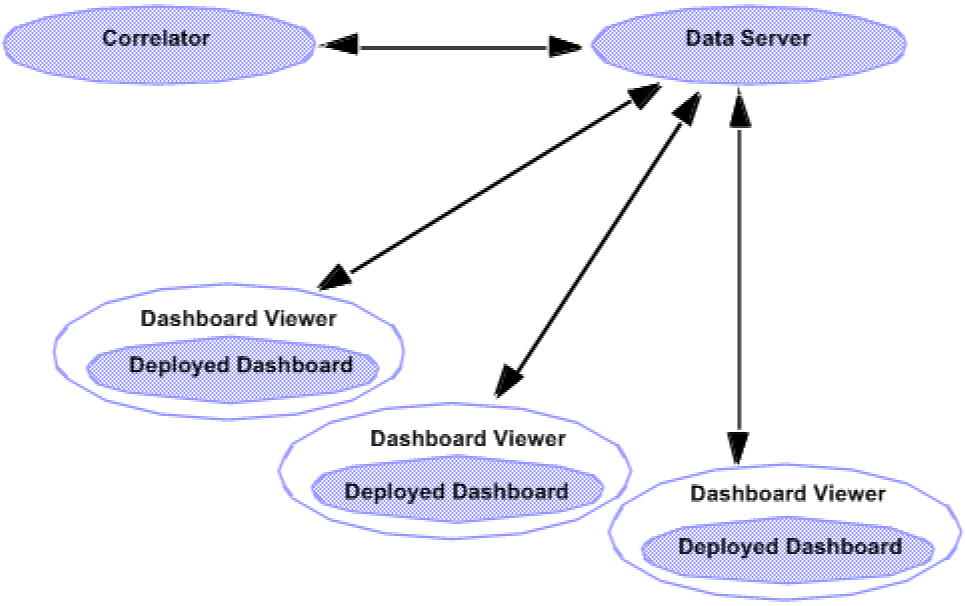
Figure 2. Process architecture: Thin-client, Web-page deployments



Locally-deployed dashboards communicate directly with the Data Server.

As illustrated in "Figure 3" on page 88, with local deployments, dashboards communicate with the Dashboard Data Server, which communicates.

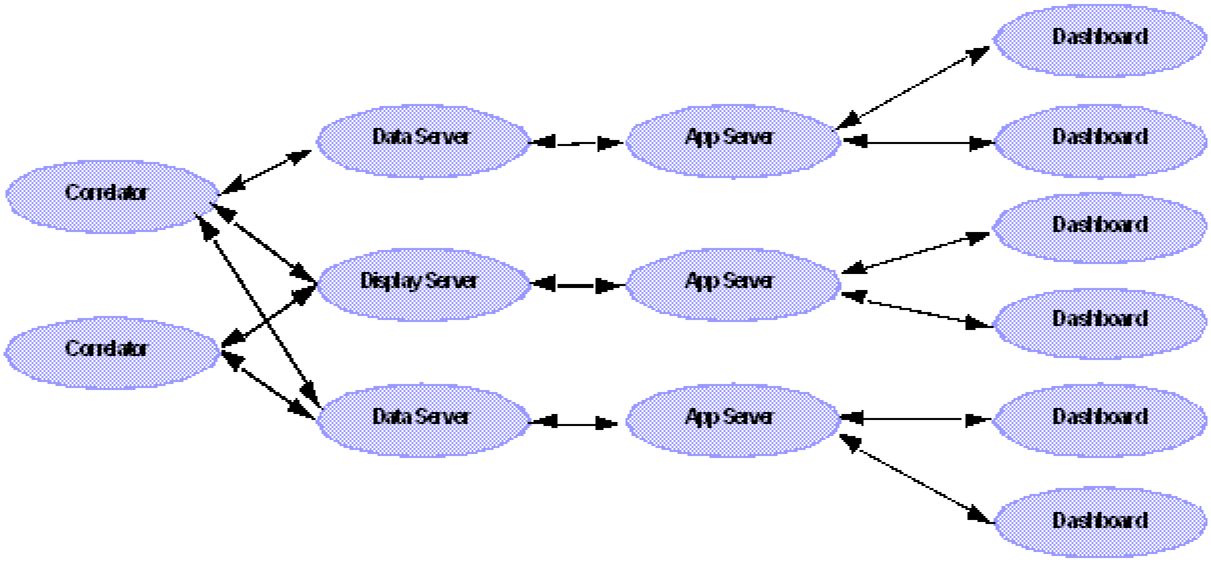
Figure 3. Process architecture: Locally-deployed dashboards



You can scale your application by adding Data Servers to your configuration. Each Correlator can communicate with multiple Data Servers, and each Data Server can communicate with multiple Correlators.

As illustrated in "Figure 4" on page 89, each correlator can communicate with multiple Data Servers and Display Servers. Each Data Server and Display Server can communicate with multiple Correlators.

Figure 4. Process architecture: Adding Data Servers to your configuration



Deployed dashboards have a unique associated default Data Server or Display Server, but advanced users can associate non-default Data Servers with specific attachments and commands. This provides additional scalability by allowing loads to be distributed among multiple servers. This is particularly useful for Display Server deployments. By deploying one or more Data Servers behind a Display Server, the labor of display building can be separated from the labor of data handling. The Display Server can be dedicated to building displays, while the overhead of data handling is offloaded to Data Servers. See "Working with multiple Data Servers" on page 338 for more information.

Dashboard Deployment Concepts

Builders and administrators

There are two types of activity involved in making dashboards available to end users:

- Dashboard development, which requires the use of the Apama Dashboard Builder, as well as the use of the Dashboard Deployment Configuration Editor to generate a deployment package. See "Preparing dashboards for deployment" in *Using Apama Studio*, available if you selected Developer during installation.
- Dashboard deployment, which requires installing the deployment package, as well as administering the Data Server or Display Server and managing dashboard security.

Sometimes these activities are performed by different individuals. In such a case, the dashboard developer must be sure to communicate the following information to the dashboard administrator regarding the dashboards to be deployed:

- The location and file name of the `.war` file or `.zip` file that was generated by the Deployment Configuration Editor when the developer prepared the dashboard for deployment
- For Display Server deployments, the location of the dashboard project directory (the directory that contains the project’s `.rtv` files)
- For Web-based deployments, the Data Sever or Display Server host, port, and update rate that the builder supplied to the Configuration Editor

- The logical name for each correlator as well as the host name and port for each *deployment* correlator (if any) that was specified by the dashboard developer in the Apama tab of the Tools > Options... dialog prior to the generation of the deployment package. See Changing correlator definitions for deployment in *Using Apama Studio*.
- The trend-data caching requirements for the deployed dashboards. See "Configuring Trend-Data Caching" on page 334.

Dashboard Deployment Concepts

6 Deploying Dashboards

Apama dashboards can be deployed to supported application server environments. Use the Dashboard Deployment wizard to generate a `.war` file that you then deploy manually using the deployment tools of your application server.

There are additional considerations that are covered in the following topics in this section.

If you are a dashboard developer, see also Preparing Dashboards for Deployment in *Using Apama Studio*.

For a current list of Apama-supported application servers, see Software AG's Knowledge Center in Empower at https://empower.softwareag.com/KnowledgeCenter/default.asp.

Generating the dashboard .war file

Before deploying a dashboard, you need to first generate the `.war` file for that dashboard. The `.war` file is the deployment package for a dashboard. It contains the webapp, servlets, and supporting resources for deploying a dashboard.

You can generate the dashboard `.war` file from Apama Studio using the Dashboard Deployment wizard. You can also generate the `.war` file using a command prompt or script with the `dashboard_management` utility.

For more information on generating dashboard `.war` files, see "Preparing Dashboards for Deployment" in *Using Apama Studio*, available if you selected Developer during installation.

Deploying Dashboards

Installing a dashboard .war file

When deploying to an application server you need to manually install the dashboard `.war` file to that application server using the tools provided by the server. The details of this vary by application server.

1. Ensure that you are installing to a supported application server. For an up-to-date listing of supported application servers, see Software AG's Knowledge Center in Empower: (http://empower.softwareag.com).
2. Copy the `.war` file generated by the Dashboard Deployment wizard to the appropriate location for your application server. For example it may have a `webapps` folder.
3. Configure your application server as desired to support this and to secure access to the dashboard as required. The generated `.war` file will have form authentication enabled.
4. Use the deployment tools of your application server to install the dashboard `.war` file.
5. Test that you can access your dashboard and that access is secured as intended.

Deploying Dashboards

Inside a dashboard .war file

A dashboard `.war` file contains the webapp, servlets, and supporting resources necessary to deploy your dashboard and for it to connect to your Apama dashboard data or display server. For the most part you do not need to be aware of the contents of the .war file. However, there are several points to consider if you encounter problems.

The generated .war file will have form authentication enabled. You must supply the login page for this and configure your application server accordingly. A servlet in the dashboard .war file needs the ability to determine the identity of the user displaying a dashboard. This is to enable user based filtering. For this the servlet calls `request.getRemoteUser()`. Any problems calling this will prevent access to the dashboard.

Deploying Dashboards

Additional steps for display server deployments

For thin client deployments you need to provide the Apama dashboard display server with access to the resources used by the dashboard. When the display server builds the displays for users, it requires the .rtv files, images, and other files used by the dashboards.

For Display Server deployments, copy your project's dashboard directory (the directory that contains the project's .rtv files) to the system on which you want to deploy. The Display Server looks for the dashboard directory in its current directory, so when you start the Display Server, start it so that its current directory is the directory on the deployment system that contains the dashboard project directory.

You need to copy the dashboard project directory, as well as its contents. So, for example, if your dashboard files on the development system are `apama-work/dashboards/MyProject/*.rtv`, they might be located here on the deployment machine:

```
deploy-dir/MyProject/*.rtv
```

In this case, run the Display Server from *deploy-dir*, the parent of the dashboard project directory. Do not run the Display Server from the directory `MyProject`.

Note that the Dashboard Deployment Configuration Editor automatically copies a project's dashboard files to a directory named after the project under the *APAMA_WORK*\dashboards folder. By default, the `display_server` process will be running in the *APAMA_WORK*\dashboards directory so the project files will be picked up automatically.

Deploying Dashboards

Applet and WebStart Deployment

WebStart dashboard deployments require that Java WebStart be installed on the end user’s local machine, and applet dashboard deployments require that the Java plug-in be installed on the end user’s local machine.

Deploying Dashboards

7 Managing and Monitoring over REST

Apama provides a REpresentational State Transfer (REST) HTTP API with which you can monitor Apama components. The monitoring capabilities are available to third-party managing and monitoring tools or to any application that supports sending and receiving XML documents over the HTTP protocol.

Apama components expose several URIs which can be used to either monitor or manage different parts of the system. Some are exposed by most Apama components. These are the generic management URIs. Some are exposed only by specific types of components. For example a correlator running on the default port of 15903 will expose a URI at `http://localhost:15903/correlator/status`. If an HTTP `GET` is issued against the URI, the correlator will return an XML document with the current status of the correlator.

Most URIs are purely for informational purposes and will only respond to HTTP `GET` requests and interacting with them will not change the state of the component. However, some URIs allow the state of the correlator to be modified. They will respond to one or more of the other HTTP methods. For example the `/logLevel` URI will accept an HTTP `PUT` request containing an XML document describing what the log level of the component should be set to. Note that in this case it will also accept a `GET` request which will report the current log level of the component, in keeping with REST principles.

All requests and responses over these interfaces have the same, simple elements. Those elements are: `prop`, `map` and `list`. All elements have a `name` attribute. The `prop` element simply represents a name-value pair with the name contained in the name attribute and the value being the content of the element. The `map` element is an unordered list of named elements which might be any of the three sets of elements though it is quite typically simply a `map` of `prop` elements. See the `/info` URI as an example. The list is very similar to the `map` element except that here the order is typically regarded as significant. All responses from these URIs have a top-level element with the name “`apama-response`” and similar all requests which are sent to these URIs should have a top-level element with the name “`apama-request`”. If there is an error then a response called “`apama-exception`” will be returned.

The `/connections` URL in the Generic Management section is a good example of all these elements being used together. The top level element is a `map` which has two children, both lists, called `senders` and `receivers`. The lists contain a `map` for each sender and receiver. Each sender or receiver has a set of `prop` elements.

Method	URI	Description
GET	`/info`	Summary information about the component including its name, version, etc.
GET	`/ping`	Check if the component is running.
GET	`/deepPing`	Check if the component is running.
GET	`/logLevel`	Display the current log level of the component.
PUT	`/logLevel`	Issues a request to change the log level of the component
GET	`/connections`	Display the connections to the component.

Method	URI	Description
GET	`/info/argv`	Display the arguments that were specified when the component was started.
GET	`/info/envp`	Display the names and values of the environment variables in use.
GET	`/info/categories`	Display the categories available; currently `argv` and `envp`.
GET	`/correlator/status`	Display the runtime operational status of a running correlator.
GET	`/correlator/info`	Display information about the state of a running event correlator.
GET	`/iaf/status`	Summary information about the IAF component.

Generic Management

The Apama REST API `GET` methods return information about correlators, sentinel_agents and IAFs. The `PUT /logLevel` method changes the specified log level.

GET /info

This method returns summary information about the component including its name, version, etc. This is analogous to the data that can be retrieved with the Apama `engine_management` utility, for example, the `hostname` field is exactly what `engine_management --gethostname` gives you.

Response

The XML response to this method looks like this:

```
<?xml version="1.0"?>
<map name="apama-response">
  <prop name="name">Apama Studio Correlator for Demo - Iceberg
             (Demo - Iceberg:Default Correlator)</prop>
  <prop name="type">correlator</prop>
  <prop name="componentVersion">5.1</prop>
  <prop name="productVersion">5.1</prop>
  <prop name="buildNumber">rel/5.1.x@202939</prop>
  <prop name="buildPlatform">ia32-win32-msvc9</prop>
  <prop name="hostname">NBBEDUSERNAME.location.myco.com</prop>
  <prop name="username">username</prop>
  <prop name="pid">4684</prop>
  <prop name="port">15903</prop>
  <prop name="physicalId">5881164212676086470</prop>
  <prop name="logicalId">5881164212676086470</prop>
  <prop name="currentDirectory">C:\Users\username\SoftwareAG\ApamaWork_5.1_32bit\
             studio_workspace\Demo - Iceberg</prop>
  <prop name="uptime">3970854</prop>
</map>
```

GET /ping and GET /deepPing

Checks if the component is still running. If the component is running the client receives an empty response. A failure is a timeout waiting for the response.

Response

The empty response for the `Ping` and `deepPing` looks like this:

```
<?xml version="1.0"?>
<map name="apama-response"> </map>
```

GET /logLevel

This method displays the log level of a component.

Response

```
<?xml version="1.0"?>
<map name="apama-response">
  <prop name="logLevel">INFO</prop>
</map>
```

PUT /logLevel

This methods sets the log level of a component.

Request

```
<?xml version="1.0"?>
<map name="apama-request">
  <prop name="logLevel">INFO</prop>
</map>
```

GET /connections

Gets the incoming and outgoing messaging connections to the given component, along with the channels subscribed to and whether or not the receivers are slow.

Response

The response for this method looks like the following

```
<?xml version="1.0"?>
<map name="apama-response">
  <list name="senders">
    <map name="correlator (on port 57042)">
      <prop name="remoteHost">127.0.0.1</prop>
      <prop name="remotePort">57042</prop>
      <prop name="physicalId">432487434834</prop>
      <prop name="logicalId">432487434834</prop>
    </map>
    <map name="engine_send">
      <prop name="remoteHost">127.0.0.1</prop>
      <prop name="remotePort">57077</prop>
      <prop name="physicalId">643690913</prop>
      <prop name="logicalId">643690913</prop>
    </map>
  </list>
  <list name="receivers">
    <map name="engine_receive">
      <prop name="remoteHost">127.0.0.1</prop>
      <prop name="remotePort">57053</prop>
      <prop name="physicalId">69086982542</prop>
      <prop name="logicalId">69086982542</prop>
      <prop name="slow">false</prop>
      <prop name="disconnectIfSlow">false</prop>
      <list name="channels">
        <prop name="">chan1"</prop>
        <prop name="">chan2"</prop>
      </list>
    </map>
    <map name="engine_receive">
      <prop name="remoteHost">127.0.0.1</prop>
      <prop name="remotePort">58032</prop>
      <prop name="physicalId">4325769021054</prop>
      <prop name="logicalId">4325769021054</prop>
      <prop name="slow">false</prop>
      <prop name="disconnectIfSlow">true</prop>
      <list name="channels">
        <prop name="">*</prop>
      </list>
    </map>
    <map name="correlator (on port 57042)">
      <prop name="remoteHost">127.0.0.1</prop>
      <prop name="remotePort">57071</prop>
      <prop name="physicalId">432487434834</prop>
      <prop name="logicalId">432487434834</prop>
      <prop name="slow">true</prop>
      <prop name="disconnectIfSlow">false</prop>
      <list name="channels">
        <prop name="">chan1"</prop>
      </list>
    </map>
  </list>
</map>
```

GET /info/argv

This method returns the arguments used when starting the component as a list of name/value pairs.

Response

```
<?xml version="1.0"?>
<list name="apama-response">
  <prop name="">C:\Program Files (x86)\Software AG\Apama 5.1\
             bin\correlator.exe</prop>
  <prop name="">--logQueueSizePeriod</prop>
  <prop name="">0</prop>
  <prop name="">-l</prop>
  <prop name="">C:\Users\username\SoftwareAG\ApamaWork_5.1_32bit\
             license\license.txt</prop>
  <prop name="">--port</prop>
  <prop name="">15903</prop>
  <prop name="">--loglevel</prop>
  <prop name="">INFO</prop>
  <prop name="">--name</prop>
  <prop name="">Apama Studio Correlator for Demo - Iceberg(Demo -
             Iceberg:Default Correlator)</prop>
  <prop name="">-j</prop>
  <prop name="">--replayLog</prop>
  <prop name="">Default_Correlator_${START_TIME}_${ID}.replay.log</prop>
</list>
```

GET /info/envp

This method returns a list the environment variables in use as a list of name/value pairs.

Response

```
<?xml version="1.0"?>
<list name="apama-response">
  <prop name="ALLUSERSPROFILE">C:\ProgramData</prop>
  <prop name="APAMA_HOME">C:\Program Files (x86)\Software AG\Apama 5.1</prop>
  <prop name="APPDATA">C:\Users\username\AppData\Roaming</prop>
  <prop name="COMMONPROGRAMFILES">C:\Program Files (x86)\Common Files</prop>
  <prop name="COMMONPROGRAMFILES(X86)">C:\Program Files (x86)\
             Common Files</prop>
  <prop name="COMMONPROGRAMW6432">C:\Program Files\Common Files</prop>
  <prop name="COMPUTERNAME">NBBEDUSERNAME</prop>
  <prop name="COMSPEC">C:\Windows\system32\cmd.exe</prop>
  <prop name="DRVDIR">C:\DRV</prop>
  <prop name="FP_NO_HOST_CHECK">NO</prop>
  <prop name="HOMEDRIVE">C:</prop>
  <prop name="HOMEPATH">\Users\username</prop>
…
  <prop name="TEMP">C:\Users\username\AppData\Local\Temp</prop>
  <prop name="TMP">C:\Users\username\AppData\Local\Temp</prop>
  <prop name="TYPE">Notebook</prop>
  <prop name="USERDNSDOMAIN">LOCATION.MYCO.COM</prop>
  <prop name="USERDOMAIN">MYCO</prop>
  <prop name="USERNAME">username</prop>
  <prop name="USERPROFILE">C:\Users\username</prop>
  <prop name="WINDIR">C:\Windows</prop>
  <prop name="WINDOWS_TRACING_FLAGS">3</prop>
  <prop name="WINDOWS_TRACING_LOGFILE">C:\BVTBin\Tests\installpackage\
             csilogfile.log</prop>
</list>
```

GET /info/category

This method returns the names categories for which you can get general information.

Response

```
<?xml version="1.0"?>
<list name="apama-response">
  <prop name="categories">ha</prop>
  <prop name="categories">categories</prop>
  <prop name="categories">argv</prop>
  <prop name="categories">envp</prop>
</list>
```

Managing and Monitoring over REST

Correlator Management

The Apama REST API provides URIs to use with the `GET` method in order to return information about running Apama correlators. This information includes data such as the number of listeners, monitors, and contexts in the correlator, the number and types of events, and the number of Java applications. For details about the type of information returned by these methods, see Watching correlator runtime status and Inspecting correlator state.

GET /correlator/status

This is analogous to the information reported by the Apama `engine_watch` utility.

Response

```
<?xml version="1.0"?>
<map name="apama-response">
  <prop name="numConsumers">1</prop>
  <prop name="numOutEventsQueued">0</prop>
  <prop name="numOutEventsUnAcked">0</prop>
  <prop name="numOutEventsSent">62</prop>
  <prop name="uptime">1061116</prop>
  <prop name="numMonitors">15</prop>
  <prop name="numProcesses">16</prop>
  <prop name="numJavaApplications">0</prop>
  <prop name="numListeners">63</prop>
  <prop name="numEventTypes">110</prop>
  <prop name="numQueuedFastTrack">0</prop>
  <prop name="numQueuedInput">0</prop>
  <prop name="numReceived">3</prop>
  <prop name="numFastTracked">101</prop>
  <prop name="numEmits">227</prop>
  <prop name="numProcessed">260</prop>
  <prop name="numSubListeners">63</prop>
  <prop name="numContexts">1</prop>
  <prop name="virtualMemorySize">235624</prop>
  <prop name="numSnapshots">0</prop>
  <prop name="numInputQueuedInput">0</prop>
  <prop name="mostBackedUpInputContext"><none></prop>
  <prop name="mostBackedUpICQueueSize">0</prop>
  <prop name="mostBackedUpICLatency">0.0</prop>
</map>
```

`GET /correlator/info`

This is analogous to the information reported by the Apama `engine_inspect` utility.

Response

```
<?xml version="1.0"?>
<map name="apama-response">
  <list name="eventTypes">
    <map name="eventType">
      <prop name="nameSpace">com.apama.samples.vwap</prop>
      <prop name="name">Match</prop>
      <prop name="eventTemplates">0</prop>
    </map>
    <map name="eventType">
      <prop name="nameSpace">com.apama.samples.vwap</prop>
      <prop name="name">Trade</prop>
      <prop name="eventTemplates">1</prop>
    </map>
    <map name="eventType">
      <prop name="nameSpace">com.apama.samples.vwap</prop>
      <prop name="name">VwapWatch</prop>
      <prop name="eventTemplates">2</prop>
    </map>
  </list>
  <list name="timers">
    <map name="timer">
      <prop name="nameSpace"></prop>
      <prop name="name">wait</prop>
      <prop name="timers">1</prop>
    </map>
  </list>
  <list name="containerTypes"/>
  <list name="monitors">
    <map name="monitor">
      <prop name="nameSpace">com.apama.samples.vwap</prop>
      <prop name="name">Vwap</prop>
      <prop name="subMonitors">2</prop>
    </map>
  </list>
  <list name="javaApplications"/>
  <list name="aggregates"/>
  <list name="contexts">
    <map name="context">
      <prop name="name">main</prop>
      <prop name="subMonitors">2</prop>
      <prop name="queueSize">0</prop>
    </map>
  </list>
</map>
```

Managing and Monitoring over REST

IAF Management

The Apama REST API provides a URI to use with the `GET` method in order to return information about adapters running in the Apama Integration Adapter Framework (IAF). This information includes data such as the number of codecs and transports in use and the number of events sent and received.

`GET /iaf/status`

This returns the same information as the output of the Apama `iaf_watch` utility.

Response

```
<?xml version="1.0"?>
<map name="apama-response">
  <prop name="uptime">1235213</prop>
  <prop name="uptimeSinceLastReconfiguration">43432</prop>
  <prop name="numReconfigurations">3</prop>
  <prop name="numTransports">2</prop>
  <prop name="numCodecs">1</prop>
  <prop name="numDownstreamMappings">5</prop>
  <prop name="numUpstreamMappings">2</prop>
  <prop name="numApamaSinks">1</prop>
  <prop name="numApamaSources">1</prop>
  <prop name="numTransportReceived">441414</prop>
  <prop name="numTransportSent">34134</prop>
  <prop name="numApamaReceived">34134</prop>
  <prop name="numApamaSent">44414</prop>
  <list name="transportStatus">
    <map name="ATITransportStatus">
      <prop name="name">ATITransport</prop>
      <prop name="status">All OK</prop>
    </map>
    <map name="ReutersTransport">
      <prop name="name">ReutersTransport</prop>
      <prop name="status">Stuffed</prop>
    </map>
  </list>
  <list name="codecStatus">
    <map name="NullCodec">
      <prop name="name">NullCodec</prop>
      <prop name="status">Lovely</prop>
    </map>
  </list>
  <map name="mapperStatus">
    <prop name="name">Semantic Mapper</prop>
    <prop name="status">OK</prop>
  </map>
  <map name="apamaStatus">
    <prop name="name">Apama</prop>
    <prop name="status">OK</prop>
  </map>
</map>
```

Managing and Monitoring over REST

8 Event Correlator Utilities Reference

The Apama event correlator is at the heart of Apama applications. The correlator is Apama's core event processing and correlation engine. This section provides information and instructions for using command-line tools to monitor and manage event correlators.

The command-line tools documented in this book are in the `bin` directory of the Apama installation. By default, the installation directory is `/opt/apama_5.1` on UNIX, and `\Program Files \Software AG\Apama 5.1` on Windows.

For information about EPL, event definitions, monitors, namespaces and packages, see "Getting Started with Apama EPL" in *Developing Apama Applications in EPL* (available if you selected Developer during installation).

Starting the event correlator

This topic provides information about starting the event correlator. The executable for this tool is located in the `bin` directory of the Apama installation. Running the tool in the Apama Command Prompt ensures that the environment variables are set correctly.

This topic is organized as follows:

- "Synopsis" on page 104
- "Description" on page 105
- "Options" on page 105
- "Exit status" on page 111
- "Logging correlator status" on page 111
- "Text internationalization issues" on page 113
- "About the low latency and high throughput options" on page 113
- "Determining whether to disconnect slow receivers" on page 114
- "Determining whether to disable the correlator's internal clock" on page 119

Synopsis

To start the event correlator:

- On Windows, run `correlator.exe`.
- On UNIX, run `correlator`.

To obtain the following usage message, run the command without any options or with the `-h` option:

```
Usage: correlator [ options ]
Where options include:
  -V | --version                 Print version info and exit
  -h | --help                    This message
  -p | --port <port>             Listening on <port>
  -f | --logfile <file>          Correlator status log file name is <file>
  -v | --loglevel <level>        Log level is <level>
  -t | --truncate                Truncate log file at startup
  -I | --logicalID <id>          Component logical ID is <id>
  -N | --name <name>             Component name is <name>
  -l | --license <file>          License filename is <file>
  -m | --maxoutstandingack <num>  Buffer up to <num> secs of events/receiver
  -M | --maxoutstandingkb <num>  Buffer up to <num> kb of events/receiver
  -x | --qdisconnect           Disconnect slow event consumers
       --lowLatency            Tune output algorithm for low latency
       --highThroughput        Tune output algorithm for high throughput
       --logQueueSizePeriod <p>  Send info to log every <p> seconds
       --jmsConfig <dir>       Enable JMS messaging using configuration
                                 files in <dir>
       --distMemStoreConfig <dir> Enable Distributed MemoryStore using
                                 configuration files in <dir>
  -j | --java                  Enable Java application support
  -J | --javaopt <option>      Option to pass to JVM
       --replayLog <file>      Replay log file name is <file>
       --inputLog <file>       Input log file name is <file>
                                 (can only specify replay or input log)
  -XsetRandomSeed <num>        Set the seed of correlator random number
                                 generator to <num>
  -XignoreEnqueue              Ignore all enqueue statements (for replay)
  --replayMode                 Replay IDs from a replay log (implies
                                 -XignoreEnqueue)
  --inputQueueSize <num>       Set the capacity of the input queue
  -g | --nooptimize            Disable optimizations
  -P                           Enable persistence
  -PsnapshotInterval=<interval>  The persistent correlator snapshot interval
  -PstoreLocation=<path>       The path where the persistent correlator
                                 stores its recovery datastore
  -PadjustSnapshot=<true/false>  Automatically adjust the snapshot interval
  -PstoreName=<file>           The name of the recovery datastore file
  -Pclear                      Clear contents of recovery datastore if
                                 one exists
  -XrecoveryTime <num>         The time used after the recovery(seconds
                                 since the epoch)
  -noDatabaseInReplayLog       Disallow database dumps to Replay/Input log
  -Xclock | --externalClock    Use external clocking (&TIME events)
  -Xconfig | --configFile <file> Use service configuration file <file>
The loglevel argument must be one of:
  OFF, CRIT, FATAL, ERROR, WARN, INFO, DEBUG, TRACE
Note: Without specification of a license file, the correlator runs for
      30 minutes and accepts only local clients.
```

Description

By default, the `correlator.exe` (on Windows) or `correlator` (on UNIX) tool starts an event correlator process on the current host, and configures it to listen on port `15903` for monitoring and management requests.

On start-up, the executable displays the current version number and configuration settings.

To terminate an event correlator process, press Ctrl-C in the window in which it was started. Alternatively, you can issue the `engine_management` command with the `--shutdown` option. See "Shutting down and managing components" on page 136. Regardless of which technique you use to terminate the correlator, Apama first tries to shut down the correlator cleanly. If this is not possible, for example, perhaps because of a monitor in an infinite loop, Apama forces the correlator to shut down.

Options

When you start the correlator, there are a number of options that you can specify.

The following table describes the options you can specify when you start the correlator.

Option	Description
`-V | --version`	Displays version information for the correlator.

Option	Description
`-h | --help`	Displays usage information.
`-p` *port* |--port *port*	Specifies the port on which the event correlator should listen for monitoring and management requests. The default is `15903`.
`-f` *file* |--logfile *file*	Specifies the path of the status log file that the event correlator writes messages to. The default is `stdout`. See "Logging correlator status" on page 111.
`-v` *level* |--loglevel *level*	Specifies the level of information that the event correlator sends to the correlator status log file. In increasing amount of information order, the value must be one of the following: `OFF`, `CRIT`, `FATAL`, `ERROR`, `WARN`, `INFO`, `DEBUG`, `TRACE`. The default is `INFO`.
`-t | --truncate`	Specifies that if the correlator status log file already exists, the correlator should empty it first. The default is to append to it.
`-I` *id* |--logicalID *id*	Assigns a logical Id to the correlator. The logical Id must be an integer. The default is the value of the physical ID, which is a 19-digit integer. The correlator's messaging system generates the physical Id and ensures that it is unique in the scope of your network.
`-N` *name* |--name *name*	Assigns a name to the correlator. The default is `correlator`. If you are running a lot of correlators you might find it useful to assign a name to each correlator. A name can make it easier to use the `engine_management` tool to manage correlators and adapters.
`-l` *file* |--license *file*	Specifies the path to the file that contains the license string. If a license file is not specified, the correlator will start, but in a restricted mode that prevents connections to other hosts and it will terminate after 30 minutes of operation. If the license file exists but is expired or invalid then the correlator will fail to start.
`-m` *num* |--maxoutstandingack *num*	Specifies that you want the correlator to buffer messages for up to *num* seconds for each receiver that the correlator sends events to. The default is `10`. For additional information, see "Determining whether to disconnect slow receivers" on page 114.
`-M` *num* |--maxoutstandingkb *num*	Specifies that you want the correlator to buffer the events for each receiver up to the size in kb represented by *num*.
`-x | --qdisconnect`	Specifies that you want the correlator to disconnect receivers that are consuming events too slowly. For details, see "Determining whether to disconnect slow receivers" on page 114. The default is that the correlator does not disconnect slow receivers.

Option	Description
`--lowLatency`	Indicates that the correlator transport waits until an event is on the correlator's output queue. The transport then sends to receivers, in one batch, as many events as are available on the output queue. Because of thread scheduling effects, this can be more than the event that woke up the transport. This is the default behavior for multiprocessor machines. For additional information, see "About the low latency and high throughput options" on page 113.
`--highThroughput`	Indicates that the correlator transport waits 10 milliseconds before it tries to send any events to receivers. Typically, during that time, the correlator emits enough events to its output queue to form an efficient batch. This is the default behavior for uniprocessor machines. For additional information, see "About the low latency and high throughput options" on page 113.
`--logQueueSizePeriod` *p*	Sets the interval at which the correlator sends information to its log file. The default is 5 seconds. Replace *p* with a `float` value for the period you want.CAUTION: Setting `logQueueSizePeriod` to `0` turns logging off. Without correlator logging information, it is impossible to effectively troubleshoot problems.
`--jmsConfig` *`dir`*	Specifies that the correlator-integrated JMS messaging is enabled using the configuration files in the specified directory. Note that the configuration filenames must end with `*-spring.xml` and the correlator will not start unless the specified directory contains at least one configuration file. For more information on the configuration files for JMS messaging, see "Configuration files" on page 301.
`--distMemStoreConfig` *`dir`*	Specifies that the distributed MemoryStore is enabled, using the configuration files in the specified directory. Note that the configuration filenames must end with `*-spring.xml` and the correlator will not start unless the specified directory contains at least one configuration file. For more information on a distributed MemoryStore's configuration files, see "Using the Distributed MemoryStore" in *Developing Apama Applications in EPL*.
`-j | --java`	Enables support for Java applications. If you do not specify this option, any attempt to inject a Java application using `engine_inject -j` results in an error.
`-J` *option* |--javopt *option*	Specifies an option or property that you want the correlator to pass to the embedded JVM. You must specify the `-J` option for each JVM option. You can specify the `-J` option multiple times in the same `correlator.exe` command line. For example, `-J "-Da=value1" -J "-Db=value2" -J "-Xmx400m"`

Option	Description
	You cannot use this mechanism to pass the classpath to the JVM. The correlator sets the classpath implicitly as the last option, which overrides any value you might have set. Set the `CLASSPATH` environment variable if you want to set a particular classpath.
`--replayLog`*file*	Specifies the path of the replay log file. The event correlator writes incoming messages to the replay log. If there is a problem with your application, Software AG Global Support can use the replay log to diagnose it. See "Using a replay log to diagnose problems" on page 157. A full replay log lets you replay the exact same behavior, ordering of emit operations, and so on, as the run that generated the replay log.
`--inputLog`*file*	Specifies the path of the input log file. The event correlator writes only input messages to the input log file. If there is a problem with your application, Software AG Global Support can use the input log to diagnose it. See "Using a replay log to diagnose problems" on page 157. An input-only replay log contains only the external inputs to the correlator. There is no information about multi-context ordering. Consequently, if there is more than one context, there is no guarantee that you can replay execution in the same order as the original execution.
`--frequency`*hz*	Instructs the correlator to generate clock ticks at a frequency of *hz* per second. Defaults to `10`, which means there is a clock tick every 0.1 seconds. Be aware that there is no value in increasing *hz* above the rate at which your operating system can generate its own clock ticks internally. On UNIX and some Windows machines this is 100 Hz and on other Windows machines it is64 Hz.
`-XsetRandomSeed` *int*	Starts the correlator with the random seed value you specify. Specify an integer whose value is in the range of 1 to 2 32.The correlator uses the random seed to calculate the random numbers returned by the `integer.rand()` and `float.rand()` functions. The same random seed returns the same sequence of random numbers. This option is useful when your application uses the `integer.rand()` and `float.rand()` functions and you are capturing correlator behavior in a replay log. If you need to reproduce correlator behavior from that replay log, you will want the correlator to generate the same random numbers as it generated when the original behavior was captured.
`-XignoreEnqueue`	Instructs the correlator to ignore the output of `enqueue` statements. This is required when "Reproducing correlator behavior from a replay log" on page 160.
`--replayMode`	Enables replay of correlator behavior previously captured in a replay log. Always specify the `--replayMode` option when you start a

Option	Description
	correlator that will be reproducing behavior captured in a replay log. Specify this option in addition to the other options described in .
`--inputQueueSize` *int*	Sets the maximum number of spaces in every context's input queue. The default is that each input queue has 20,000 spaces. If events are arriving on an input queue faster than the correlator can process them the input queue can fill up. You can set the `inputQueueSize` option to allow all input queues to accept more events. Typically, the default input queue size is enough so if you find that you need to increase the size of the input queue you should try to understand why the correlator cannot process the events fast enough to leave room on the default-sized queue. If you notice that adapters or applications are blocking it might be because a public context's input queue is full. To determine if a public context's input queue is full, use output from the `engine_inspect` utility in conjunction with the status messages in the correlator log file.
`-g | --nooptimize`	Disables some correlator optimizations. Specify this option when you plan to run the `engine_debug` utility. If you run the `engine_debug` utility and you did not specify the `-g` option when you started the correlator, the optimizations hinder the debugging process. For example, the correlator might simultaneously execute multiple statements over multiple lines even if you are using debugger commands to step through the program line by line. When you use Apama Studio, it automatically uses the `-g` option when it starts a correlator from a debug launch configuration.
`-P`	Enables correlator persistence. You must specify this option to enable correlator persistence. If you do not specify any other correlator persistence options, the correlator uses the default persistence behavior as described in "Enabling correlator persistence" in *Developing Apama Applications in EPL* (available if you selected Developer during installation). If you specify one or more additional correlator persistence options, the correlator uses the settings you specify for those options and uses the defaults for the other persistence options.
`-PsnapshotInterval=`*`interval`*	Specifies the period between persistence snapshots. The default is 200 milliseconds.
`-PstoreLocation=`*`path`*	Specifies the path for the directory in which the correlator stores persistent state. The default is the current directory, which is the directory in which the correlator was started.
`-PadjustSnapshot=`*`boolean`*	Indicates whether or not the correlator should automatically adjust the snapshot interval according to application behavior. The default is true, which means that the correlator does automatically adjust the snapshot interval. You might want to set this to false to diagnose a problem or test a new feature.

Option	Description
`-PstoreName=`*`filename`*	Specifies the name of the file that contains the persistent state. This is the recovery datastore. The default is `persistence.db`.
`-Pclear`	Specifies that you want to clear the contents of the recovery datastore. This option applies to the recovery datastore you specify for the `-PstoreName` option or to the default `persistence.db` file if you do not specify the `-PstoreName` option. When the correlator starts it does not recover from the specified recovery datastore.
`-XrecoveryTime` *`num`*	For correlators that use an external clock, this is a time expression that represents the time of day that a correlator starts at when it recovers persistent state and restarts processing. The default is the time expression that represents the time of day captured in the last committed snapshot. This option is useful only for replaying replay logs that contain recovery information. To change the default, specify a number that indicates seconds since the epoch.
`-noDatabaseInReplayLog`	Specifies that the correlator should not copy the recovery datastore to the replay/input log when it restarts a persistence-enabled correlator. The default is that the correlator does copy the recovery datastore to the replay/input log upon restarting a persistence-enabled correlator. You might set this option if you are using an input log as a record of what the correlator received. The recovery datastore is a large overhead that you probably do not need. Or, if you maintain an independent copy of the recovery datastore, you probably do not want a copy of it in the replay log.
`-Xclock` |--externalClock	Instructs the correlator to disable its internal clock. By default, the correlator uses internally generated clock ticks to assign a timestamp to each incoming event. When you specify the `-Xclock` option, you must send time events (`&TIME`) to the correlator. These time events set the correlator's clock. For additional information, see "Determining whether to disable the correlator's internal clock" on page 119.
`-Xconfig` *file* |--configFile *file*	Specifies a special configuration file that the correlator uses to initialize its networking. Specify this option only as directed by Apama Technical Support to troubleshoot networking problems. For more information, see "Using the Apama Component Extended Configuration File" on page 363.
`--runtime` *`mode`*	On Linux 64-bit systems, you can specify whether you want the correlator to use the compiled runtime or the interpreted runtime, which is the default. Specify `--runtime compiled` or `--runtime interpreted`. The interpreted runtime compiles EPL into bytecode whereas the compiled runtime compiles EPL into native code that is directly executed by the CPU. For many applications, the compiled runtime

Option	Description
	provides significantly faster performance than the interpreted runtime. Applications that perform dense numerical calculations are most likely to have the greatest performance improvement when the correlator uses the compiled runtime. Applications that spend most of their time managing listeners and searching for matches among listeners typically show a smaller performance improvement. Other than performance, the behavior of the two runtimes is the same except • The interpreted runtime allows for the profiler and debugger to be switched on during the execution of EPL. The compiled runtime does not permit this. For example, you cannot switch on the profiler or debugger in the middle of a loop. • The amount of stack space available is different for the two runtimes. This means that recursive functions run out of stack space at a different level of recursion on the two runtimes.

Exit status

The `correlator.exe` tool returns the following exit values:

Table 2. Exit values for correlator.exe

Value	Description
0	The event correlator terminated successfully.
1	An error occurred which caused the event correlator to terminate abnormally.

Event Correlator Utilities Reference

Logging correlator status

The correlator sends information to its status log file every five seconds or at an interval that you set with the `--logQueueSizePeriod` option. When logging at `INFO` level, this information contains the following:

```
Status: sm=2 nctx=2 ls=3 rq=0 eq=0 iq=0 oq=0 icq=12 lcn="input ctx one"
lcq=12 lct=0.8 rx=5 tx=20 rt=7 nc=1 vm=369768 runq=0 si=0.0 so=0.0
```

Where the fields are as follows:

Table 3. Correlator status log fields

Field	Description
sm	Number of monitor instances, sometimes called sub-monitors. This is the sum of monitor instances in the main context plus monitor instances in any other context.

Field	Description
`nctx`	Number of contexts. For applications developed prior to Apama 4.1, this is always `1`.
`ls`	Number of listeners. This is the sum of listeners in the main context plus listeners in any other context. This includes each `on` statement and each stream source template, for example, `all Tick(symbol=”APMA”)` in the following: `stream<Tick> ticks := all Tick(symbol=”APMA”);`
`rq`	Sum of the number of routed events on the input queues of all contexts. A routed event is any event that has been generated by the EPL’s `route` keyword or Java’s `route()` method. A routed event goes to the front of that context's input queue. This ensures that the correlator processes routed events before processing external events. This number can go up and down, and it tends to be `0` for an idle correlator.
`eq`	Number of events on the enqueued events queue. An enqueued event is any event that has been generated with the EPL `enqueue` keyword (not `enqueue...to`) or Java’s `enqueue()` method. A separate thread moves events from the enqueued events queue to the input queue of each public context. The size of the enqueued events queue is unbounded. Consequently, it is possible for this queue to use a large amount of memory if one or more input queues are full.
`iq`	Sum of the number of entries on the input queues of all contexts. This number excludes routed events. It includes events from external sources, injections of EPL files, delete requests, time ticks, pending spawn-to operations, and enqueued events. This number goes up and down, and tends to be `0` for an idle correlator. It is possible for the total number of input queue entries (`iq`) to be greater than the number of events the correlator has received from external sources (`rx`).
`oq`	Number of events on the correlator’s output queue. This is the number of events that the correlator has emitted but not yet sent to any receivers. If the correlator is idle, this number is `0`.
`icq`	Sum of enqueued events on the input queues of all public contexts. These events are a subset of the events included in the `iq` count.
`lcn`	The name of the public context whose input queue is most backed up in time. This is not necessarily the public context whose input queue has the largest number of entries. If no public context has entries on its input queue then this value is `"<none>"`. The name of the main context is always `main`.
`lcq`	For the context identified by `lcn`, this is the number of entries on the most backed up input queue.
`lct`	For the context identified by `lcn`, this is the time difference between its current logical time and the most recent time tick added to its input queue.
`rx`	Number of events the correlator has received from external sources since the correlator was started. The correlator increments this count as soon as it receives an event. After incrementing this count, the correlator parses the event to determine if it is an event for which a definition has already been injected. If the correlator can successfully parse the

Field	Description
	event, the event goes to the input queue of each public context. If the correlator cannot parse the event, the correlator discards the event. This is not the number of events that the correlator has processed. This count does not include routed and enqueued events. This number never goes down; it can only go up.
tx	Number of events the correlator has sent to receivers since the correlator was started. This number includes duplicate events sent to multiple receivers. For example, suppose you inject EPL code that emits an event, and there are five receivers that are subscribed to channels that publish that event. In this situation, the tx count goes up by five. Although there was 1 event, it was sent five times — once to each subscribed receiver.
rt	Total number of events that have been routed across all contexts since the correlator was started.
nc	Number of receivers.
vm	Number of kilobytes of virtual memory being used by the correlator process.
runq	Number of contexts (public and private) that have work to do but are not currently running. These contexts are waiting for processing resources. This indicator is a measure of the load on the system. When this number is consistently more than 0 and latency is a problem you might want to consider adding CPUs to your configuration.
si	The rate (pages per second) at which pages are being read from swap space.
so	The rate (pages per second) at which pages are being written to swap space.

Logging for correlators with JMS support enabled

Correlators with Correlator-integrated JMS support enabled send additional information to the status log file. For details on this information, see "Logging JMS enabled correlator status" on page 292.

Starting the event correlator

Text internationalization issues

Apama translates the contents of the correlator status log from UTF-8 to the local character set before displaying it in a console or terminal.

Starting the event correlator

About the low latency and high throughput options

There are two correlator options that determine the behavior of the correlator with regard to low latency and high throughput. Exactly one of these options is always set. The default varies:

- On multiprocessor machines, the --lowLatency option is set.

- On uniprocessor machines, the `--highThroughput` option is set.

Typically, you do not need to change which option is set. However, if performance is not satisfactory, changing from one option to the other might have a positive effect. Some background about how the correlator works is required to determine whether to change which option is set.

Within the correlator, EPL emits one event at a time. However, when the Apama transport sends events from the correlator to receivers, there is a fixed amount of overhead regardless of whether the transport sends one event or a batch of events. In other words, it is more efficient to send batches of events rather than single events. Consequently, the correlator puts emitted events on a queue. This keeps processing and communications separate, and allows flexibility with regard to latency and throughput.

The Apama transport pulls events off the correlator’s queue and sends them to receivers. The behavior of the transport is determined by which option is set:

- Low latency — When `--lowLatency` is set, the transport waits until an event is on the correlator’s queue. The transport then sends, in one batch, as many events as are available on the queue. Because of thread scheduling effects, this can be more than the event that woke up the transport. This is the default behavior for multiprocessor machines.
- High throughput — When `--highThroughput` is set, the transport waits 10 milliseconds before it tries to send any events. Typically, during that time, the correlator emits enough events to its output queue to form an efficient batch. This is the default behavior for uniprocessor machines.

As you can imagine, waiting 10 milliseconds has a negative impact on the latency of the system. On multiprocessor systems, waiting 10 milliseconds is not desired. However, on uniprocessor machines, the consequences for not waiting for a batch of events to transport is too great and so the default is high throughput.

Apama technical support recommends that you run a correlator on a system that has at least dual CPUs.

Starting the event correlator

Determining whether to disconnect slow receivers

The correlator sends events to multiple receivers. Sometimes, a receiver cannot consume its events fast enough for the correlator to continue sending them. When this happens, the default behavior is that the correlator suspends processing until the receiver disconnects or starts consuming events fast enough. In other words, a slow receiver can prevent other consumers from receiving events. However, you might prefer to have the correlator disconnect a slow receiver and continue processing and sending events to other consumers. Information in this section can help you determine whether to disconnect slow receivers.

See also "Handling slow or blocked receivers", in the testing and tuning chapter of *Developing Apama Applications in EPL* (available if you selected Developer during installation).

Starting the event correlator

Description of slow receivers

Every event that the correlator sends to one of its receivers has a sequence number. After a receiver processes an event, it sends the event's sequence number back to the correlator as an acknowledgment that the receiver processed that event. By default, if the correlator does not receive an acknowledgment within `10` seconds after the correlator sent the event, the correlator marks that receiver as being slow to consume events.

You can control the length of time within which the receiver must acknowledge an event before it is marked as a slow receiver. When you start the correlator, you can specify the `-m` (or `--maxoutstandingack`) option and specify a number.

For example:

```
correlator -l etc\license.txt -m 15
```

If you start the correlator with this command, the correlator marks a receiver as slow if the correlator does not receive an acknowledgment within `15` seconds. If you do not specify the `-m` option, the default is `10` seconds. You should not specify a value under `1` second because doing so raises the risk that the correlator might designate a receiver as slow when it is in fact not slow.

The mechanism that flags a receiver as slow is not precise. If a receiver does not acknowledge an event sequence after `10` seconds (the default setting), the correlator does not immediately designate the receiver as slow. Typically, the designation happens within the next `5` seconds. If you change the value of the `-m` option, the slow designation takes effect between `1` and `1.5` times the value of the `-m` option.

Starting the event correlator

Sending events to correlators

The `engine_send` tool sends Apama-format events into an event correlator or IAF adapter. The executable for this tool is located in the `bin` directory of the Apama installation. Running the tool in the Apama Command Prompt ensures that the environment variables are set correctly.

If the events you want to send are not in Apama format, you must use an adapter that can transform your event format into Apama event format.

Synopsis

To send Apama-format events to an event correlator or IAF adapter:

- On Windows, run `engine_send.exe`.
- On UNIX, run `engine_send`.

To obtain the following usage message, run the command without any options or with the `-h` option:

```
Usage: engine_send [ options ] [ file1 [ file2 ... ] ]
In order to send to a Correlator or IAF.
Where options include:
    -h | --help               This message
    -n | --hostname <host>    Connect to an engine on <host>
    -p | --port <port>        Engine is listening on <port>
    -l | --loop <count>       Loop through input <count> times
    -v | --verbose            Be more verbose
    -u | --utf8               Assume input is in UTF8
    -V | --version              Print program version info
    Use '-' to read from <stdin>
    Loop count < 0 means loop forever
```

Description

The engine_send tool sends Apama-format events to an event correlator. In Apama-format event files, you can specify whether to send the events in batches of one or more events or at set time intervals.

The correlator reads events from one or more specified files. Alternatively, you can specify "-"or not specify a filename and the correlator reads events from stdin until it receives an end-of-file signal (Ctrl-D on UNIX and Ctrl-Z on Windows).

For details about Apama-format events, see "Event file format" on page 163.

By default, the engine_send tool is silent unless an error occurs. To view progress information during engine_send execution, specify the -v option when you invoke engine_send.

You can also use engine_send to send events directly to the Integration Adapter Framework (IAF). To do this, specify the port of the IAF, by default this is 16903.

Options

The engine_send tool takes the following command line options:

Option	Description
-h	Displays usage information. Optional.
-n *host*	Name of the host on which the event correlator to which you want to send events is running. Optional. The default is localhost. Note: Non-ASCII characters in host names are not supported.
-p *port*	Port on which the event correlator is listening. Optional. The default is 15903.
-v	Requests verbose output during execution. Optional.
-V	Displays version information for the engine_send tool. Optional.
-l *loop*	Number of times to cycle through and send the input events. Optional. Replace *loop* with one of the following values: • 0 — Indicates that you want the engine_send tool to iterate through and send the input data once. This is the default. • Any negative integer— Indicates that you want the engine_send tool to indefinitely cycle through and send the input events. • Any positive integer — Indicates the number of times to cycle through and send the input events. The engine_send tool ignores this option if you specify it and the input is from stdin.
-u	Indicates that input files are in UTF-8 encoding. This specifies that the engine_send utility should not convert the input to any other encoding.

Operands

The `engine_send` tool takes the following operands:

`[` *file1* `[` *file2* `... ] ]`	Specify zero, one, or more files that contain event data. Each file you specify must comply with the event file format described in "Event file format" on page 163. If you do not specify any filenames, the `engine_send` tool takes input from `stdin`.

Exit status

The following exit values are returned:

0	The events were sent successfully.
1	No connection to the event correlator was possible or the connection failed.
2	One or more other errors occurred while sending the events.

Operating notes

To end an indefinite cycle of sending events, press Ctrl-C in the window in which you invoked the `engine_send` tool.

You might want to indefinitely cycle through and send events in the following situations:

- In test environments. For example, you can use `engine_send` to simulate heartbeats. If you then kill the `engine_send` process, you can test your EPL code that detects when heartbeats stop.
- In production environments. For example, you can use the `engine_send` tool to initialize a large data table in the correlator.

Text encoding

By default, the `engine_send` tool checks the environment variable or global setting that specifies the locale because this indicates the local character set. The `engine_send` tool then translates EPL text from the local character set to UTF-8. Consequently, it is important to correctly set the machine's locale.

However, some input files might start with a UTF-8 Byte Order Mark. The `engine_send` tool treats such input files as UTF-8 and does not do any translation. Alternatively, you can specify the `-u` option when you run the `engine_send` tool. This forces the tool to treat each input file as UTF-8.

Description of slow receivers

How frequently slow receivers occur

In practice, sending acknowledgements should not be slow because a dedicated thread sends acknowledgments. Network interruptions are the most common cause of delayed acknowledgments. Of course, network interruptions affect events being sent as well.

Most receivers, including the `engine_receive` tool, normally send acknowledgments `0.1` seconds after the message was sent. Consequently, there is very little chance of a receiver being mistakenly designated as slow. In production, slow receivers should be rare as long as you have done the appropriate load testing before deployment.

If an engine library client blocks in the middle of a `sendEvents` call, the receiver cannot acknowledge messages while the client is blocked. As you know, a receiver is made up of an engine library and a client. Clients receive events by registering a `sendEvents` callback with the engine library. When the engine library gets an event from the correlator, it calls `sendEvents`. Problems that can cause a client to block are typically related to I/O, networking, or synchronization. The `sendEvents` call cannot complete until the problem is resolved. The receiver cannot send the acknowledgement until the `sendEvents` call completes. For example, the `engine_receive` tool is a receiver that is made up of an engine library and a client whose `sendEvents` callback writes events received to a disk file. If the client has to wait for the disk, then it is blocked. The likelihood of a `sendEvents` callback being blocked depends on what the client is doing. If the client is writing to a local disk, the process might block sometimes, but never more than a fraction of a second. However, sending the events over a slow or unreliable network might block for a while if the network, or the remote system cannot keep up with the event rate.

During development of event consumers, however, slow receivers are more likely. This can happen when a newly developed consumer receives an event from the correlator but cannot send the acknowledgment because of a deadlock. Another development problem might be that the event consumer cannot keep up with the load. If you have problems with slow receivers during development, you probably need to evaluate the design of your application.

Description of slow receivers

Correlator behavior when there is a slow receiver

When the correlator has a slow receiver, it can behave in one of two ways:

- The default behavior is that the correlator blocks further processing. This is because a slow receiver causes the correlator's event output queue to become full. When the queue is full, the correlator stops processing because it has no place to put events. The processing thread stops and does not execute any more EPL code. The transport thread does not send any more events to any of the correlator's other receivers. The correlator resumes processing when the slow receiver disconnects or acknowledges the outstanding sequence number.
- The correlator disconnects the slow receiver, and continues processing events and sending them to its other receivers. To obtain this behavior, you specify the `-x` (or `--qdisconnect`) option when you start the correlator. The correlator sends a message to the receiver to indicate that the correlator is disconnecting the receiver. It is up to the receiver to reconnect.

To ensure that it has received an acknowledgment for every event sent, the correlator buffers each event that it sends until it receives the message's corresponding acknowledgment. When there is a slow receiver, this can use a lot of memory if the correlator is sending a large number of messages.

Description of slow receivers

Tradeoffs for disconnecting a slow receiver

When you specify the `-x` option when you start the correlator, it means that the correlator always disconnects a slow receiver. There are two main disadvantages to this:

- The correlator loses the events that it sent to that receiver.
- It is possible for the correlator to disconnect a receiver that is temporarily overloaded, and to therefore lose events unnecessarily.

Clearly, losing events can be a very serious problem. This is why the default is that the correlator does not disconnect slow receivers.

The advantage of disconnecting a slow receiver is that the correlator continues processing events.

The correlator always sends a warning message to its status log when it detects a slow receiver. This lets you see where there are potential problems.

If you cannot allow the correlator to lose events, do not specify the -x option when you start the correlator.

Description of slow receivers

Determining whether to disable the correlator's internal clock

When you start the correlator, you can specify the -Xclock option to disable the correlator's internal clock. By default, the correlator uses internally generated clock ticks to assign a timestamp to each incoming event. When you specify the -Xclock option, you must send time events (&TIME) to the correlator. These time events set the correlator's clock.

Use &TIME events in place of the correlator's internal clock when you want to reproduce the historic behavior of an application. For example, Apama Studio's Data Player starts a correlator with a command that specifies the -Xclock option. The Data Player then sends &TIME events that let you play back events from the database.

A situation in which you might want to generate &TIME events is when you want to run experiments at faster than real time but still obtain correct timestamp behavior. In this situation, the correlator uses the externally generated time events instead of real time to invoke timers and wait events.

Disabling the correlator's internal clock, and sending external time events, affects all temporal operations, such as timers and wait statements.

Regardless of whether the correlator generates internal clock ticks, or receives external time events, the correlator assigns a timestamp to each incoming event. The timestamp indicates the time that the event arrived on the context's input queue. The value of the timestamp is the same as the last internally-generated clock tick or externally-generated time event. For example, suppose you have the following events and clock ticks:

```
&TIME(1)
A()
B()
&TIME(2)
C()
```

A and B receive a timestamp of 1. C receives a timestamp of 2.

For additional information about using external time events, see, "Apama Concepts" in *Introduction to Apama* (available if you selected Developer during installation).

Description of slow receivers

Injecting code into a correlator

To inject EPL files, Apama Java applications, or Correlator Deployment Packages (CDPs) into the correlator invoke the engine_inject tool. The executable for this tool is located in the bin directory of the Apama installation. Running the tool in the Apama Command Prompt ensures that the environment variables are set correctly.

Synopsis

To inject applications into the event correlator:

- On Windows, run `engine_inject.exe`.
- On UNIX, run `engine_inject`.

To obtain the following usage message, run the command without any options or with the `-h` option:

```
Usage: engine_inject [ options ] [ file1 [ file2 ... ] ]
In order to inject to a correlator.
Where options include:
-h | --help             This message
-n | --hostname <host> Connect to an engine on <host>
-p | --port <port>     Engine is listening on <port>
-v | --verbose          Be more verbose
-u | --utf8             Assume input is in UTF8
-V | --version          Print program version info
-j | --java             Operate on Java applications rather than EPL or CDP
-c | --cdp              Operate on CDP files rather than EPL or Java
-s | --hashes           Print out hashes of (UTF8-encoded) files, rather
                        than injecting, to obtain hashes for Java or CDP
                        files also specify -j or -c
 Use '-' to read from <stdin>
```

Description

The `engine_inject` tool reads application definitions from the specified file(s) and injects them into an event correlator. If you do not specify a filename, or if you specify `"—"` as the filename, the correlator reads data from the standard input device (`stdin)` until you indicate the end of the file: Ctrl-D on UNIX and Ctrl-Z on Windows.

Application definitions can be monitors scripted in Apama's EPL language. For more information on EPL, see "Introduction to Apama Event Processing Language" in *Developing Apama Applications in EPL* (available if you selected Developer during installation). Alternatively, you can specify the `-j` or `-c` options. The `-j` option specifies that you will inject an application written in Java. The `-c` option specifies that you will inject a Correlator Deployment Package file.

When you specify the `-j` option, each file you inject must be a Java archive file (`JAR`) that contains a single Java application. For more information, see "Overview of Apama Java Applications" in *Developing Apama Applications in Java* (available if you selected Developer during installation).

When you specify the `-c` option, the file you inject must be an Apama Correlator Deployment Package (CDP). For more information on preparing a CDP, see "Packaging EPL and Java code" on page 124.

By default, the `engine_inject` tool is silent unless an error occurs. To view information about `engine_inject` execution, specify the `--verbose` option.

If you try to inject invalid EPL files or invalid Apama Java applications, the event correlator generates an error. None of the application data in the invalid file is loaded. The `engine_inject` tool terminates. If you specify multiple EPL or Java files for injection the `engine_inject` tool injects all of them or terminates when it reaches the first file that contains an error. For example:

```
engine_inject 1.mon 2.mon 3.mon
```

If the `2.mon` file contains an error then `engine_inject` successfully injects `1.mon` and then terminates when if finds the error in `2.mon`. The tool does not operate on `3.mon`.

If you try to inject a CDP the correlator processes each EPL file packaged in the CDP separately. If one file in a CDP contains an error then the correlator reports an error for that file and does not run it but it does run the other files in the CDP (if they have no errors). It does not matter which file in the CDP contains the error. That is, the first file in the CDP that the correlator processes can contain an error and the correlator still runs the other files in the CDP if they contain no errors.

Options

The following table describes the options you can specify when you inject applications into the correlator:

Option	Description
`-h`	Displays usage information
`-n` *host*	Name of the host on which the event correlator is running. The default is `localhost`. Note: Non-ASCII characters in host names are not supported.
`-p` *port*	Port on which the event correlator is listening. The default is `15903`.
`-v`	Requests verbose output during `engine_inject` execution.
`-u`	Indicates that input files are in UTF-8 encoding. The default is that the `engine_inject` tool assumes that the EPL files to be injected are in the native character set of your platform. Set the `-u` option to override this assumption. The `engine_inject` tool then assumes that all input files are in UTF-8.
`-V`	Displays version information for the `engine_inject` tool.
`-j`	Indicates that each operand is a Java archive file (`JAR` file) that contains a single Java application.
`-c`	Indicates that each operand is a Correlator Deployment Package file.
`-s`	Indicates that instead of injecting the specified files you want to print the hashes (UTF8-encoded) for the files. If `engine_inject` is operating on Java or Correlator Deployment Package (CDP) files then you must also specify `-j` or `-c`.

Operands

The `engine_inject` tool takes the following operands:

[*file1* [*file2* ...]]	The names of zero or more files that contain application data in Apama EPL, Apama Java, or Correlator Deployment Package files. If you do not specify one or more filenames, the `engine_inject` tool takes input from `stdin`.

Exit status

The `engine_inject` tool returns the following exit values:

Status	Description
`0`	All definitions were injected into the event correlator successfully.

Status	Description
1	No connection to the event correlator was possible or the connection failed.
2	Other error(s) occurred while injecting the supplied definitions.

Text encoding

By default, the `engine_inject` tool uses the default system encoding to determine the local character set. The `engine_inject` tool then translates all submitted EPL text from the local character set to UTF-8. Consequently, it is important to correctly set the machine's locale.

However, some input files might start with a UTF-8 Byte Order Mark. The `engine_inject` tool treats such input files as UTF-8 and does not do any translation. Alternatively, you can specify the `-u` option when you run the `engine_inject` tool. This forces the tool to treat each input file as UTF-8.

Event Correlator Utilities Reference

Deleting code from a correlator

The `engine_delete` tool removes EPL code and Apama Java applications from the correlator. The executable for this tool is located in the `bin` directory of the Apama installation. Running the tool in the Apama Command Prompt ensures that the environment variables are set correctly.

Synopsis

To remove applications from the event correlator:

- On Windows, run `engine_delete.exe`.
- On UNIX, run `engine_delete`.

To obtain the following usage message, run the command without any options or with the `-h` option:

```
Usage: engine_delete [ options ] [ name1 [ name2 ... ] ]
To delete named objects from a correlator.
Where options include:
    -h | --help                 This message
    -n | --hostname <host>      Connect to an engine on <host>
    -p | --port <port>          Engine is listening on <port>
    -f | --file <filename>      Read names from <filename>
    -F | --force                Force deletion of names even if they are in use
    -k | --kill                 Kill name even if it is a running monitor
    -a | --all                  Delete everything in the engine - use with care
    -y | --yes                  Don't ask 'are you sure?' when deleting all
    -v | --verbose              Be more verbose
    -u | --utf8                 Assume input is in UTF8
    -V | --version              Print program version info
    Use '-' to read from <stdin>
    Multiple -f options may be given
```

Description

The `engine_delete` tool deletes named applications, monitors and event types from an event correlator. Names are the full package names as previously assigned to an application monitor or event type when injected into the event correlator.

To specify the items you want to delete, you can specify any one of the following in the `engine_delete` command line:

- Names of the items to delete.
- The `-f` option with the name of a file that contains the names of the items you want to delete. In this file, specify each name on a separate line.
- Neither of the above. In this case, the `engine_delete` tool reads names from `stdin` until you type an end-of-file signal, (`CTRL-D` on UNIX and `CTRL-Z` on Windows). If you want, you can specify "-" in the command line to indicate that input will come from `stdin`.

The tool is silent by default unless an error occurs. To receive progress information, specify the `-v` option.

The tool permits two kinds of operations — delete and kill. These cause different side-effects and you must use them carefully.

- When you delete a monitor, the correlator tries to terminate all of that monitor's instances. If they are responsive (not in some deadlocked state) each one executes its `ondie` action, and when the last one exits the correlator calls the monitor's `onunload` action. This assumes that the monitor you are deleting defines `ondie` and `onunload` actions.

If a monitor instance does not respond to a delete request, the correlator cannot invoke the monitor's `onunload` action. In this case, you must kill, rather than delete, the monitor instance.

- When you kill a monitor, the correlator immediately terminates all of the monitor's instances, without invoking `ondie` or `onunload` actions.

Options

The tool `engine_delete` tool takes the following command line options:

Option	Description
`-h`	Displays usage information. Optional.
`-n` *`host`*	Name of the host on which the event correlator is running. Optional. The default is `localhost`. Note: Non-ASCII characters in host names are not supported.
`-p` *`port`*	Port on which the event correlator is listening. Optional. The default is `15903`.
`-f` *`filename`*	Indicates that you want the `engine_delete` tool to read names of items to delete from the specified file. In this file, each line contains one name. Optional. The default is that input comes from `stdin`.
`-F`	Forces deletion of named event types even if they are still in use — that is, they are referenced by active monitors or applications. A forced delete also removes all objects that refer to the event type you are deleting. For example, if monitor `A` has listeners for `B` events and `C` events and you forcibly delete `C` events the operation deletes monitor `A`, which of course means that the listener for `B` events is deleted. Optional. The default is that event types that are in use are not deleted.
`-k`	Kills the named item (or items) regardless of whether it is in use. You can specify this option to remove a monitor or application that is stuck in an infinite loop.

Option	Description
`-a`	Forces deletion of all applications, monitors, and event types. The correlator finishes processing any events on input queues and then does the deletions. Any events sent after invoking `engine_delete -a` are not recognized. Specifying this option does not stop a monitor that is in an infinite loop. You must explicitly kill such monitors. Specifying the `-a` option is equivalent to specifying the `-F` option and naming every object in the correlator. If you want to kill every object in the correlator, shut down and restart the correlator. See "Shutting down and managing components" on page 136.
`-y`	Removes the "are you sure?" prompt when using the `-a` option.
`-v`	Requests verbose output.
`-u`	Indicates that input files are in UTF-8 encoding. This specifies that the `engine_delete` utility should not convert the input to any other encoding.
`-V`	Displays version information for the `engine_delete` tool.

Operands

The `engine_delete` tool takes the following operands:

`[ name1 [ name2 ... ] ]`	The name of zero or more EPL or Apama Java applications, monitors and/or event types to delete from the event correlator. If you do not specify at least one item name, and you do not specify the `-f` option, the `engine_delete` tool expects input from `stdin`.

Exit status

The following exit values are returned:

Status	Description
0	The items were deleted from the event correlator successfully.
1	No connection to the event correlator was possible or the connection failed.
2	Other error(s) occurred while deleting the named items.

Event Correlator Utilities Reference

Packaging EPL and Java code

The `engine_package` tool assembles EPL files and `.jar` files into a Correlator Deployment Package (CDP). You can inject a CDP file into the correlator just as you inject an EPL file or a JAR file containing an Apama Java application. CDP files use a proprietary, non-plaintext format that treats files in a manner similar to the way a JAR file treats a collection of Java files. In addition, using a CDP file guarantees that all files, assuming no errors, are injected and are injected in the correct order. See

"Injecting code into a correlator" on page 119 for details about how the correlator handles an error in a file that is in a CDP.

While the names of events, monitors, aggregates, and `.jar` files that are contained in a CDP file are visible to the correlator utilities `engine_inspect`, `engine_manage`, and `engine_delete`, the code that defines them is not.

The executable for this tool is located in the `bin` directory of the Apama installation. Running the tool in the Apama command prompt ensures that the environment variables are set correctly.

Synopsis

To package files into a CDP file:

- On Windows, run `engine_package.exe`.
- On UNIX, run `engine_package`.

To obtain the following usage message, run the command without any options or with the `-h` option:

```
Usage: engine_package [ options ] [ file1 [ file2 ... ] ]
In order to create a Correlator Deployment Package (CDP) from multiple files.
  Where options include:
    -h | --help             This message
    -V | --version          Print program version info
    -o | --output  <file>   Write the CDP to <file>
    -m | --manifest <file>  Create the CDP from the manifest at <file>
    -u | --utf8             Assume input is in UTF-8
```

Description

The `engine_package` tool creates a Correlator Deployment Package (CDP). A CDP file contains one or more files. You specify the name of the CDP file to create as an argument to the `-o` option.

You can specify the files you want to include on the command line or you can use the `-m` option and specify a manifest file that contains the names of the files. The manifest file is a text file; each line in the file specifies a relative or absolute path to a file. Files should be listed in the order in which you want them to be injected into the correlator.

Options

The following table describes the options you can specify when you package files into a Correlator Deployment Package:

Option	Description
`-h`	Displays usage information
`-V`	Displays version information for the engine_package tool.
`-o` *filename*	Name of the CDP file to create. Required.
`-m` *filename*	Name of the manifest file that lists the files you want to package.
`-u`	Indicates that input files are in UTF-8 encoding. The default is that the `engine_package` tool assumes that the files to be packaged are in the native character set of your platform. Set the `-u` option to override this assumption. The `engine_package` tool then assumes that all input files are in UTF-8.

Operands

The engine_package tool takes the following operands:

[*file1* [*file2* ...]]	The names of the EPL or .jar files that contain code. The order in which these files are specified will become the order in which they are injected into the correlator when the CDP file is injected. Instead of listing the files on the command line, you can list them in a manifest file and use the -m option.

Exit status

The engine_package tool returns the following exit values:

Status	Description
0	On success.
1	On any error.

Example

The following example describes how to create a Correlator Deployment Package file with multiple monitor files and inject the CDP file into a running correlator.

1. Create a manifest file containing a list of files to include in the CDP. For this example, the file is named manifest.txt and each line contains the full path name of an EPL file or .jar file:

```
c:\dev\sample\monitor1.mon
c:\dev\sample\monitor2.mon
C:\dev\sample\jmon-app.jar
```

2. To create the CDP file, call the engine_package utility stating the output filename and the manifest file to include in the CDP. (Note, instead of using a manifest file, you can list the files individually in the engine_package arguments.)

```
engine_package.exe -o c:\sample.cdp -m c:\dev\sample\manifest.txt
```

3. To inject the CDP file, call the engine_inject utility with -c (or --cdp). This injects each file that is included in the CDP file into the correlator.

```
engine_inject.exe -c c:\sample.cdp
```

Sample output from the correlator

```
2012-07-11 13:51:33.156 INFO  [3852] - Injected CDP from file
        c:\sample.cdp (b2f097b02791e5dd4ac73cda38e153e9),
        size 313 bytes, decoding and compile time 0.00 seconds
```

Event Correlator Utilities Reference

Sending events to correlators

The engine_send tool sends Apama-format events into an event correlator or IAF adapter. The executable for this tool is located in the bin directory of the Apama installation. Running the tool in the Apama Command Prompt ensures that the environment variables are set correctly.

If the events you want to send are not in Apama format, you must use an adapter that can transform your event format into Apama event format.

Synopsis

To send Apama-format events to an event correlator or IAF adapter:

- On Windows, run `engine_send.exe`.
- On UNIX, run `engine_send`.

To obtain the following usage message, run the command without any options or with the `-h` option:

```
Usage: engine_send [ options ] [ file1 [ file2 ... ] ]
In order to send to a Correlator or IAF.
Where options include:
    -h | --help                This message
    -n | --hostname <host>     Connect to an engine on <host>
    -p | --port <port>         Engine is listening on <port>
    -l | --loop <count>        Loop through input <count> times
    -v | --verbose             Be more verbose
    -u | --utf8                Assume input is in UTF8
    -V | --version             Print program version info
    Use '-' to read from <stdin>
    Loop count < 0 means loop forever
```

Description

The `engine_send` tool sends Apama-format events to an event correlator. In Apama-format event files, you can specify whether to send the events in batches of one or more events or at set time intervals.

The correlator reads events from one or more specified files. Alternatively, you can specify "-"or not specify a filename and the correlator reads events from `stdin` until it receives an end-of-file signal (Ctrl-D on UNIX and Ctrl-Z on Windows).

For details about Apama-format events, see "Event file format" on page 163.

By default, the `engine_send` tool is silent unless an error occurs. To view progress information during `engine_send` execution, specify the `-v` option when you invoke `engine_send`.

You can also use `engine_send` to send events directly to the Integration Adapter Framework (IAF). To do this, specify the port of the IAF, by default this is `16903`.

Options

The `engine_send` tool takes the following command line options:

Option	Description
`-h`	Displays usage information. Optional.
`-n` *host*	Name of the host on which the event correlator to which you want to send events is running. Optional. The default is `localhost`. Note: Non-ASCII characters in host names are not supported.
`-p` *port*	Port on which the event correlator is listening. Optional. The default is `15903`.

Option	Description
`-v`	Requests verbose output during execution. Optional.
`-V`	Displays version information for the `engine_send` tool. Optional.
`-l` *loop*	Number of times to cycle through and send the input events. Optional. Replace *loop* with one of the following values: • `0` — Indicates that you want the `engine_send` tool to iterate through and send the input data once. This is the default. • Any negative integer— Indicates that you want the `engine_send` tool to indefinitely cycle through and send the input events. • Any positive integer — Indicates the number of times to cycle through and send the input events. The `engine_send` tool ignores this option if you specify it and the input is from `stdin`.
`-u`	Indicates that input files are in UTF-8 encoding. This specifies that the `engine_send` utility should not convert the input to any other encoding.

Operands

The `engine_send` tool takes the following operands:

`[` *file1* `[` *file2* `... ] ]`	Specify zero, one, or more files that contain event data. Each file you specify must comply with the event file format described in "Event file format" on page 163. If you do not specify any filenames, the `engine_send` tool takes input from `stdin`.

Exit status

The following exit values are returned:

0	The events were sent successfully.
1	No connection to the event correlator was possible or the connection failed.
2	One or more other errors occurred while sending the events.

Operating notes

To end an indefinite cycle of sending events, press Ctrl-C in the window in which you invoked the `engine_send` tool.

You might want to indefinitely cycle through and send events in the following situations:

- In test environments. For example, you can use `engine_send` to simulate heartbeats. If you then kill the `engine_send` process, you can test your EPL code that detects when heartbeats stop.
- In production environments. For example, you can use the `engine_send` tool to initialize a large data table in the correlator.

Text encoding

By default, the engine_send tool checks the environment variable or global setting that specifies the locale because this indicates the local character set. The engine_send tool then translates EPL text from the local character set to UTF-8. Consequently, it is important to correctly set the machine's locale.

However, some input files might start with a UTF-8 Byte Order Mark. The engine_send tool treats such input files as UTF-8 and does not do any translation. Alternatively, you can specify the -u option when you run the engine_send tool. This forces the tool to treat each input file as UTF-8.

Event Correlator Utilities Reference

Receiving events from correlators

The engine_receive tool lets you connect to a running event correlator and receive events from them. Events received and displayed by the engine_receive tool are in Apama event format. This is identical to the format used to send events to the correlator with the engine_send tool. Consequently, it is possible to reuse the output of the engine_receive tool as input to the engine_send tool. The executable for this tool is located in the bin directory of the Apama installation. Running the tool in the Apama Command Prompt ensures that the environment variables are set correctly.

Synopsis

To receive Apama-format events from an event correlator:

- On Windows, run engine_receive.exe.
- On UNIX, run engine_receive.

To obtain the following usage message, run the command without any options or with the -h option:

```
Usage: engine_receive [ options ]
To receive and log output events from a remote correlator.
Where options include:
 -h | --help               This message
 -n | --hostname <host>    Connect to an engine on <host>
 -p | --port <port>        Engine is listening on <port>
 -c | --channel <channel>  Listen on output channel <channel>
 -f | --filename <file>    Write to <file> instead of to standard out
 -s | --suppressBatch      Do not include BATCH timestamps in the output
    | --suppressbatch
 -z | --zeroAtFirstBatch   Measure BATCH timestamps from when the first
    | --zeroatfirstbatch   batch arrived, instead of from when
                           engine_receive was started
 -r | --reconnect          Automatically (re)connect to the server when available
 -x | --qdisconnect        Disconnect from correlator if we don't keep up
 -v | --verbose            Be more verbose
 -u | --utf8               Write output in UTF8
 -V | --version            Print program version info
 -Xconfig | --configFile <file> Use service configuration file <file>
    Multiple -c options may be given
```

Description

The engine_receive tool receives events from an event correlator and writes them to stdout or to a file that you specify. The correlator output format is the same as that used for event input and is described in "Event file format" on page 163.

You can specify one or more channels on which to listen for events from the correlator. The default is to receive all output events. For information about event channel, see "Introduction to Apama Event Processing Language" in *Developing Apama Applications in EPL* (available if you selected Developer during installation).

To view progress information during `engine_receive` execution, specify the `-v` option.

You can also use `engine_receive` to receive events emitted by the Integration Adapter Framework (IAF) directly. To do this, specify the port of the IAF, by default this is `16903`. Note, however that `engine_receive` must either not specify a channel (i.e., listen to all channels), or specify the `com.apama.input` channel, which echoes back all Apama events sent to the IAF.

Options

The tool `engine_receive.exe` (on Windows) or `engine_receive` (on UNIX) takes a number of command line options. These are:

Option	Description
`-h`	Displays usage information. Optional.
`-n` *`host`*	Name of the host on which the event correlator is running. Optional. The default is `localhost`. Note: Non-ASCII characters in host names are not supported.
`-p` *`port`*	Port on which the event correlator is listening. Optional. The default is `15903`.
`-c` *`channel`*	Named channel on which to listen for output events from the correlator. Optional. The default is to listen for all output events. You can specify the `-c` option multiple times to listen on multiple channels.
`-f` *`file`*	Dumps all received events in the specified file. Optional. The default is to write the events to `stdout`.
`-s`	Omits `BATCH` timestamps from the output events. Optional. The default is to preserve `BATCH` timestamps in events.
`-z`	Records the first received batch of events as being received at `0` milliseconds after the `engine_receive` tool was started. Optional. The default is that the first received batch of events is received at the number of milliseconds since `engine_receive` actually started.
`-r`	Automatically (re)connect to the server when available
`-x`	Disconnect from the correlator if the `engine_receive` utility cannot keep up with the events from the correlator.
`-v`	Requests verbose output during `engine_receive` execution. Optional.
`-u`	Indicates that received event files are in UTF-8 encoding. This specifies that the `engine_receive` utility should not convert the input to any other encoding.
`-V`	Displays version information for the `engine_receive` tool. Optional.
`-Xconfig` *`file`*	Specifies a special configuration file that the correlator uses to initialize its networking. Specify this option only as directed by Apama Technical Support to troubleshoot networking problems. For more

Option	Description
	information, see "Using the Apama Component Extended Configuration File" on page 363.

Exit status

The `engine_receive` tool returns the following exit values:

Status	Description
0	All events were received successfully.
1	No connection to the event correlator was possible or the connection failed.
2	Other error(s) occurred while receiving events.

Text encoding

The `engine_receive` tool translates all events it receives from UTF-8 into the current character locale. It is therefore important that you correctly set the machine's locale. To force the `engine_receive` tool to output events in UTF-8 encoding, specify the `-u` option.

Event Correlator Utilities Reference

Watching correlator runtime status

The `engine_watch` tool lets you monitor the runtime operational status of a running event correlator. The executable for this tool is located in the `bin` directory of the Apama installation. Running the tool in the Apama Command Prompt ensures that the environment variables are set correctly.

Synopsis

To monitor the operation of an event correlator:

- On Windows, run `engine_watch.exe`.
- On UNIX, run `engine_watch`.

To obtain the following usage message, run the command without any options or with the `-h` option:

```
Usage: engine_watch [ options ]
To periodically poll status from a remote correlator.
Where options include:
  -h | --help              This message
  -n | --hostname <host>   Connect to an engine on <host>
  -p | --port <port>       Engine is listening on <port>
  -i | --interval <ms>     Poll every <ms> milliseconds
  -f | --filename <file>   Write to <file> instead of to standard out
  -r | --raw               Raw (i.e. parser friendly) output
  -t | --title             Add header to output (Raw mode only)
  -o | --once              Poll once then exit
  -v | --verbose           Be more verbose
  -V | --version           Print program version info
```

Description

This tool periodically polls an event correlator for status information, writing the returned status data to `stdout`. For additional progress information use the `-v` option.

The `engine_watch` tool returns the following information:

Status message	Meaning
`Uptime (ms)`	The time in milliseconds since this correlator was started. This figure is unaffected if the state of the correlator is restored from a checkpoint file.
`Number of monitors`	The number of EPL monitor definitions injected into the correlator. This figure changes upwards and downwards as monitors are injected, deleted or just expire. A monitor expires when each of its instances dies, or it has no listeners or streams left, or it causes a runtime error.
`Number of sub-monitors`	The number of EPL monitor instances across all contexts in the correlator. In monitors, `spawn` actions create monitor instances. This figure changes upwards and downwards as monitor instances are spawned, killed or just expire.
`Number of contexts`	The number of contexts in the correlator. This includes the main context plus any user-defined contexts.
`Number of Java applications`	The number of Java applications loaded in the correlator. Java applications do not expire, so this value only decreases when you explicitly unload a Java application.
`Number of listeners`	This is the sum of listeners in all contexts. This includes each `on` statement and each stream source template, for example, `all Tick(symbol=”APMA”)` in the following: `stream<Tick> ticks := all Tick(symbol=”APMA”);`
`Number of sub-listeners`	The number of sub-listeners that have been created by listeners across all contexts. A stream source template always has exactly one sub-listener. An `on` statement can have multiple sub-listeners.
`Number of event types`	The total number of event types defined within the correlator. This figure decreases when you delete event types from the correlator.
`Events on input queue`	Total number of events waiting to be processed on all input queues. The main context has its own input queue and any user-defined contexts each have an input queue.
`Events received`	The total number of events ever received by the correlator. A checkpoint preserves this value. If you restore the state of the correlator from a checkpoint file, this number reflects the total number of the events seen by the correlator from which the checkpoint was originally made. Note that if an event is on an input queue, it has been received but not processed.
`Events processed`	The total number of events processed by the correlator in all contexts. This includes external events and events routed to contexts by monitors. An event

Status message	Meaning
	is considered to have been processed when all listeners and streams that were waiting for it have been triggered, or when it has been determined that there are no listeners for the event.
`Events on internal queue`	Total number of routed events waiting to be processed across all contexts. The internal routing queue in each context is a high priority queue for events that you internally routed with the `route` instruction in EPL. The correlator always processes events on the internal queue before any events on the normal input queue.
`Events routed internally`	The total number of events ever routed internally to the internal queues on this correlator. A checkpoint preserves this value. If you restore the state of a correlator from a checkpoint file, this number reflects the total number of the events routed to the internal queues for the correlator from which the checkpoint was originally made.
`Number of consumers`	The number of event consumers registered with the correlator. Event consumers receive events emitted by the correlator.
`Events on output queue`	The number of events waiting on the correlator's output queue to be dispatched to any registered event consumers.
`Output events created`	The total number of output events created by the correlator. A checkpoint preserves this value. If you restore the state of a correlator from a checkpoint file, this number reflects the total number of output events created by the correlator from which the checkpoint was originally made.
`Output events sent`	The total number of output events that the correlator has sent to event consumers. For example, suppose the correlator created 10 output events and sent each event to two consumers. The number of output events sent is 20. A checkpoint preserves this value. If you restore the state of a correlator from a checkpoint file, this number reflects the total number of output events sent by the correlator from which the checkpoint was originally made.
`Event rate over last interval (ev/s)`	The number of events per second currently being processed by the correlator across all contexts. This value is computed with every status refresh and is only an approximation.

Options

The tool `engine_watch.exe` (on Windows) or `engine_watch` (on UNIX) takes a number of command line options. These are:

Option	Description
`-h`	Displays usage information. Optional.
`-n` *`host`*	Name of the host on which the event correlator is running. Optional. The default is `localhost`. Note: Non-ASCII characters in host names are not supported.

Option	Description
`-p` *port*	Port on which the event correlator is listening. Optional. The default is `15903`.
`-i` *ms*	Specifies the poll interval in milliseconds. Optional. The default is `1000`.
`-f` *file*	Writes status output to the named file. Optional. The default is to send status information to `stdout`.
`-r`	Indicates that you want raw output format, which is more suitable for machine parsing. Raw output format consists of a single line for each status message. Each line is a comma-separated list of status numbers. This format can be useful in a test environment. If you do not specify that you want raw output format, the default is a multi-line, human-readable format for each status message.
`-t`	If you also specify the `--raw` option, you can specify the `--title` option so that the output contains headers that make it easy to identify the columns.
`-o`	Outputs one set of status information and then quits. Optional. The default is to indefinitely return status information at the specified poll interval.
`-v`	Displays process names and versions in addition to status information. Optional. The default is to display only status information.
`-V`	Displays version information for the `engine_watch` tool. Optional. The default is that the tool does not output this information.

Exit status

The `engine_watch` tool returns the following exit values:

Status	Description
0	All status requests were processed successfully.
1	No connection to the event correlator was possible or the connection failed.
2	Other error(s) occurred while requesting/processing status.

Event Correlator Utilities Reference

Inspecting correlator state

The `engine_inspect` tool lets you inspect the state of a running event correlator. This means you can review the applications loaded and running on a correlator. The executable for this tool is located in the `bin` directory of the Apama installation. Running the tool in the Apama Command Prompt ensures that the environment variables are set correctly.

Synopsis

To inspect applications on a running event correlator:

- On Windows, run `engine_inspect.exe`.
- On UNIX, run `engine_inspect`.

To obtain the following usage message, run the command without any options or with the `-h` option:

```
Usage: engine_inspect [ options ]
To list names of monitors and events in a correlator.
Where options include:
    -m | --monitors            List monitor types
    -j | --java                List java applications
    -e | --events              List event types
    -t | --timers              List timers
    -x | --contexts            List contexts
    -a | --aggregates          List aggregate functions
    -r | --raw                 Raw (i.e. parser friendly) output
    -h | --help                This message
    -n | --hostname <host>     Connect to an engine on <host>
    -p | --port <port>         Engine is listening on <port>
    -v | --verbose             Be more verbose
    -V | --version             Print program version info
```

Description

The `engine_inspect` tool retrieves state information from a running event correlator and sends it to `stdout`. By default, the tool outputs information on the monitors, Java applications, event types and container types currently injected in an event correlator.

You can filter this list by specifying command-line options. When you specify one or more of the `-m`, `-j`, `-e`, -t, or `-x`, options, the `engine_inspect` tool displays only the information indicated by the option(s) you specify. See the Options table below for a description of these options.

Options

The `engine_inspect` tool takes the following options.:

Option	Description
`-m`	Displays the names of all EPL monitors in the event correlator and the number of sub-monitors each monitor has spawned.
`-j`	Displays the names of all Apama Java applications in the event correlator and the number of event listeners each Apama Java application has created.
`-e`	Displays the names of all event types in the correlator and the number of templates of each type, as defined in listener specifications. This includes each event template in an `on` statement and each stream source template, for example, `stream<A> := all A().` For more information about event types and listeners, see "Introduction to Apama Event Processing Language" in *Developing Apama Applications in EPL* (available if you selected Developer during installation).

Option	Description
`-t`	Displays the current EPL timers active within the system. The four types of timers which may be displayed here are `wait`, `within`, `at`, and `stream`. The `stream` timers are those set up to support the operation of stream network
`-x`	Displays the names of any user-defined contexts, how many monitor instances are running in each context, and how many entries are on the context's input queue.
`-a`	Displays a list of the custom (user-defined) aggregate functions that have been injected. You use aggregate functions in stream queries. Apama built-in aggregate functions are not listed.

`-r`	Indicates that you want raw output, which is more suitable for machine parsing. Raw output provides the name of each entity in the correlator followed by the number of instances associated with that entity. For a monitor, you get the number of its monitor instances. For a Java application, you get the number of its listeners. For an event type, you get the number of its templates. For example:
	`com.apama.samples.stockwatch.StockWatch 1`
	`Tick 1`

`-h`	Displays usage information.
`-n` *`host`*	Name of the host on which the event correlator is running. The default is `localhost`. Note: Non-ASCII characters in host names are not supported.
`-p` *`port`*	Port on which the event correlator is listening. The default is `15903`.
`-v`	Displays process names and versions in addition to application information. Optional. The default is to display only application information.
`-V`	Displays version information for the `engine_inspect` tool.

Exit status

The `engine_inspect` tool returns the following values:

Status	Description
0	All status requests were processed successfully.
1	No connection to the event correlator was possible or the connection failed.
2	Other error(s) occurred while requesting/processing status.

Event Correlator Utilities Reference

Shutting down and managing components

All Apama components (the correlator, the IAF, and the Sentinel Agent) implement an interface with which they can be asked to shut themselves down, provide their process ID, and respond to communication checks. Running the tool in the Apama Command Prompt ensures that the environment variables are set correctly.

For historical reasons, there are three commands that all do the same thing. You can enter any of these commands to manage any component:

```
engine_management
component_management
iaf_management
```

However, the recommendation is to always use `engine_management`. The only differences in behavior among these commands is as follows:

- `engine_management` and `component_management` default to the local correlator port (`15903`).
- `iaf_management` defaults to the default IAF port (`16903`)

Synopsis

To use the event correlator's management tool:

- On Windows, run `engine_management.exe`.
- On UNIX, run `engine_management`.

When you specify the `-h` command line option, the tool displays the following usage information:

```
Usage: engine_management [ options ]
Where options include:
 -V | --version             Print program version info
 -h | --help                Display this message
 -v | --verbose             Be more verbose
 -n | --hostname <host>     Connect to a component on <host>
 -p | --port <port>         Component is listening on <port>
 -w | --wait                Wait forever for component to start
 -W | --waitFor <num>       Wait <num> seconds for component to start
 -N | --getname             Get the name of the component
 -T | --gettype             Get the type of the component
 -Y | --getphysical         Get the physical ID of the component
 -L | --getlogical          Get the logical ID of the component
 -O | --getloglevel         Get the log level of the component
 -C | --getversion          Get the version of the component
 -R | --getproduct          Get the product version of the component
 -B | --getbuild            Get the build number of the component
 -F | --getplatform         Get the build platform of the component
 -P | --getpid              Get the process ID of the component
 -H | --gethostname         Get the hostname of the component
 -U | --getusername         Get the username of the component
 -D | --getdirectory        Get the working (current) directory of the component
 -E | --getport             Get the port of the component
 -c | --getconnections      Get all the connections to the component
 -a | --getall              Get all of the above values
 -xs| --disconnectsender <id> <reason> Disconnect sender with physical id <id>
 -xr| --disconnectreceiver <id> <reason>Disconnect receiver with physical id <id>
 -I | --getinfo <category>  Get component-specific info for <category>
                            Use empty string to get all available categories
                            Multiple -I options may be specified
 -d | --deepping            Deep-ping the component
 -l | --setloglevel <level> Set logging verbosity to <level>. Available levels
                            are TRACE, DEBUG, INFO, WARN, ERROR, FATAL, CRIT and OFF
 -r | --dorequest <req>     Send component-specific request <req>
 -s | --shutdown <why>      Shutdown the component with reason <why>
```

Description

Use the `engine_management` tool to connect to a running component. Once connected, the tool can shut down the component or return information about the component. The `engine_management` tool can connect to any of the following types of components. The `engine_management` tool sends output to `stdout`.

- Event correlator
- Adapter
- Sentinel Agent process

Options

When you run the `engine_management` tool, you can specify any of the options described in the following table. By default, the `engine_management` tool connects to a local correlator that is listening on port `15903` (the correlator default port). To obtain all information for a particular component, specify the `-a` option. All options are optional:

Option	Description
`-V`	Displays version information for the `engine_management` tool.
`-h`	Display usage information.
`-v`	Displays information in a more verbose manner. For example, when you specify the `-v` option, the `engine_management` tool displays status messages that indicate that it is trying to connect to the component, has connected to the component, is disconnecting, is disconnected, and so on. If you have having trouble obtaining the information you want, specify the `-v` option to help determine where the problem is.
`-n` *<host>*	Name of the host on which the component is running (default is `localhost`).Non-ASCII characters are not allowed in host names.
`-p` *<port>*	Port on which the component you want to connect to is listening The default is `15903`.
`-w`	Instructs the `engine_management` tool to wait for the component to start and be in a state that is ready to receive EPL files. This option is similar to the `-W` option, except that this option (the `-w` option) instructs the tool to wait forever. The `-W` option lets you specify how many seconds to wait. See the information for the `-W` option for an example.
-W<*num*>	Instructs the `engine_management` tool to wait *num* seconds for the component to start and be in a state that is ready to receive EPL files. If the component is not ready to receive EPL files before the specified number of seconds has elapsed, the `engine_management` tool terminates with an exit code of `1`. This option is most useful in scripts, when the component you want to operate on has not yet started. For example, suppose a script specifies the following commands: `correlator.exe` *`options`*`engine_inject` *`some_EPL_files`* It can sometimes take a few seconds for a component to start, and this number of seconds is not always exactly predictable. If the `engine_inject` tool runs before the correlator is ready to receive EPL files, the `engine_inject` tool fails. To avoid this for a local correlator that is listening on the default port, insert the following command between these commands: `engine_management -W 10` This lets the `engine_management` tool wait for up to 10 seconds for the correlator's management interface to be available. To set an appropriate wait time for your application, monitor your application's performance and adjust as needed.

Option	Description
-N	Displays the name of the component. For example, when you start a correlator, you can give it a name with the -N option. This is the name that the engine_management tool returns. If you do not assign a name to a correlator when you start it, the default name is correlator.
-T	Displays the type of the component that the engine_management tool connects to. The returned value is one of the following: correlator, iaf, sentinel_agent. If you see that a port is in use, you can specify this option to determine the type of component that is using that port.
-Y	Displays the physical ID of the component. This can be useful if you are looking at status log information that identifies components by their physical IDs.
-L	Displays the logical ID of the component. This can be useful if you are looking at status log information that identifies components by their logical IDs.
-O	Displays the log level of the component. The returned value is one of the following: TRACE, DEBUG, INFO, WARN, ERROR, CRIT, FATAL, or OFF.
-C	Displays the version of the component. For example, when the tool connects to a correlator, it displays the version of the correlator software that is running.
-R	Displays the product version of the component. For example, when the tool connects to a correlator, it displays the version of the UNIX software that is running.
-B	Displays the build number of the component. This information is helpful if you need technical support. It indicates the exact software contained by the component you connected to
-F	Displays the build platform of the component. This information is helpful if you need technical support. It indicates the set of libraries required by the component you connected to
-P	Displays the process ID of the correlator you are connecting to. This can be useful if you are looking at log information that identifies components by their process ID.
-H	Displays the host name of the component. When debugging connectivity issues, this option is helpful for obtaining the host name of a component that is running behind a proxy or on a multihomed system.
-U	Displays the user name of the component. On a multiuser machine, this is useful for determining who owns a component.
-D	Displays the working (current) directory of the component. This can be helpful if a plug-in writes a file in a component’s working directory.
-E	Displays the port of the component.
-c	This option is for use by technical support. It displays all the connections to the component
-a	Displays all information for the component.

Option	Description
-xs	Disconnects the sender that has the physical ID you specify. If you specify a reason, the `engine_management` tool sends the reason to the correlator. The correlator then logs the message, sends the reason to the sender, and disconnects the sender.
-xr	Disconnects the receiver that has the physical ID you specify. If you specify a reason, the `engine_management` tool sends the reason to the correlator. The correlator then logs the message, sends the reason to the receiver, and disconnects the receiver.
`-I` *<category>*	This option is for use by technical support. It displays component-specific information for the specified category.
`-d`	Ping the component. This confirms that the component process is running and acknowledging communications.
`-l` *<level>*	Sets the amount of information that the component logs. In order of decreasing verbosity, you can specify `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL`, `CRIT`, or `OFF`.
`-r`	This option sends a component-specific request. For example:engine_management -r "profiling frequency" This returns the profiling frequency in Hertz. The following requests are available: `applicationEventLogging` — Sends detailed application information to the correlator log file. See "Viewing garbage collection information" on page 141. `flushAllQueues` — Sends a request into the correlator that waits until every event/injection sent or enqueued to a context before the `flushAllQueues` request started has been processed, and every event emitted as a result of those events has been acknowledged. This may block if a slow receiver is connected to the correlator. Events enqueued to a context after the request has started may or may not be processed — thus if you want to see the results of one context enqueueing to a second, which enqueues to a third, you should execute `engine_management -r "flushAllQueues"` three times, to ensure it has been processed by each context. This does not change the behavior of the correlator (the correlator will always flush all queues as soon as it is able to), it just waits for events currently on input queues to complete. `profiling` — Lets you profile Apama EPL applications. See "Using the profiler command-line interface" on page 142. `rotateReplayLog` — For details on this request, see "Rotating the replay log" on page 159. `setApplicationLogFile` — A set of commands lets you set, get, and unset the log files for packages and monitors. See "Setting logging attributes for packages, monitors and events" on page 145. `setApplicationLogLevel` — A set of commands lets you set, get, and unset logging levels for packages and monitors. See "Setting logging attributes for packages, monitors and events" on page 145.

Option	Description
	`verbosegc` — Enables logging of garbage collection events. See "Viewing garbage collection information" on page 141. Certain other requests for this option are available for use by Apama technical support.
`-s` *<why>*	Instructs the component to shut down and specifies a message that indicates the reason for termination. The component inserts the string you specify in its status log file with a `CRIT` flag, and then shuts down.

Exit status

The following exit values are returned:

Status	Description
0	All status requests were processed successfully.
1	Indicates one of the following: • No connection to the specified component was possible. • The connection failed. • You specified the `waitFor` option and the specified time elapsed without the component starting.
2	One or more errors occurred while requesting/processing status.
3	Deep ping failed.

Event Correlator Utilities Reference

Viewing garbage collection information

A handy way to view garbage collection information is to execute the following command:

```
engine_management -r "verbosegc on"
```

This command enables logging of garbage collection events, and is particularly useful in production environments. The additional garbage collection information goes to the correlator log. To disable logging of garbage collection information, execute the following:

```
engine_management -r "verbosegc off"
```

These commands provide an alternative to the following command, which provides a great deal of detailed output in addition to garbage collection information. Again, this output goes to the correlator log.

```
engine_management -r "applicationEventLogging on"
```

To turn this off:

```
engine_management -r "applicationEventLogging off"
```

Shutting down and managing components

Using the profiler command-line interface

You can profile applications written with EPL in Apama Studio. Data collected in the profiler allows you to identify possible bottlenecks in an EPL application. When testing an application, or after you deploy an application, you might find it handy to write a script that includes obtaining profile information. Or, you might want to obtain profile information without the overhead of Apama Studio. In these situations, you can use the command-line interface to the profiler, which is described here.

The command-line interface to the profiler consists of specifying the `-r` | `--dorequest` option with the `engine_management` utility:

```
engine_management -r "profiling on"
engine_management -r "profiling off"
engine_management -r "profiling get"
engine_management -r "profiling gettotal"
engine_management -r "profiling reset"
engine_management -r "profiling frequency"
```

`on`	Starts to capture the state of all contexts in the correlator.
`off`	Stops capturing profile data.
`get`	Returns the samples collected since the correlator was started or since the profiler was reset. Returned data is in CSV (comma separated values) format. A sample is the state of the correlator at the moment the profiler collects data.
gettotal	Returns totals for all contexts.
`reset`	Clears profiling samples collected.
`frequency`	Returns the profiling frequency in Hertz.

If a context is executing, it is typically in the EPL interpreter. However, it might also be doing something such as matching events or collecting garbage. For EPL execution, there is a call stack for each context. For the purposes of the profiler, there is one entry at the top for the monitor name, then comes the listener/`onload` action, and then any actions that is calling, and so on. The only action that the correlator is actually executing is at the bottom of the stack.

A context can be in one or two of the following states:

- CPU — the correlator is executing code in this context.
- Runnable — the correlator has work to do in this context but it has been rescheduled because the correlator is executing code in another context.
- Idle —the correlator has no work to do in this context.
- Non-Idle — the correlator has work to do in this context. When a context is in this state, it is also in one other state, either CPU, Plugin, Blocked, or Runnable).
- Plugin — the correlator is executing a plugin in this context.
- Blocked — the correlator cannot make progress in this context. It is blocked because of a full queue. The full queue might be the correlator output queue (the context is trying to emit an event) or another context's input queue.

When the profiler takes a sample it examines every context in the correlator. Every entry in each context's call stack results in addition or modification of a line in the profiler output. The Cumulative column is incremented for all samples, and one or more of the other columns is incremented for the lowest (deepest) call stack element according to what states the context is in.

When the correlator is not executing EPL code, there is only one element in the stack, for example, when the correlator is processing an event.

The profiler's resolution is to a EPL action. That is, the profiler does not distinguish between lines within an action. The line number in the output is the first line of the action that generates code. For example, variable declarations without initializers, and comments do not generate code, while statements, and declarations with initializers, do generate code. The profiler treats the body of a listener (the code the correlator executes when the listener fires) as an action with the name `::listenerAction::`.

If you want to profile parts of a single large action, you need to split the action into multiple actions in order to determine where time is spent. Remember that action calls have some cost, so that could skew the results.

The `"profiling get"` or `"profiling gettotal"` request returns samples to `stdout` as lines of comma separated values. Output is sorted by context and then by CPU time. For example:

```
Context Id,Context Name,Location,Filename and line number,Cumulative time,CPU time,
Empty,Non-Idle,Idle,Runnable,Plugin,Blocked,total ticks:573
3,3,processor:processor.::listenerAction::,/users/ukcam/cr/dev/4.3.0.0/apama-test/
system/correlator/corba/testcases/correctness/Corr_Corba_cor_543/Input/
create-state.mon:50,556,293,0,556,0,0,263,0
```

In the previous output, nearly all of the time of this context (3) is spent in the listener that starts on line 50 of `create-state.mon`. The time is spread between executing EPL code (293 samples) and executing a plugin (263 samples). Each context spent similar amounts of time executing EPL and executing plug-ins but in different listeners (notice the different line numbers).

Here is more sample output:

```
3,3,Idle,,1,1,0,1,14,0,0,0
3,3,Only just started profiling,,0,0,0,0,2,0,0,0
3,3,Monitor:processor,,556,0,0,0,0,0,0,0
2,2,processor:processor.::listenerAction::,/users/ukcam/cr/dev/4.3.0.0/apama-test/
system/correlator/corba/testcases/correctness/Corr_Corba_cor_543/Input/create-state.mon:
34,556,261,0,556,0,0,295,0
2,2,Idle,,1,1,0,1,14,0,0,0
2,2,Only just started profiling,,0,0,0,0,2,0,0,0
2,2,Monitor:processor,,556,0,0,0,0,0,0,0
4,4,processor:processor.::listenerAction::,/users/ukcam/cr/dev/4.3.0.0/apama-test/
system/correlator/corba/testcases/correctness/Corr_Corba_cor_543/Input/create-state.mon:
65,556,296,0,556,0,0,260,0
4,4,Idle,,1,1,0,1,14,0,0,0
4,4,Only just started profiling,,0,0,0,0,2,0,0,0
4,4,Monitor:processor,,556,0,0,0,0,0,0,0
1,main,processor:processor.::listenerAction::,/users/ukcam/cr/dev/4.3.0.0/apama-test/
system/correlator/corba/testcases/correctness/Corr_Corba_cor_543/Input/create-state.mon:
18,556,283,0,556,0,0,273,0
1,main,Idle,,1,1,0,1,14,0,0,0
1,main,Only just started profiling,,0,0,0,0,2,0,0,0
1,main,Monitor:processor,,556,0,0,0,0,0,0,0
```

This output is intended to be imported to a spreadsheet, such as Excel. If you do that, then the values in one sample (one row) provide the following information in the following order:

Column Content	Description
Context ID	ID of the context. A context ID is not present in data returned by `-r "profiler gettotal"`.
Context Name	Name of the context. A context name is not present in data returned by `-r "profiler gettotal"`.
Location	What the correlator is doing or where the correlator is executing code at the moment the sample was collected. The value is one of the following: ● `Monitor:`*monitor_name* — The top-level entry for the monitor. ● *monitor_name.code_owner*`.`*action_name* — For example, if monitor `monny` calls an action `act` on event `pkg.evie`, this location would be `monny.pkg.evie.act`. If a listener has been triggered, the action name is always `::listenerAction::`. ● *monitor_name*`.;GC` — Garbage collection. ● `Event:`*event_name* — Event matching or chastenment of an event of that type ● `Idle` — Correlator has no work to do. ● There are other possible values that you might rarely see. They are self explanatory.
Filename and line number	If the correlator is executing EPL code, indicates the filename and line number of the beginning of the action that is executing.
Cumulative time	Cumulative time indicates time spent in this location or in something that this location was calling (directly or indirectly). CPU time shows time spent in this location, not the actions it called.I
CPU time	Number of samples in which the correlator is executing the location/action and is not in a plugin (see Plugin later in this table). CPU time is a subset of Cumulative time. It does not include time spent in the location(s) called by this location.
Empty	Number of samples in which the context was empty.An empty context should happen very rarely. A context might be empty if there is a race between getting the location and the state.
Non-idle	Number of samples in which the context was at this row's location and not idle. Each sample in this count is also in the count for CPU time, Runnable, Plugin, or Blocked.
Idle	Number of samples in which the context was idle. This should correspond to a location of Idle or Only just started profiling, which means it is an unknown state. As with other cumulative counters, races can result in misleading results. For example, `Idle` in an action, but those are best ignored and should be small.

Column Content	Description
Runnable	Number of samples in which the location was the lowest point on the call stack and the context was runnable. Runnable means it could have made progress, but the scheduler determined that the correlator should run something else instead. When all rows contain `0` for this entry it means that the correlator never (or very rarely) had to re-schedule one context to run another context. A non-zero value means this location was running for a long time, and it was suspended so that other contexts could run
Plugin	Number of samples in which the location is executing a correlator plug-in.
Blocked	Number of samples in which the context was unable to make progress. For example, it was trying to emit an event but the correlator output queue was full, or it was trying to enqueue an event to a particular context but that context's input queue was full.

Shutting down and managing components

Setting logging attributes for packages, monitors and events

In EPL code, you can specify `log` statements as a development or debug tool. By default, `log` statements that you specify in EPL send information to the correlator log file. If a log file was not specified when the correlator was started, and you have not executed the `engine_management` utility to associate a log file with the correlator, `log` statements send output to `stdout`.

In place of this default behavior, you can specify different log files for individual packages, monitors and events. This can be helpful during development. For example, you can specify a separate log file for a package or monitor you are implementing, and direct log output from only your development code to that file.

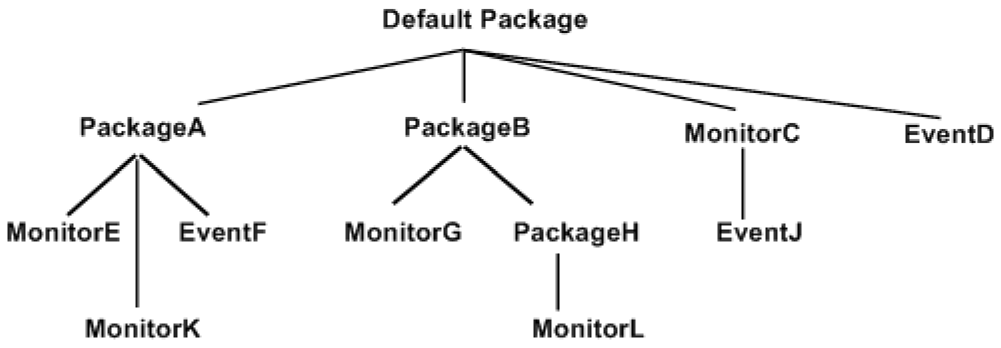
Also, you can specify a particular log level for a package, monitor, or event. The settings of log files and log levels are independent of each other. That is, you can set only a log level for a particular package, monitor or event, or you can set only a log level for a particular element. The following topics provide information for managing individual log files and log levels.

- "Tree structure of packages, monitors, and events" on page 145
- "Managing application log levels" on page 146
- "Managing application log files" on page 147

Tree structure of packages, monitors, and events

Packages, monitors and events form a tree as illustrated in the figure below. For each node in the tree, you can specify a log file and/or a log level. Nodes for which you do not specify log settings inherit log settings from their parent node.

Figure 5. Tree structure of packages, monitors, and events



The root of the tree is the default package, which contains code that does not explicitly specify a package with the `package` statement. Specified packages are intermediate nodes. Packages can nest inside each other. Monitors and events in specified packages are leaf nodes. If you specify an event type in a monitor, that event is a leaf node and its containing monitor is an intermediate node.

For example, suppose you specify `packageA.log` as the log file for `packageA`. The `packageA.log` file receives output from `log` statements in `MonitorE` and `MonitorK`. If `EventF` contains any `action` members that specify `log` statements, output would go to the `packageA.log` file.

Now suppose that you set `ERROR` as the log level for the default package and you set `INFO` as the log level for `PackageB`. For `log` statements in `MonitorG`, `PackageH`, and `MonitorL`, the correlator compares the log statement's log level with `INFO`. For `log` statements in the rest of the tree, the correlator compares the `log` statement's log level with `ERROR`. For details, see the table in "Managing application log levels" on page 146.

Managing application log levels

To set the log level for a package, monitor or event, invoke the `engine_management` utility as follows:

```
engine_management -r "setApplicationLogLevel logLevel [node]"
```

logLevel	Specify `OFF`, `CRIT`, `FATAL`, `ERROR`, `WARN`, `INFO`, `DEBUG`, or `TRACE`.
node	Optionally, specify the name of a package, monitor or event. If you do not specify a node name, the utility sets the log level for the default package.

To obtain the log level for a particular node, invoke the utility as follows:

```
engine_management -r "getApplicationLogLevel [node]"
```

If you do not specify a node, the utility returns the log level for the default package. To remove the log level for a node, so that it takes on the log level of its parent node, invoke the utility as follows. Again, if you do not specify a node, you remove the log level for the default package. The default package then takes on the log level in effect for the correlator. The default correlator log level is `INFO`.

```
engine_management -r "unsetApplicationLogLevel [node]"
```

To manage the log level for an event that you define in a monitor, see "Managing event logging attributes" on page 147.

After the correlator identifies the applicable log level, the log level itself determines whether the correlator sends the `log` statement output to the appropriate log file. The following table indicates which log level identifiers cause the correlator to send the log statement to the appropriate log file.

Log Level in Effect	Log Statements With These Identifiers Go to the Appropriate Log File	Log Statements With These Identifiers are Ignored
OFF	None	CRIT, FATAL, ERROR, WARN, INFO, DEBUG, TRACE
CRIT	CRIT	FATAL, ERROR, WARN, INFO, DEBUG, TRACE
FATAL	CRIT, FATAL	ERROR, WARN, INFO, DEBUG, TRACE
ERROR	CRIT, FATAL, ERROR	WARN, INFO, DEBUG, TRACE
WARN	CRIT, FATAL, ERROR, WARN	INFO, DEBUG, TRACE
INFO	CRIT, FATAL, ERROR, WARN, INFO	DEBUG, TRACE
DEBUG	CRIT, FATAL, ERROR, WARN, INFO, DEBUG	TRACE
TRACE	CRIT, FATAL, ERROR, WARN, INFO, DEBUG, TRACE	None

See also "Log levels determine results of log-statements" in the section "Defining what happens when matching events are found" in *Developing Apama Applications in EPL*.

Managing application log files

To specify a log file for a package, monitor or event, invoke the `engine_management` utility as follows:

```
engine_management -r "setApplicationLogFile logFile [node]"
```

logFile	Specify the path of the log file. You cannot specify a space in a log file name.
node	Optionally, specify the name of a package, monitor or event. If you do not specify a node name, the utility associates the log file with the default package.

To obtain the path of the log file for a particular node, invoke the utility as follows:

```
engine_management -r "getApplicationLogFile [node]"
```

If you do not specify a node, the utility returns the log file for the default package. To disassociate a log file from its node, so that the node uses the log file of its parent node, invoke the utility as follows. Again, if you do not specify a node, you disassociate the log file from the default package. The correlator log file is then in effect for the default package. If a log file has not been specified for the correlator, the default is `stdout`.

```
engine_management -r "unsetApplicationLogFile [node]"
```

Managing event logging attributes

If you specify an event type in a monitor, that event does not inherit the logging configuration from the enclosing monitor. It is expected that this will change in a future release. To explicitly set logging attributes for an event type defined in a monitor, invoke the `engine_management` utility and specify an unqualified event type name. Do not specify an enclosing scope, such as `com.apamax.myMonitor.NestedEventType`. For example:

```
engine_management -r "setApplicationLogFile foo.log NestedEventType"
engine_management -r "setApplicationLogLevel DEBUG NestedEventType"
```

Shutting down and managing components

Using the command-line debugger

The `engine_debug` tool lets you control execution of EPL code in the correlator and inspect correlator state. This tool is a correlator client that runs a single command from the command line. It is not an interactive command-line debugger.

In general, this tool is expected to be most useful when you are ready to deploy your application or after deployment. During development, the interactive debugger in Apama Studio will probably be most useful to you.

Before you run the `engine_debug` tool, specify the `-g` option when you start the correlator. Specification of this option disables some correlator optimizations. If you run the `engine_debug` utility and you did not specify the `-g` option when you started the correlator, the optimizations hinder the debugging process. For example, the correlator might simultaneously execute multiple statements over multiple lines even if you are using debugger commands to step through the program line by line.

Information about the command-line debugger is organized as follows:

- "Synopsis" on page 148
- "Debug commands" on page 149
- "Exit status" on page 152
- "Obtaining online help for the command-line debugger" on page 153
- "Enabling and disabling debugging in the correlator" on page 153
- "Working with breakpoints using the command-line debugger" on page 153
- "Controlling execution with the command-line debugger" on page 154
- "Command shortcuts for the command-line debugger" on page 155
- "Examining the stack with the command-line debugger" on page 156
- "Displaying variables with the command-line debugger" on page 157

Synopsis

To debug applications on a running event correlator:

- On Windows, run `engine_debug.exe`.
- On UNIX, run `engine_debug`.

To obtain a usage message, run the command with the `help` option.

Description

Debugging a running correlator has some effect on the other programs that connect to that correlator. While you pause a correlator, the expected behavior of connected components is as follows:

- Sending events to the correlator continues to put events on the input queue of each public context. However, since the input queues are not being drained, if an input queue fills up, this will block senders, including the `engine_send` tool and adapters.
- The correlator sends out any events on its output queue. When the output queue is empty, receivers no longer receive events; no contexts are sending events.
- Other inspections of the correlator proceed as normal. For example, `engine_watch`, `engine_management`, and profiling data.
- You can shut down the correlator.
- You can inject monitors while the correlator is stopped. They will not run any of the `onload()` or similar code until the correlator resumes, but the inject call should succeed.
- Java applications continue to run completely independently of whether the correlator is stopped.
- All other requests block until the correlator resumes processing. This includes dumping correlator state, loading, and changing debug or profiling state.

The `engine_debug` tool is stateless. Consequently, during debugging, you can have multiple concurrent connections to the same correlator.

Debug commands

The ordering of arguments to `engine_debug` commands works as follows:

- All arguments before the first command apply to all commands in that command line. This is useful for setting the host and port if you are not using the local defaults.
- All arguments following a command apply to only that command and they override any applicable arguments specified before the first command.
- The arguments to a particular command can be in any order
- When there are multiple commands in a line, the debugger executes them in the order in which they are specified. Execution continues until either all complete, or one fails, which prevents execution of any subsequent commands.

The `engine_debug` tool takes the following commands as options:

Abbv.	Command	Description
	`help [`*`command`*`]`	Displays a usage message. To obtain help for a particular `engine_debug` command, specify that command.
`p`	`status`	Displays the current debugger state, and position if stopped.
`si`	`stepinto`	Steps into an action.
`sot`	`stepout`	Steps out of an action.
`sov`	`stepover`	Steps over an instruction.

Abbv.	Command	Description
r	run	Begins processing instructions.
b	stop	Stops processing instructions.
w [-to*int*]	wait [--timeout timeout]	Waits for the correlator to stop processing instructions. Specify an integer that indicates the number of seconds to wait. The debugger waits forever if you do not specify a timeout. See "The wait command" on page 155 for more information.
s	stack [--context contextid] | [--frame frameid]	Displays current stack information for all contexts. The output includes the frame ID associated with each variable. To display stack information for only a particular context, specify the --context argument. To display stack information for only a particular frame, specify the --frame argument.
i	inspect --instance monitorinstance | --instance monitorinstance --frame frameid | --instance monitorinstance --variable variablename | --instance monitorinstance --frame frameid --variable variablename | --frame frameid | --frame frameid --variable variablename	Displays the value of one or more variables. Specify a monitor instance and/or a frame ID and/or a variable name to display a list of variables in that monitor or in a particular monitor frame, or to display the value of a particular variable. Obtain monitor instance IDs from engine_inspect output or correlator log statements. Obtain frame IDs from engine_inspect stack output.
c	context [--context contextid]	Displays information about all contexts in the correlator or about only the context you specify. Information displayed includes context name, context ID, monitor instances in the context, and monitor instance IDs.

Abbv.	Command	Description
e	enable	Enables debugging. You must run this in order to do any debugging.
d	disable	Disables debugging. You must run this to disable debugging. If you do not disable debugging, the correlator runs more slowly and continues to stop when it hits breakpoints.
boe	breakonerror enable	Causes the debugger to pause if it encounters an error.
boeoff	breakonerror disable	Causes the debugger to continue processing if it encounters an error.
ba	breakpoint add [--breakonce] --file filename --line linenumber | [--breakonce] --owner ownername --action actionname --line linenumber	Adds a breakpoint at the beginning of the specified line. If you do not specify --breakonce, the correlator always pauses at this point when debugging is enabled. You must specify the line number where you want the breakpoint. As usual, this is the absolute offset from the beginning of the file. You must specify either the name of the file that contains the breakpoint or the owner and action name that contains the breakpoint. When the owner is a monitor instance, specify *package_name.monitor_name* or just *monitor_name* if there is no package.
bd	breakpoint delete --file filename --line linenumber | --owner ownername --action actionname --line linenumber | --breakpoint breakpointid	Removes a breakpoint. Specify one of the following: • File name and line number • Owner name, action name and line number. When the owner is a monitor instance, specify *package_name.monitor_name* or

Abbv.	Command	Description
		just *monitor_name* if there is no package. - Breakpoint ID. You can obtain a breakpoint ID by executing the `breakpoint list` command.
`bls`	`breakpoint list`	For each breakpoint in the correlator, this displays the following: - Breakpoint ID - Name of file that contains the breakpoint - Name of the action that contains the breakpoint - Name of the owner of the breakpoint - Number of the line that the breakpoint is on. The breakpoint owner is the name of the monitor that contains the breakpoint or the name of the event type definition that contains the breakpoint. If the breakpoint is in an event type definition, the definition must specify an action and processing must create a closure between an event instance and an action call. For information about closures, see "Using action type variables" in *Developing Apama Applications in EPL* (available if you selected Developer during installation).

Exit status

The `engine_debug` tool returns the following values:

Status	Description
`0`	Success — All requests were processed successfully.

Status	Description
1	Failure — The correlator could not parse the command line, or an exception occurred, such as losing a connection or trying to use a non-existent ID.

Event Correlator Utilities Reference

Obtaining online help for the command-line debugger

The command-line debugger provides online help. To obtain general information, enter the following:

engine_debug help

To get help for a particular command, specify that command after the `help` keyword. For example:

engine_debug help status

```
status: Displays the current debugger state, and position if stopped
```

engine_debug help breakpoint add

```
breakpoint add [--breakonce] --line linenumber [--file filename |
--owner ownername --action actionname]:
Add a breakpoint at the specified location
```

Using the command-line debugger

Enabling and disabling debugging in the correlator

To use the debugger, you must enable debugging in the correlator. To enable debugging locally on the default port, enter the following:

```
engine_debug enable
```

When you are done debugging, you should disable debugging in the correlator. If you do not, the correlator runs more slowly and continues to pause when it hits a breakpoint. To disable debugging in the local correlator on the default port, enter the following:

```
engine_debug disable
```

You can also use the debugger in a remote correlator by specifying the host name and the port number. For example:

```
engine_debug enable --host foo.bar.com --port 1234
engine_debug disable --host foo.bar.com --port 1234
```

Using the command-line debugger

Working with breakpoints using the command-line debugger

You can use the command-line debugger for:

- "Adding breakpoints" on page 154
- "Listing breakpoints" on page 154
- "Removing breakpoints" on page 154

Adding breakpoints

There are two ways to add a breakpoint. If you know the EPL file name and the line number you can enter something like the following:

```
engine_debug breakpoint add --file filename.mon --line 27
```

When you specify a file name you must specify the exact path you specified when you injected the monitor. For example, suppose you ran the following:

```
engine_inject foo.mon
```

You can then specify `foo.mon` for the file name. Now suppose you ran this:

```
engine_inject c:\foo\bar\baz.mon
```

You must then specify `c:\foo\bar\baz.mon` for the file name.

If you prefer to use the monitor and action name, along with the line number, enter something like this:

```
engine_debug breakpoint add --monitor package.monitor --action actionName --line 27
```

The debugger output indicates the line number where it added the breakpoint. In some cases, the debugger does not set the breakpoint on the line you specified. For example, when a statement runs over multiple lines.

Listing breakpoints

To obtain a list of the breakpoints currently set in the correlator, enter the following:

```
engine_debug breakpoint list
```

Removing breakpoints

To remove a breakpoint by specifying the file name and the line number, enter something like the following:

```
engine_debug breakpoint delete --file filename.mon --line 27
```

To use the monitor name to remove a breakpoint, enter something like this:

```
engine_debug breakpoint delete --monitor package.monitor --action actionName --line 27
```

To delete a breakpoint by using the breakpoint ID that appears in the breakpoint list returned by the debugger, enter something like this:

```
engine_debug breakpoint delete --breakpoint 1
```

Controlling execution with the command-line debugger

When the correlator stops at a breakpoint, you can use the debugger to step over the next line:

```
engine_debug stepover
```

However, you most likely want to step over the line, confirm that the correlator stopped, and learn about the current state of the debugger. You can do this by entering multiple commands in one line. For example:

```
engine_debug stepover wait --timeout 10 status
```

This is the equivalent of the following three commands:

- `engine_debug stepover` — Causes the debugger to step over one line of EPL.
- `engine_debug wait --timeout 10` — Causes the debugger to pause until either a breakpoint is hit, or ten seconds pass.
- `engine_debug status` — Displays the debugger's current status.

Following are more examples of entering multiple commands in one line.

```
engine_debug stepinto wait --timeout 10 status
engine_debug stepout wait --timeout 10 status
```

To instruct the correlator to continue executing EPL code, run the following command:

```
engine_debug run
```

You use the `engine_debug run` command regardless of how the correlator was stopped — a breakpoint was reached, a step operation, a `wait` command.

To stop the correlator, enter the following command:

```
engine_debug stop
```

Using the command-line debugger

The wait command

The `wait` command connects to the correlator to determine if the correlator has suspended processing. If the correlator is in suspend mode, the `wait` command returns immediately and debugging continues. If the correlator is not in suspend mode, the `wait` command remains connected to the correlator. The `wait` command returns when something else suspends the correlator or when the timeout is reached. Operations that can suspend the correlator include reaching a breakpoint, stepping into or over a line, or some other client explicitly stopping the correlator. If the `wait` command reaches the timeout, it suspends the correlator before it returns.

Stepping can take a variable amount of time. For example, suppose the debugger stops at the end of a listener and you execute a step command. The debugger is now outside the flow of execution until another event comes in. The time that the debugger has to wait for the step to finish is dependent upon when the next matching event arrives.

Controlling execution with the command-line debugger

Command shortcuts for the command-line debugger

Putting multiple commands in the same command line can get verbose. For example, suppose you want to step out of an action on a remote machine. You would need to enter something like this:

```
engine_debug stepout --host foo.bar.com --port 1234 wait --timeout 10
   --host foo.bar.com --port 1234 status --host foo.bar.com --port 1234
```

The command-line debugger provides easier ways to invoke this.

- Any arguments that you specify before the first debugging command apply to the entire command line.
- All individual commands and their arguments have abbreviations.

For example, the following command does the same thing as the previous verbose command:

```
engine_debug -h foo.bar.com -p 1234 sot w -to 10 p
```

The following table lists the abbreviations you can use for command arguments. For abbreviations of commands, see "Debug commands" on page 149

Command	Abbreviation
`--action`	`-a`
`--breakonce`	`-bo`
`--breakpoint`	`-bp`
`--context`	`-c`
`--file`	`-f`
`--frame`	`-fm`
`--host`	`-n`
`--instance`	`-mt`
`--line`	`-l`
`--owner`	`-o`
`--port`	`-p`
`--raw`	`-R`
`--timeout`	`-to`
`--utf8`	`-u`
`--variable`	`-v`
`--verbose`	`-V`

Using the command-line debugger

Examining the stack with the command-line debugger

When the correlator stops at a breakpoint you can display the stack with the following command:

```
engine_debug stack
```

The results of this command show the number of the frame that contains each variable. In the following example, the frame number is the number before the right parenthesis:

```
0 )
        C:/dev/adbc/apama-test/system/correlator-debug/testcases/
           correctness/Corr_Debug_cor_002/Input/test.mon:35
        foo.baz.test.runtest[master(2)/foo.baz.test(3)]
1 )
        C:/dev/adbc/apama-test/system/correlator-debug/testcases/
```

```
          correctness/Corr_Debug_cor_002/Input/test.mon:19
       foo.baz.test.::listenerAction::[master(2)/foo.baz.test(3)]
```

You can use these frame numbers (frame IDs) as arguments to the `engine_debug inspect` command.

To see just the contents of the top frame, run this command:

```
engine_debug stack --frame 0
```

Using the command-line debugger

Displaying variables with the command-line debugger

To list all variables in the current stack frame, enter the following:

```
engine_debug inspect
```

To obtain the value for a variable in the current stack frame, enter the following:

```
engine_debug inspect -variable variableName
```

To obtain the value for a variable further down the stack, run the `stack` command to determine the frame number and then enter the following:

```
engine_debug inspect -variable variableName -frame frameid
```

Using the command-line debugger

Using a replay log to diagnose problems

When you start the correlator, you can specify that you want it to copy all incoming messages to a special file, called a replay log. Incoming messages include the following:

- Events
- MonitorScript
- Java
- Correlator Deployment Packages (CDPs)
- Connection, deletion, and disconnection requests

A replay log is useful if there is a problem with either the correlator process or an application running on the correlator. If there is a problem, you can reproduce correlator behavior by replaying the messages captured in the replay log. If you are unable to diagnose the problem, you can provide the replay log to Software AG Global Support. A support engineer can then feed your replay log into a new correlator to diagnose the problem.

The information in the following topics can help you implement and use a replay log:

- "Comparison of full replay logs and input-only replay logs" on page 158
- "Creating a replay log" on page 158
- "Rotating the replay log" on page 159
- "Command line examples for creating a replay log" on page 159
- "Performance when generating a replay log" on page 160
- "Reproducing correlator behavior from a replay log" on page 160

Comparison of full replay logs and input-only replay logs

You can generate two different kinds of replay logs:

- A full replay log lets you replay the exact same behavior, ordering of emit operations, and so on, as the run that generated the replay log.
- An input-only replay log contains only the external inputs to the correlator. There is no information about multi-context ordering. Consequently, if there is more than one context, there is no guarantee that you can replay execution in the same order as the original execution.

Following is a comparison of full and input-only replay logs:

- Input-only replay logs are smaller and the correlator can write them faster than full replay logs.
- Full replay logs always generate the same output and state. Input-only replay logs generate the same output and state only when there is a single context.
- Both full replay logs and input-only replay logs contain information that is sufficient for auditing what events were sent to the correlator.

Creating a replay log

To create a replay log, specify one of the following options when you start a correlator:

```
--replayLog filename[${START_TIME}][${ID}].log
--inputLog filename[${START_TIME}][${ID}].log
```

Option	Description
--replayLog	Indicates that you want to generate a full replay log. See "Comparison of full replay logs and input-only replay logs" on page 158.
--inputLog	Indicates that you want to generate a replay log from only input events. Both input-only replay logs and full replay logs record incoming events from `engine_send` and operations that change the contents of the correlator, such as `engine_inject`, `engine_delete`, or equivalents, and record connections and disconnections.See "Comparison of full replay logs and input-only replay logs" on page 158.
filename	Replace filename with the name of the file that you want to be the replay log. If you also specify `${START_TIME}` and/or `${ID}`, the correlator prefixes the filename you specify to the time the file was started and/or an ID, beginning with `001`. See "Command line examples for creating a replay log" on page 159.Be sure to specify a location that allows fast access. If you specify the name of a file that exists, the correlator overwrites it.
`${START_TIME}`	Tag that indicates that you want the correlator to insert the date and time that it starts sending messages to the replay log into the filename of the replay log. Optional, however you probably want to always specify this option to avoid overwriting replay logs. See "Command line examples for creating a replay log" on page 159. This tag is also useful for correlators that you start from Apama's Management and Monitoring console, because it lets you distinguish the replay logs from different correlators.

Option	Description
`${ID}`	Tag that indicates that you want the correlator to insert a three-digit ID into the filename of the replay log. The ID that the correlator inserts first is `001`. Optional. The ID allows you to break up the replay log into a sequence of replay logs. A sequence of replay logs have the same name except for the ID. The log ID increment is related only to the specification of the rotate log option when you run the engine management tool. To rotate the replay log, invoke the `engine_management` utility and specify the `-r rotateReplayLog` option. Each time you rotate the log, the correlator closes the replay log it was using and starts a new replay log file and increments the ID portion of the filename by 1. See "Command line examples for creating a replay log" on page 159. This tag is useful when you plan to rotate the replay log. See the next topic for details. Note that restarting the correlator always resets the ID portion of the replay log filename to `001`.

In addition, specify any other options that you would normally specify when you start the correlator.

Rotating the replay log

While the replay log can get rather large, most file systems can handle large replay logs with no special action on your part. However, you might encounter one of the following situations:

- You want to archive your replay logs.
- Your operating system enforces a limit on file size.
- The replay log has become too large.

In these situations, you can rotate the replay log. Rotating the replay log means that the correlator closes the current replay log and starts sending messages to a new replay log. You should rotate the replay log only when you have a specific need to do so. You do not want to have thousands of replay logs in a directory since file systems do not handle this efficiently.

If you plan to rotate replay logs, specify the `${ID}` tag when you specify the `--replayLog` option when you start the correlator.

To rotate the replay log, invoke the `engine_management` utility and specify the `-r rotateReplayLog` option. The name of the new replay log is the same as the name of the closed replay log except that the correlator increments the ID portion of the replay log filename by `1`.

Command line examples for creating a replay log

The following command starts a correlator and specifies that the name of the replay log is `replay.log`.

```
correlator -l license.txt --replayLog replay.log
```

Suppose that the correlator processes events for a while, sends information to `replay.log`, and then you find that you need to restart the correlator. If you restart the correlator and specify the exact same command line, the correlator overwrites the first `replay.log` file. To avoid overwriting a replay log, specify `${START_TIME}` when you start the correlator. For example:

```
correlator -l license.txt  --replayLog replay_${START_TIME}.log
```

This command opens a replay log with a name something like the following:

```
replay_2008-07-12_15:12:23.156.log
```

This ensures that the correlator does not overwrite a replay log file. Now suppose that you want to be able to rotate the replay log, so you specify the `${START_TIME}` and `${ID}` tags:

```
correlator -l license.txt  --replayLog replay_${START_TIME}_${ID}.log
```

This command opens a replay log with a name something like the following:

```
replay_2008-07-12_15:12:23.156_001.log
```

If you then rotate the replay log, the correlator closes that file and opens a new file called

```
replay_2008-07-12_15:12:23.156_002.log.
```

UNIX Note

In most UNIX shells, when you start a correlator you most likely need to escape the tag symbols, like this:

```
correlator -l license --replayLog replay_\${START_TIME}_\${ID}.log
```

Performance when generating a replay log

When the file system that hosts the replay log is fast, generating a replay log should not have any noticeable effect on correlator performance. Consequently, the recommendation is to always run correlators that send information to replay logs. Just make sure you have enough disk space for the replay log. You need to monitor repeated use to determine how much space is required.

With the correlator generating a replay log, you can implement your application so that it sends a minimum amount of information to the correlator status log. You do not need to log application information because you can always recover application information from the replay log. Implementing an application that sends large amounts of application information to the correlator status log can negatively impact performance.

Another way to improve performance is to generate an input-only replay log rather than a full replay log. See "Comparison of full replay logs and input-only replay logs" on page 158.

Reproducing correlator behavior from a replay log

To use a replay log to reproduce correlator behavior, you must do the following:

1. Run the `extract_replay_log` Python utility on your replay log.
2. Start a correlator on the default port (`15903`).
3. Run the `replay_execute` script that the `extract_replay_log` utility generates.

Invoking the extract script

The `extract_replay_log.py` script is in the `utilities` directory in your Apama installation directory. You must have at least Python 2.4 to run this utility. You can download Python from http://www.python.org. If you are using Linux or Solaris, you probably already have Python installed.

The format for running the `extract_replay_log` utility is as follows:

```
extract_replay_log.py [options] replayLogFile
```

Replace *replayLogFile* with the path for the replay log you want to extract. If you specify the first replay log in a series, the subsequent replay logs must be in the same directory as the first replay log.

The options you can specify are as follows:

Option	Description
`-o=`*dir* or `--output=`*dir*	Specifies the directory that you want to contain the output from the `extract_replay_log` utility. The default is the current directory.
`-l=`*lang* or `--lang=`*lang*	Specifies the language of the script that the `extract_replay_log` utility generates. Replace *lang* with one of the following: ● `shell` generates the `replay_execute.sh` UNIX shell script. ● `batch` generates the `replay_execute.bat` Windows batch file. This is the default.
`--noReplayMode`	Indicates that you do not want to extract information needed to replay non-deterministic behavior in its original order. The script extracts all input events but not enough information to reproduce non-deterministic behavior in the same order as when the replay log captured it. Therefore the results of replaying the non-deterministic behavior might be different than the original results. For details, see "About replay logs and parallel processing" in the parallel processing section of *Developing Apama Applications in EPL*. This option has meaning only if you generated the replay log with the `--replayLog` option rather than the `--inputLog` option.
`-v` or `--verbose`	Indicates that you want verbose utility output.
`-h` or `--help`	Displays help for the utility.

The `extract_replay_log` utility generates the following:

- A script whose execution duplicates the correlator activity captured by the replay log.
- Event files — each one is prefixed with `replay_`.
- MonitorScript and possibly JAR and Correlator Deployment Package (CDP) files — each one is prefixed with `replay_`.

Starting the correlator

Before you invoke the `replay_execute` script, start the correlator that you want to use to replay the log. Start this correlator on the default port (`15903`) and specify the following in the correlator startup command:

- Specify the `--replayMode` option to indicate that you are going to invoke the `replay_execute` script. This option:
  - Enables replay of non-deterministic correlator behavior.
  - Implies the `-XignoreEnqueue` option, which instructs the correlator to ignore the output of `enqueue` statements. The replay log captures enqueued events, as well as when enqueued events appear on the input queues of public contexts. During replay, the correlator uses the captured events, not the newly enqueued events.
  - As mentioned before, the correlator typically processes events faster during replay than during the original activity. During replay, you want to ensure that the correlator processes enqueued events in the same place in the input stream as during the original correlator

activity. Consequently, you want the replay correlator to use the captured enqueued events and to process them according to the captured processing order.

  - Implies the `-Xclock` option, which disables the correlator's internal clock. The replay log captures the time ticks during the original correlator activity. During replay, you want the correlator to use those captured time ticks. This ensures that the reproduced behavior reflects the original relationship between time ticks and the event stream.

- Specify any other options that you specified when you started the correlator whose behavior was captured. For example, if the replay log will feed JAR files into the correlator, specify the `-j` option and also specify the `-J` option to pass the same options or properties that were previously passed to the embedded JVM.
- If the captured behavior includes the use of the `integer.rand()` and/or `float.rand()` functions, be sure to specify the `-XsetRandomSeed` option. This option starts the correlator with the random seed value you specify. To exactly replay the captured behavior, specify the integer that the `extract_replay_log.py` script identifies as the random seed.

The correlator uses the random seed to calculate the random numbers returned by the `rand()` functions. The same random seed returns the same sequence of random numbers.

Invoking the replay script

If you want, you can edit the generated event files, EPL files, or JAR files to slightly modify the behavior you are about to replay. For example, you might add logging for debugging purposes. However, there are restrictions on what you can change:

- You cannot insert any of the following:
  - calls to `integer.getUnique()` or `rand()`
  - `emit`, `spawn...to`, `enqueue`, or `enqueue..to` statements
  - `context` constructors
- You cannot change the number of parseable events sent to the correlator. For example, you cannot attach a dashboard component to the replay log because the dashboard components work by sending events to the correlator.
- You cannot change the number of event definitions and monitors injected.

Making any of these changes can potentially alter the behavior of later operations.

If you are using the MemoryStore and the correlator reads or writes to a store on disk then to accurately play back execution you must have a copy of that store as it was before the correlator modified it. Also, if you are using the MemoryStore from multiple contexts it is unlikely to replay correctly because the order of interaction with the MemoryStore is not preserved in the replay log.

When the correlator is running in the correct mode, and you have optionally edited the generated files, you are ready to invoke the `replay_execute` script. The `replay_execute` script tries to replay the contents of the replay log into the correlator running on the default port.

While the correlator exactly reproduces the activity captured in the replay log, it can execute the same activity faster during replay than when it was executed originally. This is because the correlator already has all the events it needs to process; it does not have to wait for any events. Replaying a log is typically significantly faster than original correlator activity. It is possible that you will find that the time it takes to replay a log is not much less than the time it took for the original activity. In this case, it is possible you were running too close to capacity during the original run. If that is the case, you risk not being able to keep up with the event flow during regular correlator execution. If you anticipate higher event flow then you should investigate optimizing your application or running it on a faster computer.

Event file format

You can use the `engine_send` tool to stream a sequence of events through the event correlator. The `engine_send` tool accepts input from one or more data files to support tests or simulations, or from `stdin` to allow dynamic generation of events. In the latter case, you can generate events from user input or by piping output from an event generation program to `engine_send`. In all cases, `engine_send` requires event data formatted as described in this section.

The `engine_receive` tool outputs events in this same file format. This means you can use events generated by the `engine_receive` tool as input to a second event correlator that is executing the `engine_send` tool.

Event Correlator Utilities Reference

Event representation

A single event is identified by the event type name and the values of all fields as defined by that type. Event type names must be ‘fully-qualified’ by prefixing the package name into which the corresponding event type was injected, unless the event was injected into the default package

Each event is given on a separate line, separated by a new-line character. Only single-line comments are allowed. Start each comment line with // or #. Any blank lines are ignored.

For example, the following:

```
// This is an event file
     // that contains some sample events.
// Here are three stock price events:
StockPrice("XRX", 11.1)
StockPrice("IBM",130.6)
StockPrice("MSFT", 70.5)
```

are three valid events given the following event type definition (injected into the default package):

```
event StockPrice {
  string stockSymbol;
  float stockValue;
}
```

If the above events were saved in an `.evt` file, `engine_send` would send each event in turn, as soon as the previous event finished transmission. This behavior can optionally be modified in two ways: specifying that events should be sent in a ‘batch’ and specifying timing data forcing events to be sent at specific time intervals.

Event file format

Event timing

In addition to batching events, it is possible to specify the time intervals at which each batch should be sent to the correlator. This is achieved by adding a time offset, in milliseconds, after the `BATCH` tag. For example, the following:

```
BATCH 50
StockPrice("XRX", 11.1)
StockPrice("IBM", 130.6)
StockPrice("MSFT", 70.5)
BATCH 100
StockPrice("XRX", 11.0)
StockPrice("IBM", 130.8)
StockPrice("MSFT", 70.1)
```

specifies two batches of events to be sent 50 milliseconds apart.

The addition of a ‘time’ allows simulations of ‘bursts’ of events, or more random distributions of event traffic. Times are measured as an offset of from when the current file was opened. If only one file of events is being read and transferred, then this would be the same as since the start of a run (i.e. from the time that the `engine_send` tool starts processing the event data). If multiple files are being read in, the timing starts all over again upon the (re)opening of each file.

If the time given for a batch is less than the current time, or if no time is given following a `BATCH` tag (or if no `BATCH` tag is provided) then the events are sent as soon as they are read in, immediately following the preceding batch.

Event file format

Event data types

The following example illustrates how each data type is specified in an event representation. Given the event type definitions:

```
event Nested {
  integer i;
}
event EveryType {
  boolean b;
  integer i;
  float f;
  string s;
  location l;
  sequence<integer> si;
  dictionary<integer, string> dis;
  Nested n;
}
```

the following is a valid event representation for an `EveryType` event:

```
EveryType (
  true,                 # boolean is true/false (lower-case)
  -10,                  # positive or negative integer
  1.73,                 # float
  "foo",                # strings are (double) quoted
  (1.0,1.0,5.0,5.0),    # locations are 4-tuples of float values
  [1,2,3],              # sequences are enclosed in brackets []
  {1:"a",2:"b"},        # dictionaries are enclosed in braces {}
  Nested(1)             # nested events include event type name
)
```

Note that this example is split over several lines for clarity; in practice this definition would all be written on the same line.

Types can of course be nested to create more complex structures. For example, the following is a valid event field definition:

```
sequence<dictionary<integer, Nested> >
```

and the following is a valid representation of a value for this field:

```
[{1:Nested(1)}, {2:Nested(2)}, {3:Nested(3)}]
```

Event file format

Using the data player command-line interface

Apama Studio's Data Player lets you play back previously saved event data as you develop your application. During playback, you can analyze the behavior of your application. Or, if you modify the saved event data, you can analyze how your application performs with the altered data. Apama Studio plays back event data that has been stored in standard data formats.

When you are ready to test your application the command-line interface to the Data Player lets you write scripts and unit tests to exercise the API layers. Or, if you just want to play back events to the correlator, using the command-line interface might be easier than using the Data Player GUI in Apama Studio.

To use the command-line interface to the Data Player, you must have already used the GUI interface in Apama Studio. That is, you must have already defined queries and query configurations in Apama Studio. When you use the command-line interface, you specify query names and query configurations that you created in Apama Studio.

The Data Player relies on Apama Database Connector (ADBC) adapters that are specific to standard ODBC and JDBC database formats as well as the comma-delimited Apama Sim format. Apama release 4.1 and earlier captured streaming data to files in the Sim format. These adapters run in the Apama Integration Application Framework (IAF), which connects the data sources to the correlator. The information here assumes that you are already familiar with the information in "Using the Data Player" in *Using Apama Studio*.

Synopsis

To use the Data Player from the command line, enter `adbc_management.bat` (Windows) or `adbc_management` (UNIX) from the `bin` directory in your `APAMA_HOME` directory:

To obtain the following usage message, run the command with the `help` option:

```
Usage:
adbc_management --query <queryName> --configFile <file> [ options ]
Where options include:
-h | --help              This message
-n | --hostname <host>   Connect to correlator on <host>.
                         Default to localhost.
-p | --port <port>       Connect to correlator on <port>.
                         Default to 15903.
--query <queryName>      Run the query <queryName> specified in
                         query configuration file.
--configFile <file>      Query configuration file to use.
--username <user>        Optional username for database connection.
--password <password>    Optional password for database connection.
--returnType <returnType>  Optional returnType of playback events.
                         Default is native.
--backTest <true|false>  Optional switch to generate time event from
                         data.
                         Use false if correlator is not running
                         -Xclock option. Default is true.
--speed <playBackSpeed>  Optional speed for playing back query.
                          <= 0.0, as fast as possible
                          > 0.0 for some multiple of playback speed.
                          (Ignored if --backTest false is used.)
```

Options

Option	Description
`-h`	Display usage information
`-n`	Name of the host on which the event correlator is running (default is `localhost`). Non-ASCII characters are not allowed in host names.
`-p`	Port on which the event correlator is listening (default is `15903`).
--query *queryName*	Run the specified query, which is defined in the query configuration file that you identify with the `--configFile` option. This is a query you created in Apama Studio in the Data Player Editor. You did this when you clicked on the + button on the action bar. You specified a query name and that is the name you need to specify here
--configFile *file*	Use this query configuration file. Specify the query configuration file associated with your project. In Apama Studio, the query configuration file is always called `dataplayer_queries.xml` (in the project/`config` directory).
--username *user*	The user name to use for the database connection. Optional.
--password *password*	The password to use for the database connection. Optional.
--returnType *returnType*	The type of the playback events returned. The default is `Native`. The only other choice is `Wrapped`. A return type of `Native` means that each matching event is sent as-is to the correlator. When you specify `Wrapped`, each matching event is inside a container event. The name of the container event is `Wrapped` followed by the name of the event in the container, for example, `HistoricalTick`. Event wrapping allows events to be sent to the correlator without triggering application listeners. A separate, user-defined monitor can listen for wrapped events, modify the contained event, and reroute it such that application listeners can match on it.
--backTest <true|false>	This option is equivalent to Apama Studio's Data Player option to "Generate time event from data". When the correlator is running with the `-Xclock` option, time in the correlator is controlled by `&TIME()` events. This is how the Data Player controls the playback speed. If the correlator is not running with the `-Xclock` option, the correlator keeps its own time. The default is true, which means that the correlator is running with the -Xclock option. Set this option to false when the correlator is not running with the `-Xclock` option.
--speed *playBackSpeed*	Specifies the speed for playing back the query. Optional. A float value less than or equal to `0.0` means that you want the correlator to play it back as fast as possible. A float value greater than 0.0 indicates a multiple for the playback speed. To play at normal speed, specify `1.0`.For half normal speed, specify `0.5`.For twice normal speed, specify `2.0`.For 100 times normal speed, specify `100.00`.

Event Correlator Utilities Reference

9 Tuning Correlator Performance

This section addresses how to scale up Apama to improve upon the performance of a single event correlator. It describes the Apama features you can use to route events to multiple event correlators to increase an application's capacity.

Scaling up Apama

Apama provides services for real-time matching on streams of events against hundreds of different applications concurrently. This level of capacity is made possible by the advanced matching algorithms developed for Apama's event correlator component.

Should it prove necessary, capacity can further be increased by using multiple event correlators. To facilitate such multi-process deployments, Apama provides features to enable routing of events to multiple event correlators, possibly running on different host computers, and to assist in the configuration of scaled-up topologies.

There are two methods of configuration: using the advanced configuration tools or programmatically through a client programming API. This guide describes both approaches, but first discusses different ways in which Apama can be distributed and what factors affect the choice of the distribution strategy.

Note: This topic focuses on scaling Apama for applications written in EPL, although the same techniques and tools can be used for Apama Java applications. Knowledge of aspects of EPL is assumed, specifically monitors, spawning, listeners and channels. Definitions of these terms can be found in "Getting Started with Apama EPL" in *Developing Apama Applications in EPL* (available if you selected Developer during installation).

The core event processing and matching service offered by Apama is provided by one or more event correlator processes. In a simple deployment, Apama comprises a single event correlator connected directly to at least one input event feed and output event receiver. Although this arrangement is suitable for a wide variety of applications (the actual size depending on the hardware in use, networking etc. available), for some high-end applications it may be necessary to scale up Apama by deploying multiple event correlator processes to partition the workload across several processors and/or machines.

Tuning Correlator Performance

Partitioning strategies

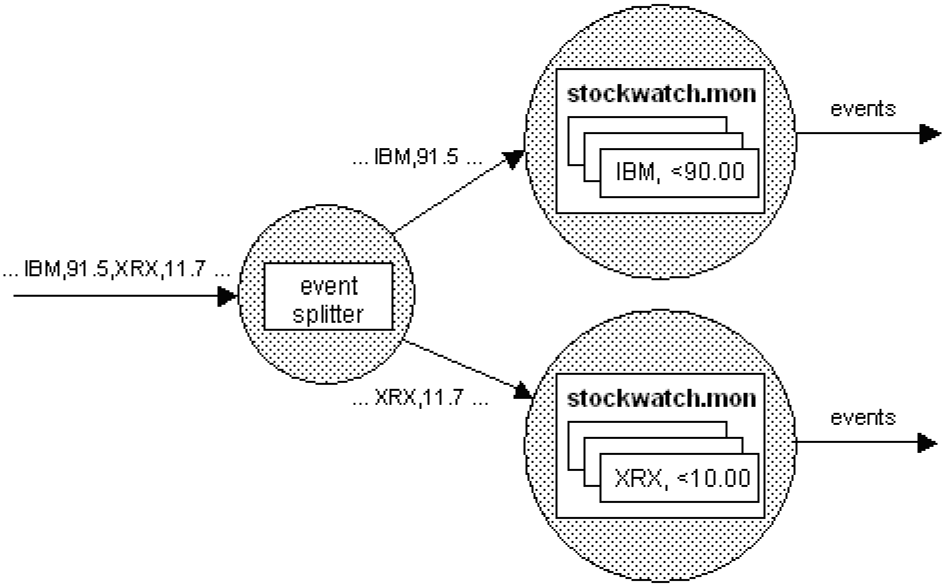
Using the tools and libraries described in this guide it is possible to configure the arrangement of multiple event correlators within Apama (the engine topology). It is important to understand that the appropriate engine topology for an application is firmly dependent on the partitioning strategy to be employed. In turn, the partitioning strategy is determined by the nature of the application itself, in terms of the event rate that must be supported, the number of monitors and/or spawned monitors expected and the inter-dependencies between monitors and events.

To illustrate this we will consider two examples.

The 'Stockwatch' sample application monitors for changes in the values of named stocks and emits an event should a stock of interest fall below a certain value (see `samples\monitorscript\stockwatch` in the directory where Apama is installed). The stocks to watch for and the prices on which to notify are set up by initialization events, which cause monitors to be spawned, containing the relevant details. In this example we assume the need for partitioning arises from a very high event rate (say hundreds of thousands of stock ticks per second) – too high a rate for a single event correlator to process in a sustainable fashion.

A suitable partitioning scheme here might be to split the event stream, such that a single event correlator processes only subsets of the events. The illustration below shows how this can be achieved, with the event stream split between two event correlators, each of which contains a duplicate of the original `stockwatch` monitor.

Figure 6. Partitioning strategy for the stockwatch application



The diagram above illustrates partitioning of the event stream on the basis of stock symbol. In this scheme the spawned monitors, which are looking for a specific stock/price combination, reside on the correlator to which events for that stock are sent (else relevant events will be missed).

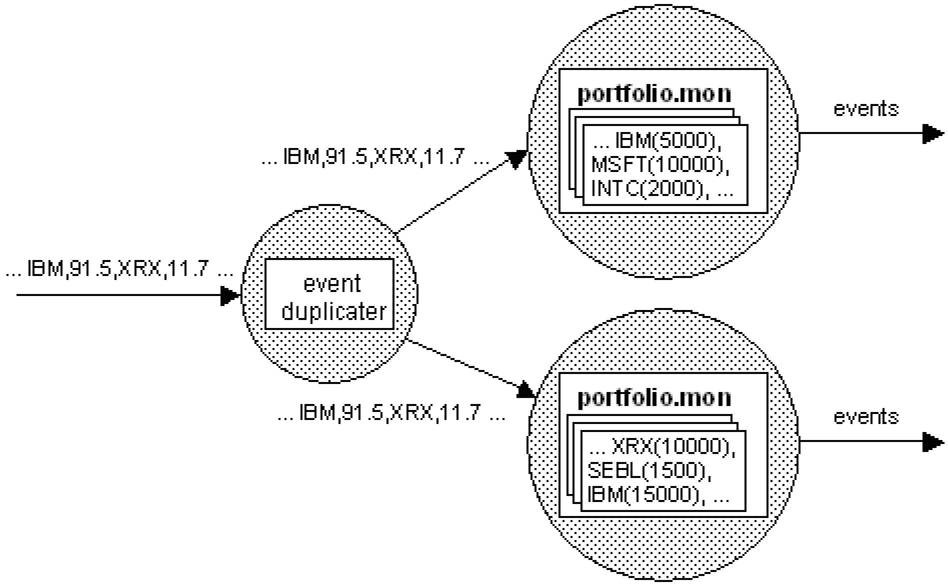
In the case of Stockwatch, this is achieved by splitting the initialization events that set up the spawned monitors using identical criteria as those used to split the stock tick events themselves, and by duplicating the `stockwatch` monitor on each of the correlators involved. So in figure 1, the event that causes spawning of a monitor to detect the stock price of IBM falling below $90 is sent to the event correlator that receives IBM stock tick events.

In the second example we will consider a portfolio monitoring application that monitors portfolios of stocks, emitting an event whenever the value of a portfolio (calculated as the sum of stock price * volume held) changes by a percentage. We assume a single spawned monitor manages each portfolio, and any stock can be added to/removed from a portfolio at any time by sending suitable events.

This application potentially calls for significant processing with each stock tick, as values of all portfolios containing that stock must be re-calculated. If the number of portfolios being monitored grows very large, it may not be possible for a single event correlator to perform the necessary recalculations for each stock tick, thus requiring the application to be partitioned across multiple correlators.

Unlike the stockwatch application, it is not possible to achieve scaling to larger numbers of portfolios by splitting the event stream. Each portfolio can contain any stock, and stocks can be dynamically added and removed, thus it cannot be predicted which events a monitor will need to see during its lifetime. In this case, a suitable partitioning scheme is to partition the monitors across correlators (as with stockwatch) but to duplicate rather than split the event stream to each event correlator. This is illustrated below.

Figure 7. Partitioning strategy for the portfolio monitoring application


... IBM,91.5,XRX,11.7 ...
event duplicater
... IBM,91.5,XRX,11.7 ...
portfolio.mon
... IBM(5000), MSFT(10000), INTC(2000), ...
events
... IBM,91.5,XRX,11.7 ...
portfolio.mon
... XRX(10000), SEBL(1500), IBM(15000), ...
events

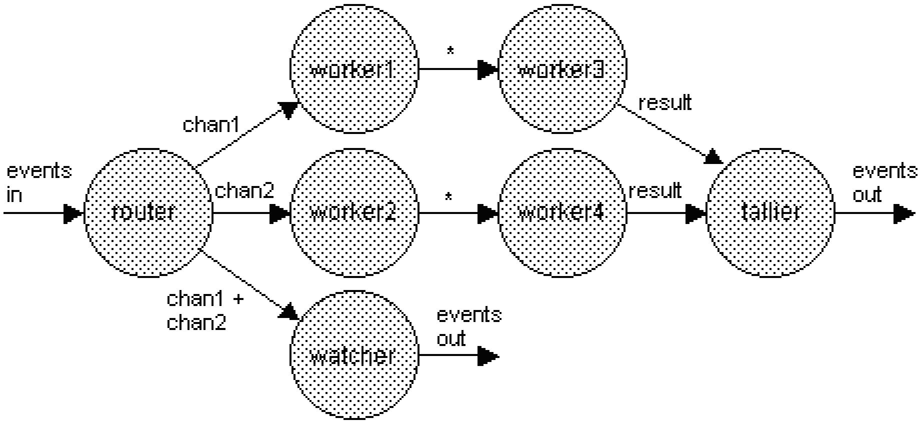
In summary, the partitioning strategy can be thought of as a formula for splitting and duplicating monitors and/or events between event correlators whilst preserving the correct behavior of the application. In some circumstances, it may be necessary to re-write monitors which work correctly on a single correlator to allow them to be partitioned across correlators, as the following section describes.

Tuning Correlator Performance

Engine topologies

Once the partitioning strategy has been defined, in terms of which events and monitors go to which correlators, it is necessary to translate this into an engine topology. This is achieved by connecting source and target event correlators on separate channels, such that events emitted by a source correlator on a specific channel find their way to the correct target correlator. A set of two or more event correlators connected in this way is known as a correlator pipeline, illustrated in the following illustration. This figure represents an example topology for a high-end application – the majority of applications use a single correlator only, or have far simpler topologies.

Figure 8. Example of a pipelined engine topology



In the illustration above, an event correlator can perform the function of each of the 7 nodes (router, worker, watcher, tallier). Each target correlator performs some processing before passing the results to a second worker correlator (`worker3`, `worker4`) in the form of events, emitted on the default channel (`*`). `tallier` collates the results from these correlators for forwarding to any registered receivers. A final correlator, `watcher`, monitors the events emitted by `router` on `chan1` and `chan2` and emits events (possibly containing status information or statistical analysis of the incoming event stream) to any registered receivers.

To deploy an application on a topology like that shown above requires separating the processing performed into a number of self-contained chunks. In figure 3 it is assumed the core processing can be serialized into three chunks, with the first two chunks split across two correlators each (`worker1/2` and `worker3/4` respectively) and the third chunk residing on a single correlator (`tallier`). Intermediate results from each stage of processing are passed to the next stage as emitted events, which connected correlators receive by registering on the appropriate channels.

To realize this application structure requires coding each chunk of processing as one or more separate monitors, which emit intermediate results as an event of known type on a pre-determined channel. These monitors can then be loaded onto the appropriate correlator. This may require an existing application, which grows beyond the capacity of a single event correlator, to be re-written as a number of (smaller) monitors to allow partitioning of the application processing into separate chunks in the manner described above.

Tuning Correlator Performance

Event correlator pipelining

To implement engine topologies comprising multiple event correlators requires a method of connecting correlator instances in pipelined configurations. This can be achieved in the following ways:

- Directly using the `engine_connect` tool, see "Configuring pipelining with engine_connect" on page 172.
- Programmatically via the C++ client API, see "Configuring pipelining through the client API" on page 176.

- Using the Enterprise Monitoring and Management console (EMM). The easiest way to connect components together is to add the components to an EMM configuration and set up connections using the Connections tab; see "Connections tab" on page 60.

Configuring pipelining with engine_connect

The `engine_connect` tool allows direct connection and disconnection of running correlator instances.

The tool is located in the `bin` folder of the Apama installation, as `engine_connect.exe` (on Windows) or `engine_connect` (on UNIX). Note that before you run it you either need to ensure that the environment variables described in the *Installation Guide* have been set, or else run it from within the Apama Command Prompt.

Synopsis

```
Usage: engine_connect [ options ]
To connect a correlator, as an event receiver, to another correlator.
Where options include:
   -h  | --help                This message
   -sn | --sourcehost <host>   Source engine on <host>
   -sp | --sourceport <port>   Source engine is listening on <port>
   -tn | --targethost <host>   Target engine on <host>
   -tp | --targetport <port>   Target engine is listening on <port>
   -c  | --channel <channel>   Listen on output channel <channel>
   -x  | --disconnect          Destroy rather than create connections
   -s  | --qdisconnect         Disconnect if slow (only takes effect on
                               first connection)
   -f  | --filename <file>     Read config data from a file
   -u  | --utf8                Assume config file is in UTF8
    -p  | --persistent         Receiver will reconnect on being disconnected
   -v  | --verbose             Be more verbose
   -V  | --version             Print program version info
   The target is connected as an event consumer to the source
   Multiple -c options may be given
   To read from stdin use -f -
```

Description

`engine_connect` connects a single target correlator (the receiver) to a single source correlator (the sender) on one or more named channels. Source and target correlators must already be running. Alternatively, if the `-f` option is specified, `engine_connect` reads connection information from the specified file and sets up each connection found therein (see "Configuring pipelining through the client API" on page 176 for details of the file format).

The `engine_connect` utility expects the specified file to be in the local character set. If the configuration file is in UTF-8, specify the `-u` option in addition to the `-f` option.

If the filename provided to `-f` is '-' then connection information is read from the standard input device (`stdin`) until end-of-file.

If the `-x` option is specified, `engine_connect` will disconnect any existing connection between source and target correlator(s) on the specified channels.

The tool is silent by default unless an error occurs – for verbose progress information use the `-v` option.

Options

Option	Description
-h	Display usage information
-sn	Name of the host on which the source (event sending) correlator is running. The default is localhost. However, you can use the default or specify localhost only when the source correlator and the target correlator are running on the same host. In all other situations, you must specify the public IP address or the name of the host. This ensures that the host of the target correlator can resolve the name/address of the source correlator host. Non-ASCII characters are not allowed in host names.
-sp	Port on which the source (event sending) correlator is listening (default is 15903)
-tn	Name of the host on which the target (event receiving) correlator is running. The default is localhost. However, you can use the default or specify localhost only when the source correlator and the target correlator are running on the same host. In all other situations, you must specify the public IP address or the name of the host. This ensures that the host of the source correlator can resolve the name/address of the target correlator host. Non-ASCII characters are not allowed in host names.
-tp	Port on which the target (event receiving) correlator is listening (default is 15903)
-c	Named channel to connect/disconnect on – if -c is not specified then: a) if -x is specified the target will receive no events emitted by the source – any existing connection will be removed b) if -x is not specified the target will receive all events emitted by the source, regardless of channel
-x	Disconnect the source/target correlator(s) on the named channels only
-s	Disconnect if slow (only takes effect on first connection)
-f	Read connection information from the named file. If this option is specified, the options -sn -sp -tn -tp -c are all ignored.This file must be in the local character set or in UTF-8 format. If it is UTF-8, specify the -u option in addition to this option.
-u	Indicates that the connection information file is in UTF-8.
-p	Stores details of the connection in the receiving component. If the sender goes down or a network partition occurs, the receiver will attempt to reconnect. If the receiver is a correlator or IAF, the connection will have the same lifetime as the receiving process, so engine_connect should be re-sent when restarting the correlator. Persistent connections are stored in a correlator dump file, so there is no need to re-run engine_connect after engine_load.
-v	Requests verbose output
-V	Displays program name and version number

Operands

None.

Exit status

The following exit values are returned:

Status	Description
0	All connections were established successfully.
1	One or more source correlators could not be contacted
2	One or more target correlators could not be contacted
3	A problem occurred establishing the connection; request invalid
4	Target correlator failed to contact the Source
5	Some other error occurred

Tuning Correlator Performance

Connection configuration file

engine_connect can take connection information from a file for connecting and disconnecting event correlators. A sample of such a configuration file is shown below, which defines the topology shown in "Example of a pipelined engine topology" on page 171.

```
# comments are allowed prefixed by a '#' - the rest of the line
# is ignored
router:dopey.apama.com:1234
worker1:sleepy.apama.com:1234:router{chan1}
worker2:grumpy.apama.com:1234:router{chan2}
worker3:sneezy.apama.com:1234:worker1
worker4:bashful.apama.com:1234:worker2
tallier:happy.apama.com:1234:worker3{result},worker4{result}
watcher:doc.apama.com:1234:router{chan1,chan2}
```

Connection configuration file format

Each entry in the configuration file specifies connection information for a single correlator in the deployment. Entries can be specified in any order. The general format of an entry is:

```
<correlator_name>[:<host>][:<port>][:<connection>[,<connection>
...]]
```

where <connection> is defined as:

```
<correlator_name>[ {<channel_name>[,<channel_name>...]
}]
```

<*correlator_name*> is a symbolic identifier for a correlator, used to identify source correlators in target correlator connection information. It can consist of any combination of characters other than whitespace, colon, comma or open/close brace characters, which are reserved as separators. <*host*> and <*port*> identify the specific correlator this entry applies to. They can be omitted, in which case the defaults of localhost and 15903 are used respectively.

Following this information are details of all connections to source correlators for the current (target) entry. This information is omitted if no correlators sit ‘upstream’ of the current entry (as with the correlator router, above). If there are multiple upstream source correlators, each name should be separated by a comma (as with tallier, above, which takes events from worker3 and worker4).

For each connection it is possible to specify the channel(s) on which the target correlator will listen. If no channels are specified the target correlator will register to receive all events emitted by the source correlator regardless of channel (as with correlators worker3 and worker4 which register for all events from worker1 and worker2 respectively). One can specify specific channel names by enclosing them in braces and separating multiple channels by commas (as with watcher which registers with router for all events on channels chan1 and chan2).

In effect, the configuration file is a convenient way of grouping several calls to engine_connect. For example to set up the connections for the correlator tallier would require two commands using engine_connect:

```
>engine_connect -sn sneezy.apama.com -sp 1234 -tn happy.apama.com
    -tp 1234 -c result
>engine_connect -sn bashful.apama.com -sp 1234 -tn happy.apama.com
    -tp 1234 -c result
```

Errors in the configuration file

The configuration file can be used to both establish and remove connections in a multi-correlator engine topology. For example, assuming the above file is saved as topology.dat, the following commands will first set up then tear down all the connections specified therein:

```
>engine_connect -f topology.dat
>engine_connect -x -f topology.dat
```

In each of these cases, engine_connect will exit with non-zero exit status on the first error it detects in the configuration file. An error message will be printed to standard error (stderr).

Re-playing the configuration file

The behavior of engine_connect without the -x option is additive. This means that successive calls to engine_connect will attempt to add the channels specified to any existing connection between the source and target correlator(s). For example, with reference to the configuration file above, these commands:

```
>engine_connect -sn dopey.apama.com -sp 1234 -tn sleepy.apama.com
    -tp 1234 -c foo
>engine_connect -f topology.dat
```

will first add a connection from correlator router to worker1 on channel foo, then (from the configuration file) extend that connection so that worker1 also receives all events from router emitted on channel chan1.

Once a connection is set up between two correlators on a channel, any further attempt to set up that connection on the same channel will have no effect. It is therefore possible to re-play the configuration file by invoking engine_connect without creating duplicate connections. This can be useful if there is an error in the configuration file signaled when engine_connect is called, as the error can be fixed and engine_connect re-run without requiring removal of connections that were successfully set up by the first call to engine_connect.

Event correlator pipelining

Configuring pipelining through the client API

Apama provides a C++ Client Software Development Kit (SDK). This allows software written in C++ to interface with a running event correlator or group of event correlators. Apama management tools such as `engine_connect` are written using this Client Software Development Kit.

The functionality of the Client SDK is found in the `lib` folder of the Apama installation and consists of the libraries `libengine_client.so.5.1` (on UNIX), or `engine_client5.1.lib` (on Windows). To code against the library use the definitions from the `engine_client_cpp.hpp` header file in the `include` folder.

Detailed information on how to use the integration library is available in "The C Cient Software Development Kit" in *Developing Clients* (available if you selected **Developer** during installation); this section looks at the specific methods that allow a developer to programmatically configure two event correlators to communicate in a pipelined arrangement.

The primary class contained in the library is

```
com::apama::engine::EngineManagement
```

An object instance of this class represents an event correlator within Apama, and allows a developer to:

- inject EPL files,
- delete EPL entities,
- send events into a correlator,
- get a correlator’s current operational status,
- connect a receiver of events,
- connect a correlator as a consumer of another correlator.

The last capability is of direct interest here, and is supported through the following methods on the `EngineManagement` class;

```
/**
 * Connect this Event Correlator as an event receiver to another
 * Event Correlator.
 *
 * @param target The Correlator to connect to
 * @param channels  An array of names representing the channels to subscribe
 *  to. This is a null-terminated array of pointers to zero-terminated
 *  char arrays. If this is null or empty, subscribe to all channels.
 * @return true if successful
 * @exception EngineException
 */
virtual bool attachAsEventConsumerTo(
    EngineManagement* target, const char* const* channels)
    =0;
/**
 * Unsubscribe as an event receiver from another Event Correlator.
 *
 * @param target The Correlator to unsubscribe from.
 * @param channels An array of names representing the channels to
 *   unsubscribe from. This is a null-terminated array of pointers to
 *   zero-terminated char arrays. If this is null or empty unsubscribe
 *   from all channels.
* @exception EngineException
 */
virtual void detachAsEventConsumerFrom(
  EngineManagement* target, const char* const* channels)
   =0;
```

The following sample code illustrates how to connect two correlators in a pipelined arrangement. The calls highlighted in bold are the important library calls.

```
// Method that connects to 2 ‘Engines’ (Correlators) and links them up
// in a pipeline
void connect(const char* host1, unsigned short port1,
  const char* host2, unsigned short port2,
  vector<const char*> channels, bool detach,
  int& rc )
{
  const char* emsg = "";
  EngineManagement* engine1 = NULL;
  EngineManagement* engine2 = NULL;
  try {
    // a bit verbose
    cerr << "Requesting connection from host:" << host1
      << ", port:" << port1 << " to host:" << host2
      << ", port:" << port2 << endl;
    // Ensure valid arguments
    rc = 3; // this just sets a return value
    if ((host1==host2) && (port1==port2)) {
      cerr << "Connecting an engine to itself is disallowed"
        << endl;
      return;
  }
    // Make sure the channel list is NULL-terminated
    channels.push_back(NULL);
    // Attempt to connect to source Correlator
    rc = 1;
    emsg = "Failed to connect to source engine";
    if (!(engine1 =
      com::apama::engine::connectToEngine(host1, port1))) {
      throw EngineException(emsg);
    }
    // Attempt to connect to target Correlator
    rc = 2;
    emsg = "Failed to connect to target engine";
    if (!(engine2 =
      com::apama::engine::connectToEngine(host2, port2))) {
      throw EngineException(emsg);
    }
    // Connect target to source
    if (detach) {
      emsg = "Detach failed";
      engine2->detachAsEventConsumerFrom(engine1,&channels[0]);
    }
    else {
      emsg = "Attach failed";
      if (
        !engine2->attachAsEventConsumerTo(engine1,&channels[0]))
      {
        rc = 4;
        emsg = "Target engine could not connect to source engine";
        throw EngineException(emsg);
      }
    }
  }
  catch (EngineException ex) {
    string errmes;
    errmes += emsg;
    errmes += ": ";
    errmes += ex.what();
    throw EngineException(errmes.c_str());
  }
  // Shutdown cleanly
  rc = 5;
  if (engine1) {
    // Disconnect from Apama
    emsg = "Failed to disconnect from source engine";
    com::apama::engine::disconnectFromEngine(engine1);
  }
  if (engine2) {
    // Disconnect from Apama
    emsg = "Failed to disconnect from target engine";
    com::apama::engine::disconnectFromEngine(engine2);
  }
}
```

Event correlator pipelining

Event partitioning

Using `engine_connect` or the Apama client library, it is possible to create topologies of correlators across which an application's monitors can be partitioned. Partitioning is achieved by using the `engine_inject` tool described in Injecting code into a correlator, or by means of the relevant functions of the client library, to load the relevant monitors directly on to the appropriate correlators.

This scheme is suitable for most applications, as monitors can be loaded once when Apama is brought online. For some applications, however, there is a requirement for a dynamic routing mechanism that (depending on the requirements of the application) continually splits and/or duplicates the incoming event stream and sends it to two or more correlators. This is achieved by using a correlator as a router. A correlator can function as a router upon the injection of EPL that determines which incoming events should be sent to each of the downstream correlators.

The EPL required to make a correlator act as a router, sending events to downstream components which need them but not to others can be very simple as in the following example. This EPL monitor listens for `StockTick` events and splits them onto two output channels on the basis of each tick's stock symbol.

```
event StockTick {
 string symbol;
 float price;
}
monitor Router {
 action onload {
  rule1();
  rule2();
 }
 action rule1 {
  StockTick st;
  on all StockTick(<"M", *):st {
   emit st to "chan1";
  }
 }
 action rule2 {
  StockTick st;
  on all StockTick(>="M", *):st {
   emit st to "chan2";
  }
 }
}
```

The expectation is that there is a correlator connected to each of the channels. This allows two correlators to process those events in parallel, potentially increasing the throughput of the system as a whole.

Event correlator pipelining

10 Using the Apama Database Connector

With the Apama Database Connector (ADBC), Apama applications can store and retrieve data in standard database formats. You can retrieve data using the ADBCHelper API or the ADBC Event API to execute general database queries or retrieve the data for playback purposes using the Apama Data Player. This section describes how to configure ADBC and how to use it to store and retrieve data.

For information about playing back data, see "Using the Data Player" in *Using Apama Studio*.

Overview

ADBC is implemented as an Apama adapter that uses the Apama Integration Adapter Framework (IAF) to connect to standard ODBC and JDBC data sources as well as to Apama Sim data sources.

When connected to JDBC or ODBC data sources, ADBC provides access to most open source and commercial SQL databases. With either of these data sources, Apama applications can capture events flowing through the correlator and play them back at a later time. In addition to storing and retrieving event data, Apama applications can store non-event data and execute queries against the data. Dashboards in Apama applications can directly access JDBC and ODBC database data.

An Apama Sim data source is a file with data stored in a comma-delimited format with a `.sim` file extension. Apama release 4.1 and earlier captured streaming data to files in this format. The Apama ADBC adapter can read `.sim` files but it does not store data in that format. For information on the format of `.sim` files, see "Format of events in .sim files" on page 224.

Apama provides Progress DataDirect ODBC and JDBC database drivers for the following Apama-certified databases (note, the ODBC drivers are available only for Windows platforms):

- DB2
- Microsoft SQL Server
- Oracle

Using the Progress DataDirect database drivers eliminates the need to install vendor-supplied drivers. In addition, they are pre-configured so when you select a Progress DataDirect database driver in an Apama Studio project, the adapter instance is automatically configured with appropriate settings.

The Progress DataDirect ODBC database drivers are licensed to be used only with the Apama ADBC adapter. The Progress DataDirect JDBC drivers can be used with any Apama component.

For more information on the supplied Progress DataDirect database drivers, see the Progress DataDirect Connect Series documentation available at `*apama_install_dir*\doc\db_drivers\jdbc\books.pdf` and `*apama_install_dir*\doc\db_drivers\odbc\books.pdf`.

Apama provides two Application Programming Interfaces (APIs) for using the ADBC Connector: the ADBCHelper API and the ADBC Event API.

The ADBCHelper API contains the basic features you need for most common use cases, such as opening and closing databases and executing SQL commands and queries. For more information on the ADBCHelper API, see "The ADBCHelper Application Programming Interface" on page 190.

The ADBC Event API contains features for more complex use cases. For example, in addition to opening and closing databases, it contains actions for discovering what data sources and databases are available. For more information on the ADBC Event API, see "The ADBC Event Application Programming Interface" on page 200.

The ADBC Adapter editor in Apama Studio includes an Event Mapping tab that lets you quickly specify the mapping rules for storing events in existing database tables. Apama Studio generates a service monitor that listens for the events of interest and stores them in the database. This monitor provides a quick and straight forward way of writing event data to a database for general analytical purposes; however, it is not meant to be a fail-safe management system.

The ADBC adapter uses separate thread pools for executing queries and commands and will execute each command and query in its own thread. The thread pools are created with a minimum of four threads but for machines with more than four CPU cores the number of threads will match the number of cores. The adapter log will show the number of threads in the thread pools, for example:

```
  Query and Command threadpools using 4 threads
```

The maximum number of concurrent queries running will match the number of threads in the thread pool. As an example, on a machine with less than four cores, this would be four concurrent queries and four concurrent commands.

Additional queries and commands submitted will be queued for execution until a thread becomes free. If more than four long running queries are submitted, additional queries will be queued. If a mix of short and known long running queries are being used, the application may want to control the submission of long running queries to ensure the shorter duration queries do not have to wait. If the execution of the short duration queries are required to be run without delay, a second adapter can also be started and used to service just the shorter duration queries.

Using the Apama Database Connector

Adding an ADBC adapter to an Apama Studio project

When you add an ADBC adapter to an Apama Studio project, Studio automatically includes all the resources associated with the adapter such as service monitors and configuration files.

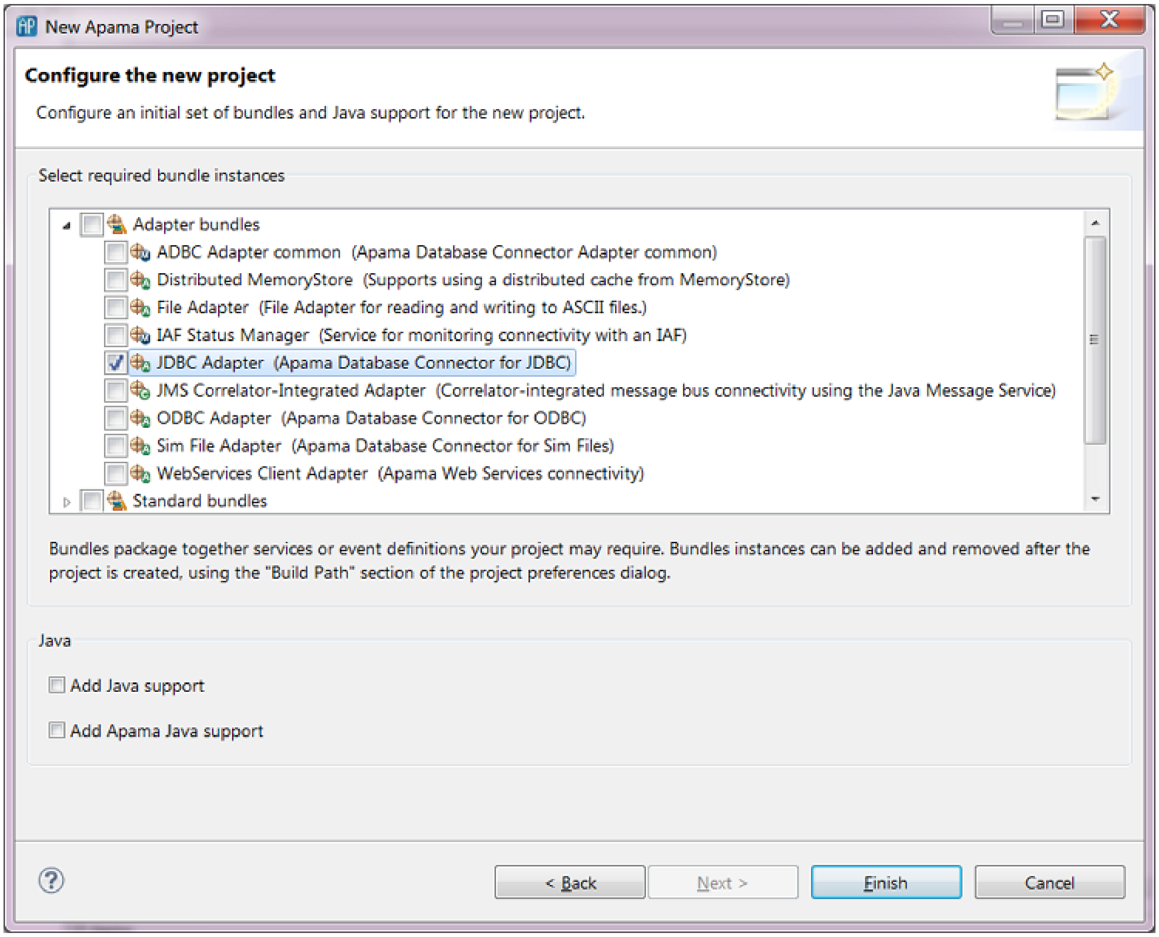
ADBC Adapters are available for three different data sources:

- JDBC Adapter (Apama Database Connector for JDBC)
- ODBC Adapter (Apama Database Connector for ODBC)
- Sim File Adapter for (Apama Database Connector for Sim Files)

To add an adapter to a project:

1. There are two ways of adding an ADBC adapter to a project.

   If you are creating a new Apama project, select File > New > Apama Project , give it a name, and click Next.



Configure the new project

Configure an initial set of bundles and Java support for the new project.

Select required bundle instances

- □ Adapter bundles
  - □ ADBC Adapter common (Apama Database Connector Adapter common)
  - □ Distributed MemoryStore (Supports using a distributed cache from MemoryStore)
  - □ File Adapter (File Adapter for reading and writing to ASCII files.)
  - □ IAF Status Manager (Service for monitoring connectivity with an IAF)
  - ☑ JDBC Adapter (Apama Database Connector for JDBC)
  - □ JMS Correlator-Integrated Adapter (Correlator-integrated message bus connectivity using the Java Message Service)
  - □ ODBC Adapter (Apama Database Connector for ODBC)
  - □ Sim File Adapter (Apama Database Connector for Sim Files)
  - □ WebServices Client Adapter (Apama Web Services connectivity)
- □ Standard bundles

Bundles package together services or event definitions your project may require. Bundles instances can be added and removed after the project is created, using the "Build Path" section of the project preferences dialog.

Java

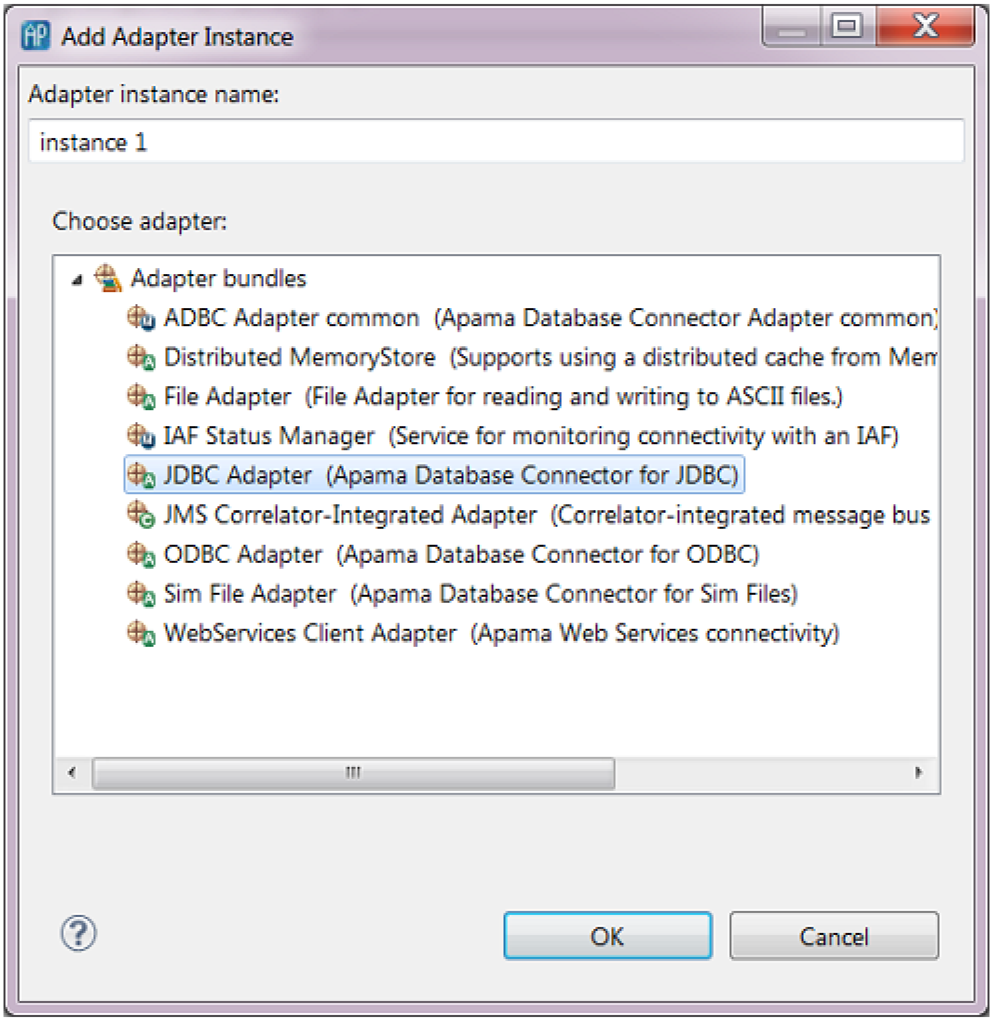
□ Add Java support

□ Add Apama Java support

If you are adding an ADBC adapter to an existing project:

a. In the Project Explorer right-click the project and select Apama > Add Adapter. The **Add Adapter Instance** dialog opens.

b. If desired, in the **Add Adapter Instance** dialog, create a new name for the adapter instance or accept the default instance name. Apama Studio prevents you from using a name that is already in use.



Adapter instance name: instance 1

Choose adapter: JDBC Adapter (Apama Database Connector for JDBC)

Add Adapter Instance
Adapter bundles
ADBC Adapter common (Apama Database Connector Adapter common)
Distributed MemoryStore (Supports using a distributed cache from Mem
File Adapter (File Adapter for reading and writing to ASCII files.)
IAF Status Manager (Service for monitoring connectivity with an IAF)
JDBC Adapter (Apama Database Connector for JDBC)
JMS Correlator-Integrated Adapter (Correlator-integrated message bus
ODBC Adapter (Apama Database Connector for ODBC)
Sim File Adapter (Apama Database Connector for Sim Files)
WebServices Client Adapter (Apama Web Services connectivity)
OK
Cancel

2. In the **New Apama Project** dialog or the **Add Adapter Instance** dialog, select the ADBC adapter bundle that is appropriate to the kind of data source your application will use. Click Finish or OK.

When you add a data source-specific adapter, the ADBC Adapter common (Apama Database Connector Adapter common) bundle will be added to the project automatically.

Using the Apama Database Connector

Configuring the Apama database connector

The Apama Database Connector is an adapter that is instantiated with the Apama Integration Adapter Framework (IAF). The IAF enables Apama applications to connect to sources of messages and events and to consumers of messages and events; with ADBC, these sources and consumers can be databases. Before using the ADBC adapter, you need to supply the correct information in the adapter’s configuration file.

If you develop your Apama application using Apama Studio, the correct configuration files are included in the application’s project file when you add the appropriate ADBC adapter bundle to the project. In order to connect to a database, you need to specify in the adapter's configuration file the properties such as the type and name of data source and the name of the database that the application will use.

If you are not using Apama Studio, you need to manually create the configuration file from the ADBC adapter template file. For more information on creating the configuration file manually, see "Manually editing a configuration file" on page 187.

Using the Apama Database Connector

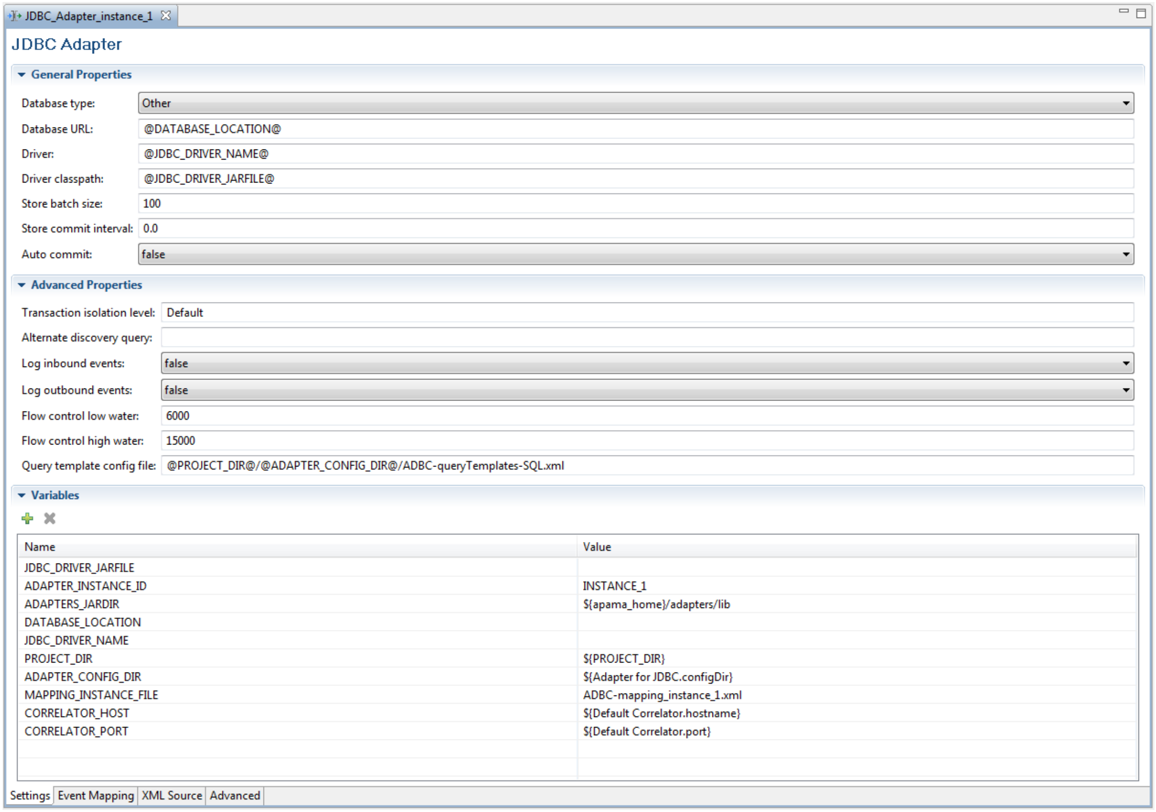
Configuring an ADBC adapter

Apama Studio opens an adapter's configuration file in the adapter editor. By default the file is displayed in the editor's graphical view, which is accessed by clicking the Settings tab. The editor's other tabs are:

- Event Mapping - Displays the Visual Event Mapper where you can quickly map Apama event fields to columns in a database table.
- XML Source - Displays the configuration file's raw XML code.
- Advanced - Provides access to other configuration files associated with the adapter instance. These other files specify, for example, the instance's mapping rules, generated monitors and events responsibe for storing events in a database, and named queries.

To configure an instance of an ADBC adapter:

1. In the Project Explorer, expand the project's `Adapters` node and open the adapter folder (either `Adapter for ODBC`, `Adapter for JDBC`, or `Adapter for Sim`).
2. Double-click the entry for the adapter instance you want to configure. The configuration file opens in the adapter editor. For example, a configuration file for an instance of the ADBC-JDBC adapter looks like this:



General Properties

Database type:	Other
Database URL:	@DATABASE_LOCATION@
Driver:	@JDBC_DRIVER_NAME@
Driver classpath:	@JDBC_DRIVER_JARFILE@
Store batch size:	100
Store commit interval:	0.0
Auto commit:	false

Advanced Properties

Transaction isolation level:	Default
Alternate discovery query:	
Log inbound events:	false
Log outbound events:	false
Flow control low water:	6000
Flow control high water:	15000
Query template config file:	@PROJECT_DIR@/@ADAPTER_CONFIG_DIR@/ADBC-queryTemplates-SQL.xml

Variables

Name	Value
JDBC_DRIVER_JARFILE	
ADAPTER_INSTANCE_ID	INSTANCE_1
ADAPTERS_JARDIR	${apama_home}/adapters/lib
DATABASE_LOCATION	
JDBC_DRIVER_NAME	
PROJECT_DIR	${PROJECT_DIR}
ADAPTER_CONFIG_DIR	${Adapter for JDBC.configDir}
MAPPING_INSTANCE_FILE	ADBC-mapping_instance_1.xml
CORRELATOR_HOST	${Default Correlator.hostname}
CORRELATOR_PORT	${Default Correlator.port}

The Settings tab of the editor's graphical display presents configuration information in three separate sections:

- General Properties
- Advanced Properties
- Variables

For an instance of the ADBC-ODBC adapter, the display is similar but with fewer items in the General Properties and Variables section. For an instance of the ADBC-Sim adapter, the display only shows the Variables section.

3. In the General Properties section, add or edit the following:

   - `Database Type` — This drop-down list allows you to select one of the database types from the list of certified vendors.
   - `Database URL` — This specifies the complete URL of the database. By default it uses the value of the `DATABASE_LOCATION` variable; for more information on this variable see Step 5.
   - `Driver` — For the ADBC-JDBC adapter, this specifies the class name of the vendor's JDBC driver. By default it uses the value of the `JDBC_DRIVER` variable as described in Step 5.
   - `Driver Classpath` — For the ADBC-JDBC adapter, this specifies the classpath for the vendor's driver jar file. By default it uses the value of the `JDBC_DRIVER_JARFILE` variable as described in Step 5.
   - `AutoCommit` — This controls the use of the ODBC/JDBC driver autocommit flag. The default value is false.
   - `StoreBatchSize` — This defines the number of events (rows) to persist using the ODBC/JDBC batch insert API. The use of this setting will significantly increase store performance, but it is not supported by all drivers. A value of 100 is appropriate and will provide good performance in most cases.

     If store performance is critical, testing is required to find the optimal value for the data and driver being used. The default is 0 which disables the use of batch inserts.
   - `StoreCommitInterval`— This defines the interval in seconds before the ADBC adapter will automatically perform a commit for any uncommitted SQL command or store operations. The default value is 0.0 which disables the use of the timed commits.

4. In the Advanced Properties section, add or edit information for the following:

   - `Transaction Isolation Level` — This specifies what data is visible to statements within a transaction. The `Default` level uses the default level defined by the database server vender. To change this setting, enter the appropriate value. For JDBC and ODBC the values can be `READ_UNCOMMITTED`, `READ_COMMITTED`, `REPEATABLE_READ`, or `SERIALIZABLE`.
   - `Alternate Discovery Query` — In most situations an entry here is not required and the ADBC `Discovery` method lists the database available based on the `DATABASE_LOCATION` variable. In some cases you may need to use a server vendor-specific SQL query statement to list the available databases, such as MySQL's `SHOW DATABASES`.
   - `Log Inbound Events` — A boolean that specifies whether or not the application logs inbound ADBC API events with information such as the exact query or command being executed. Logging these events is used for diagnostic purposes and eliminates the need to turn on IAF debug logging. The default is false; do not log incoming events.

- `Log Outbound Events` — The same as `Log Inbound Events` except for outbound ADBC API events.
- `LogCommands` — This property specifies whether or not the starts and completions of commands are written to the IAF log file. A value of "`true`" (the default) logs this information; a value of "`false`" turns logging off. This is useful in cases where logging the start and completion of a high rate of commands (many hundreds or thousands per second) does not add usable information to the log file.
- `LogQueries` — This property behaves identically to the `LogCommands` property except that it specifies whether or not to log the start and completion of queries.
- `FlowControlHighWater` — This defines a maximum threshold for the number of query responses that have not been acknowledged by the ADBC flow control monitor. If this value is reached, the query will be paused until the number of outstanding acknowledgments decreases to the `FlowControLowWater` value. This is used by the ADBC query flow control system to ensure the correlator does not get overwhelmed, especially when performing a fast as possible playback. The default value is 15000.
- `FlowControlLowWater` — This defines a threshold for the number of query responses not acknowledged by the ADBC flow control monitor before a query paused by `FlowControlHighWater` is resumed. This is used by the ADBC query flow control system to ensure the correlator does not get overwhelmed, especially when performing a fast as possible playback. The default value is 6000.
- `Query Template Config File` — This specifies the file containing the query templates that are available to the application. By default, this uses a default template file created for the individual Apama Studio project.

You can add or edit values of the following additional advanced properties by clicking the XML tab and modifying the text of the configuration file:

- `NumericSeparatorLocale` — This allows the numeric separator used in the adapter to be changed, if necessary, to match the one used by the correlator. See "Configuring ADBC localization" on page 188.
- `CloseDatabaseIfDisconnected` — This controls automatic closing of databases whose connection is found to be invalid. See "Configuring ADBC Automatic Database Close" on page 188.
- `FixedSizeInsertBuffers` — This is an ODBC-specific property that allows you to change the default buffer size used when the `StoreData` and `StoreEvent` actions perform batch inserts. Apama uses the `FixedSizeInsertBuffers` property along with the `StoreBatchSize` property to determine how large the insert buffers should be. The value specified by `StoreBatchSize` determines how many rows need to be buffered; the value specified by `FixedSizeInsertBuffers` controls the size of the buffers for the columns. The default "true" uses a fixed buffer size of 10K bytes for each column. If the value is changed to "false", the size of the column buffers is determined dynamically by examining the database table into which the data will be inserted. Allowing the buffer size to be set dynamically can significantly reduce memory usage when performing batch inserts to database tables that contain hundreds of columns or when using a very large `StoreBatchSize`.

5. In the Variables section, add or edit the appropriate values for the following tokens.

- ADAPTERS_JARDIR — For the ADBC-JDBC adapter this specifies the directory where the Apama adapter jar files are located. By default this is the Apama installation's adapters\lib directory.
- DATABASE_LOCATION —The location of the database for use with the ADBC Discovery API, for example, jdbc:mysql://localhost/trades
- PROJECT_DIR — This specifies the location of the Apama project. By default this is automatically set by Apama Studio.
- ADAPTER_CONFIG_DIR — This specifies the location of the adapter's configuration files. By default this is automatically set by Apama Studio.
- CORRELATOR_PORT — This specifies the port used by the correlator. By default this is automatically set by Apama Studio.
- JDBC_DRIVER_NAME — The class name of the driver, such as com.mysql.jdbc.Driver
- JDBC_DRIVER_JARFILE — The name of the data source driver file, for example, C:/Program Files/MySQL/mysql-connector-java-5.1.7/mysql-connector-java-5.1.7-bin.jar
- CORRELATOR_HOST — This specifies the name of the host machine where the project's default correlator runs. By default this is automatically set by Apama Studio.

6. Specify the event mapping rules of the configuration that are specific to your application using the adapter editor's Visual Event Mapper, available on the Event Mapping tab. For more information on specifiying mapping rules, see "The Visual Event Mapper" on page 219.

Configuring the Apama database connector

Manually editing a configuration file

If you are not using Apama Studio to develop your application, you need to manually copy the correct template files to your development environment. The Apama installation provides template files to use as the basis for creating the IAF configuration file to start the ADBC adapter. The templates are located in the adapters\config directory of the Apama installation. The following templates are available:

- ADBC-Sim.xml.dist — Use this configuration file template for accessing a Sim data source.
- ADBC-ODBC.xml.dist — Use this configuration file template for accessing an ODBC data source.
- ADBC-JDBC.xml.dist — Use this configuration file template for accessing a JDBC data source.

To create the configuration file for starting the ADBC adapter:

1. Copy the appropriate template to your project.
2. Edit the name attributes of the various transport properties as necessary.

When you start the IAF with the modified configuration file using the syntax iaf *path_to_modified_config_file*, it automatically includes the appropriate common configuration files shown below.

- ADBC-static.xml — Common event mapping for the ADI adapter events.
- ADBC-static-codecs.xml — The codecs to use (currently null-codec).
- ADBC-application.xml — Application specific event mappings.

- ADBC-namedQuery-Sim.xml — The named query definitions for a Sim data source.

  or
- ADDC-namedQuery-SQL.xml — The named query definitions for ODBC and JDBC data sources.
- ADBC-mapping_*instance_name*.xml — Contains the mappings defined by the user using the Visual Event Mappter.

Configuring an ADBC adapter

Configuring ADBC localization

The ADBC adapter internally handles all string data as UTF-8, and provides the same internationalization support as the correlator. The correlator internally uses the C programming language locale for formatting string versions of numeric values, so there can be conditions under which the ODBC and JDBC drivers may use a locale that is not compatible with the English numeric separator format used in ADBC. In locales that do not use English numeric separators, the ODBC and JDBC drivers for some SQL vendors may not correctly handle numeric values passed from the correlator. To address these cases, the ADBC adapter configuration property NumericSeparatorLocale allows the numeric separator used in the adapter to be changed to match the one used by the correlator. The property can be set to one of three values:

- "" (empty string): Default. Don't change/set separator format.
- C: Set numeric separator format to English.
- Native: Set numeric separator format to system default.

A value of C causes the adapter's numeric separator locale to match that used by the correlator, so that the JDBC and ODBC drivers correctly handle the numeric values. The value Native causes the adapter to set the locale to the system default. This value is not generally needed and was added for future use and for special cases in which technical support would direct it to be set. If you notice incorrect numeric values when inserting or querying data from the database when running in a locale that doesn't use the English-style numeric separators, then changing the NumericSeparatorLocale property to C should correct the problem. In Studio, you can access this property by using the XML tab in the ADBC configuration file editor.

Configuring an ADBC adapter

Configuring ADBC Automatic Database Close

The ADBC adapter performs a connectivity check when a JDBC or ODBC error is encountered, and can be configured to automatically perform the database close operation if a connection is found to be invalid. The IAF status manager will detect the database connection has been closed and report the change in connection status. Applications need to monitor the database connection status in order to take advantage of the automatic closing; this functionality is not integrated into the ADBC APIs.

The ADBC adapter configuration property CloseDatabaseIfDisconnected is used to enable the closing of databases that are detected as invalid.

- False: Default. Don't perform automatic closing.
- True: Close databases detected as invalid (that is, disconnected) .

Configuring an ADBC adapter

Service monitors

If your Apama application uses ADBC, you need to inject several required service monitors. In Apama Studio this is done automatically when you add the appropriate data source adapter bundle to the application's project as described in "Configuring the Apama database connector" on page 183. If you are not using Apama Studio to develop your application, you need to manually inject the following required service monitors in the order they are listed:

- ADBCAdapterEvents.mon — Provides definitions for all events sent to or from the ADBC Adapter.
- ADBCEvents.mon — Provides the public API for ADBC, implemented as actions on the following events:
    - Discovery — This event types defines the actions for discovering ADBC resources. It is used to find the available data sources (ODBC, JDBC, Sim, etc.) and the default databases and query templates configured for those data sources.
    - Connection — This event type defines actions for performing all operations on a database except those involving queries
    - Query — This event type defines actions for performing queries on a database.
- ADBCAdapterService.mon — Provides actions for the following:
    - Forwarding database request events to the adapter.
    - Forwarding database response events to the Java ADBC Service API layer.
    - Supporting parallel execution of blocks and event actions.
- IAFStatusManager.mon
- StatusSupport.mon
- ADBCStatusManager.mon — Manages status subscriptions for the ADBC adapter and the application.
- ADBCHelper.mon — Include this monitor for applications that use the ADBCHelper API.

The ADBC monitors and IAFStatusManager.mon are located in the adapters\monitors directory of the Apama installation. The StatusSupport.mon monitor is located in the Apama installation's monitors directory

Configuring the Apama database connector

ADBC blocks

To facilitate the use of ADBC with scenarios, Apama's Event Modeler includes standard blocks for storing and retrieving data. For more information on these blocks, see "ADBC Storage v10" and "ADBC Retrieval v10" in *Developing Apama Scenarios* (available if you selected Developer during installation).

Configuring the Apama database connector

Codecs

By default, the ADBC adapter uses the standard Apama `NullCodec`. During playback, if your application needs to modify, aggregate or perform analytics on events, you can create and specify IAF codecs to perform these operations instead of using the standard `NullCodec`. For example, capital market applications might convert quote to depth events during playback from a market database. You define the logic for performing this type of conversion in the codec.

For more information on developing codecs, see "C/C++ Codec Plug-in Development" and "Java Codec Plug-in Development" in *Developing Adapters* (available if you selected Developer during installation).

Configuring the Apama database connector

The ADBCHelper Application Programming Interface

The ADBCHelper Application Programming Interface (API) is a simplified, streamlined API for communicating with databases. In most common use cases, this API is the appropriate way to develop applications. For applications that require more complex ways of accessing databases, see "The ADBC Event Application Programming Interface" on page 200.

Using the Apama Database Connector

ADBCHelper API Overview

The ADBCHelper API is defined in the file *`apama_dir`*`\adapters\monitors\ADBCHelper.mon`. The API is implemented with the following events:

- `com.apama.database.DBUtil`
- `com.apama.database.DBAcknowledge`

The `DBUtil` event defines the actions that Apama applications call in order to interact with databases. The `DBAcknowledge` event is used by the ADBCHelper API to specify the success or failure for database actions that request an acknowledgement. Note if you specify the following lines in your code, you do not need to use the fully qualified name for `DBUtil` or `DBAcknowledge`.

```
using com.apama.database.DBUtil;
using com.apama.database.DBAcknowledge;
```

The basic steps for using the ADBCHelper API are:

1. Create an instance for the DBUtil event in your application code, for example:

   ```
   com.apama.database.DBUtil db;
   ```

2. Call the DBUtil `setAdapterInstanceName` action to identify the adapter instance. This step is only required if the adapter instance name is not the default, `INSTANCE_1`. This action is necessary, for example, if the project uses multiple adapter instances.

   For more information, see "Specifying the adapter instance" on page 194.

3. Check whether the database is already open or is in the process of being opened. This step is optional, but it is good programming practice to check for these situations before calling an open event action by calling

the DBUtil `isOpen` action. This returns a boolean that specifies if the database is already open or in the process of being opened.

For more information, see "Checking to see if a database is open" on page 194.

4. Call one of the `DBUtil` open actions to open the database.

   For more information on open actions, see "Opening databases" on page 192.

5. Call one or more `DBUtil` event actions, depending on the database task you want to implement:

   - Call a SQL query event action to retrieve data from the database, in either a result set or in Apama event format.

     For more information on query actions, see "Issuing and stopping SQL queries" on page 194.

   - Call a SQL command event action to add, update, or delete data in the database.

     For more information on SQL command actions,, see "Issuing SQL commands" on page 195.

   - Optionally, if the `autoCommit` setting has been turned off, call a commit event action to commit database changes, or call a rollback event action to rollback uncommitted changes.

     For more information on commit actions see "Committing database changes" on page 195.

     For more information on rollback actions, see "Performing rollback operations" on page 196.

6. Create actions as required to handle returned result sets. If the query returns events, create listeners for events returned by the query.

   For more information on handling query results, see "Handling query results for row data" on page 196.

7. For action calls that request an acknowledgement, your application needs to do the following:

   a. Create an instance of the `com.apama.database.DBAcknowledge` event.

      Note, if your code contains the line `using com.apama.database.DBAcknowledge;`, you do not need to use the fully qualified name for `DBAcknowledge`.

   b. Create a listener for the `DBAcknowledge` event that indicates when the `DBUtil` event action call is complete.

   For more information on handling acknowledgments, see "Handling acknowledgments" on page 197.

8. Create an action that handles errors that could occur during execution of a `DBUtil` event action call.

   For more information on handling errors, see "Handling errors" on page 198.

9. Call the `DBUtil` event's close action to close the database.

   For information on closing databases, see "Closing databases" on page 199.

The ADBCHelper Application Programming Interface

Opening databases

The ADBCHelper API provides several actions for opening databases. The "quick" open actions allow you to open JDBC and ODBC databases by passing in a minimal set of parameters, while the "full" open action provides more control by passing in a complete set of parameters. The "shared" open action allows you to use an already open existing matching connection or open a new connection if a matching one does not exist..

In the following quick open actions, you need to pass in values for the following parameters:

- `URL` — database connection string
- `user` — user name
- `password` — user password
- `handleError` — name of a default error handler

  See "Handling errors" on page 198 for more information on creating actions to handle errors.

The quick open actions use the default settings for the `autoCommit` (`true`), `batchSize` (`100`), and `timeOut` (`30.0`) properties.

```
action openQuickJDBC(
    string URL,
    string user,
    string password,
    action < string > handleError )
action openQuickODBC(
    string URL,
    string user,
    string password,
    action < string > handleError )
```

The following code snippet shows a use of the `openQuickJDBC` action.

```
 com.apama.database.DBUtil db;
   action onload {
     string dbUrl:= "jdbc:mysql://127.0.0.1:3306/exampledb";
     string user := "thomas";
     string password := "thomas-123";
     db.openQuickJDBC(dbUrl, user, password, handleError );
     // ...
```

For the following open action you need to pass in all parameters.

```
action open(
    string type,
    string serviceId,
    string URL,
    string user,
    string password,
    string autoCommit,
    boolean readOnly,
    integer batchSize,
    float timeOut,
    action < string > handleError )
```

Setting the `autoCommit`, `batchSize`, and `timeOut` parameters in the `open` action over-rides the adapter properties specified in the IAF configuration file.

- `type` — The data source type (ODBC, JDBC, Sim, etc.)
- `serviceId` — The service id for the adapter

- URL — The database connection string
- user — The user name
- password — The user password
- autoCommit — The auto commit mode to use. If this parameter is not set, the open action uses a combination of the AutoCommit and StoreCommitInterval properties specified in the adapter's configuration file. For information on these properties, see "Configuring an ADBC adapter" on page 184. The value for the autoCommit parameter can be one of the following modes:
  - "" — An empty string specifies that the value set in the configuration file should be used.
  - true — Enables the ODBC/JDBC driver's auto commit.
  - false — Disable autoCommit.
  - *x.x* — Use timed auto commit interval in seconds.
- readOnly — Specifies if the connection should be read-only. If the connection is read-only an error will be reported for any API action that requires writes (Store, Commit, or Rollback). Most databases do not prevent writes from a connection in read-only mode so it is still possible to perform writes using the Command actions.
- batchSize — The query results batch size to be used for any queries performed.
- timeOut — Controls how long the ADBC open action will wait for the adapter to become available if it is not running when the open action is called.

The following code snippet shows a use of the open action. It creates variables for each of the parameters and passes them with the openaction.

```
 com.apama.database.DBUtil db;
   action onload {
       string type := "jdbc";
       string serviceId := "com.apama.adbc.JDBC";
       string dbUrl:= "jdbc:mysql://127.0.0.1:3306/exampledb";
       string user := "thomas";
       string password := "thomas-123";
       string commit := "15.0";
       boolean readMode := false;
       float openTimeout := 30.0;
       Integer queryBatchSize := 100
       db.open(type, serviceId, dbUrl, user, password, commit,
          readMode, openTimeout, queryBatchSize, handleError );
        // ...
```

The following open action allows you to use a connection that is already open; the action opens a connection if an existing matching connection is not found. The openShared action uses the same parameters as the open action, above.

```
action openShared(
    string type,
    string serviceId,
    string URL,
    string user,
    string password,
    string autoCommit,
    boolean readOnly,
    integer batchSize,
    float timeOut,
    action < string > errorHandler )
```

The ADBCHelper Application Programming Interface

Specifying the adapter instance

By default ADBCHelper actions use the default name given to the adapter instance, which is `INSTANCE_1`. If the adapter instance name is different from the default, for example if your Apama project has multiple adapter instances, you need to specify the name of the adapter instance you want to use. Use the DBUtil `setAdapterInstanceName` action to do this.

To specify an adapter instance:

Call the `setAdapterInstanceName` action, passing in the adapter instance name variable.

For example:

```
com.apama.database.DBUtil db;
  action onload {
    string adapterInstanceName := "EXAMPLE_ADBC_INSTANCE";
    db.setAdapterInstanceName(adapterInstanceName);
    // ...
```

The ADBCHelper Application Programming Interface

Checking to see if a database is open

Checking to see whether the database is already open or is in the process of being opened before calling an open event action is optional, but it is good programming practice. An application may also want to check if a database is open before executing a query.

The following example checks these fields to ensure that the application does not try to open an already opened database.

```
 com.apama.database.DBUtil db;
 //...
if not db.isOpen then {
     db.openQuickODBC(dbUrl,””,””,handleError );
                }
```

The ADBCHelper Application Programming Interface

Issuing and stopping SQL queries

The following actions execute SQL queries. The actions expect a response and a `handleResult` action needs to be defined to handle each row returned.

```
action doSQLQuery(
    string queryString,
    action< dictionary< string, string > > handleResult )
action doSQLQueryOnError(
    string queryString,
    action< dictionary< string, string > > handleResult)
action doSQLQueryAck(
    string queryString,
    action < dictionary< string, string > > handleResult,
    integer ackNum,
    boolean onError )
```

The following query action allows you to specify a callback action for when the query completes. The parameters are (1) the query string, (2) the handler action for each row returned and (3) the handler for when the query completes. The handler for when the query completes has two parameters, an error string and an integer that specifies the number of rows returned by the query.

```
action doSQLQueryWithCallback(
    string queryString,
    action< dictionary< string, string > > handleResult,
    action < string, integer> handleDone )
```

The following actions are similar to the above query actions except they return Apama events instead of results sets.

```
action doSQLEventQuery(
    string queryString,
    string eventType )
action doSQLEventQueryWithCallback(
    string queryString,
    string eventType,
    action < string, integer> handleDone )
action doSQLEventQueryOnError(
    string queryString,
    string eventType )
action doSQLEventQueryAck(
    string queryString,
    string eventType,
    integer ackNum,
    boolean onError )
```

The following action cancels all outstanding queries in the queue.

```
action stopAll()
```

For more information on creating a `handleResult` action, see "Handling query results for row data" on page 196.

The ADBCHelper Application Programming Interface

Issuing SQL commands

The following actions execute SQL commands and expect no responses.

```
action doSQLCmd( string queryString )
action doSQLCmdAck(
    string queryString,
    integer ackNum,
    boolean onError )
action doSQLCmdOnError( string queryString )
```

The `doSQLCmdOnError` action executes only if a previous non- `*OnError` operation failed. This is useful for doing, for example, a `select * from table` command and then, if an error occurs, execute a `create table ...` command.

The ADBCHelper Application Programming Interface

Committing database changes

The default auto-commit behavior is driver auto commit, assuming the `AutoCommit` and `StoreCommitInterval` properties specified in the adapter's configuration file and the open action are using the default values. If you want more control over when changes are committed to a database, set the `open` action's `autoCommit` parameter to false and in your EPL code, manually commit data using one of the following `DBUtil` event actions in your EPL code:

```
action doSQLCommit()
action doSQLCommitAck( integer ackNum )
```

The ADBCHelper Application Programming Interface

Performing rollback operations

For rolling back uncommitted changes to database, use the following `DBUtil` actions. If you want to use rollback actions, you need to turn autocommit off.

```
action doSQLRollback()
```

For rolling back uncommitted changes to database in situations where the previous `SQLCmd`, `SQLQuery`, or `SQLCommit` operation failed, use:

```
action doSQLRollbackOnError()
```

When you want to rollback uncommitted changes to the database and receive a `DBAcknowledge` event to indicate success or failure, use:

```
action doSQLRollbackAck( integer ackNum, boolean onError )
```

The *`ackNum`* parameter is the identifier for the `DBAcknowledge` event; setting it to `-1` will disable sending the `DBAcknowledge` event and instead use the default error handler if an error occurs. For the *`onError`* parameter, setting its value to true will cause the operation to run only if the previous `SQLCmd`, `SQLQuery`, or `SQLCommit` failed.

The ADBCHelper Application Programming Interface

Handling query results for row data

For query actions that return a result set of rows of data, your application needs to define actions to handle result sets. For example:

```
com.apama.database.DBUtil db;
action onload {
   db.openQuickODBC("exampledb","thomas","thomas123",handleError);
   db.doSQLQuery("SELECT * FROM  NetworkInformation", handleNetworkInfo);
   // ...
}
action handleNetworkInfo( dictionary< string, string > data ) {
   log "Network: " + data[ "network" ] + " CountryCode: " +
   data[ "countrycode" ] + " NIC: " +
   data[ "networkidentificationcode" ] at INFO;
}
```

The ADBCHelper Application Programming Interface

Handling query results for event data

For query actions that return a result set in the form of events, your application needs to do the following.

1. Define an event type that represents the returned data.
2. Map the returned data to fields in the event type. The easiest way to do this is to use the Apama Studio's Visual Mapper, which automatically saves the mapping information in a project file. For more information on using the Visual Mapper, see "Using the Visual Event Mapper" on page 220.
3. Create a listener for the event type.
4. Execute a query that returns events.

For example, the following EPL code snippet defines an event, executes a query that returns data in the form of the defined event, and defines a listener for the defined event:

```
event NetworkInfo {
  string network;
  integer countrycode;
  integer nid;
  }
//...
com.apama.database.DBUtil db;
  action onload {
    db.openQuickODBC("exampledb","thomas","thomas123",handleError);
    db.doSQLEventQuery("SELECT * FROM network_info", NetworkInfo);
    //...
  }
  on all NetworkInfo := netInfo {
    // Code to do something with the returned event...
  }
```

The ADBCHelper Application Programming Interface

Handling acknowledgments

Apama applications can call DBUtil SQL command and query actions as well as commit and rollback actions that request a DBAcknowledgement event. The DBAcknowledgement event indicates the success or failure of the action call. This is useful, for example, to know whether or not a query has completed before performing another application operation.

The DBAcknowledgement event is defined in apama_install_dir\adapters\monitors\ADBCHelper.mon as follows:

```
event DBAcknowledge
{
 integer ackNum;
 boolean success;
 string  error;
}
```

- ackNum — A unique identifier for the action that requested the acknowledgment.
- success — A value of true indicates success; false indicates failure.
- error — A string describing the specific error.

For action calls that request an acknowledgement, your application needs to do the following:

1. Call an action that requests an acknowledgement, passing in a unique acknowledgment identifier.
2. Create an instance of the com.apama.database.DBAcknowledge event.
3. Create a listener for theDBAcknowledge event that matches the acknowledgment identifier in the calling action.

For example:

```
integer ackId := integer.getUnique();
db.doSQLQueryAck("SELECT * FROM NetworkInformation", handleNetworkInfo,ackId,false);
com.apama.database.DBAcknowledge ack;
//...
  on DBAcknowledge(ackNum = ackId) : ack {
     if ack.success then {
        log "Query complete" at INFO;
     }
     else {
        log "Query failed: " + ack.error at ERROR;
        die;
    }
  }
//...
```

The ADBCHelper Application Programming Interface

Handling errors

The `DBUtil` actions require a user-defined `handleError` action that takes a single `string` parameter. The `handleError` action handles errors that could occur during execution of a `DBUtil` event action call.

The following EPL code snippet shows a simple error handler.

```
//...
  com.apama.database.DBUtil db;
//...
  action onload {
    db.openQuickODBC("exampledb","thomas","thomas123", handleError);
                //...
      }
  action handleError( string reason ) {
    log "DB Error: " + reason;
  }
```

The ADBCHelper Application Programming Interface

Reconnection settings

Apama applications can automatically reconnect if a disconnection error is encountered. The reconnection capability is optional and the default is to not reconnect when a disconnection error occurs. The following reconnection actions are defined in the `com.apama.database.DBUtil` event.

- `action setReconnectPolicy(string reconnectPolicy)` — This action sets the policy for dealing with adapter connection errors. The *`reconnectPolicy`* parameter must be one of the constants specified in the `DBReconnectPolicy` event. The policy constants are as follows:

  - `RECONNECT` — Try to reconnect and leave the management of pending requests to the client, which will handle the pending requests in the error handler.
  - `RECONNECT_AND_RETRY_LAST_REQUEST` — Try to reconnect and leave the pending requests unchanged, retry the last request on a successful database reconnection.
  - `RECONNECT_AND_CLEAR_REQUEST_QUEUE` — Try to reconnect and remove all the pending requests.
  - `DO_NOT_RECONNECT` — Do not try to reconnect.

  The default reconnect policy is `DO_NOT_RECONNECT`.

- `action setReconnectTimeout(float timeOut)` — This action sets the timeout for the reconnection after a connection error. A value specified by the `setReconnectTimeout` action overrides the default timeout value, which is equal to twice as long as specified by the open action's *`timeOut`* parameter.

The ADBCHelper Application Programming Interface

Closing databases

The following action closes the database. If `doStopAll` is set it also cancels all outstanding queries and commands in the queue and prevents new queries and commands from being placed into the queue.

```
action close( boolean doStopAll )
```

The ADBCHelper Application Programming Interface

Getting schema information

The following actions return information about a table in a database. The actions are only valid in the `handleResult` action specified in a `doSQLQuery`, `doSQLQueryOnError`, or `doSQLQueryAck` operation when dealing with a returned row.

```
action getSchemaFieldOrder()  returns sequence< string >
action getSchemaFieldTypes()  returns dictionary< string, string >
action getSchemaIndexFields() returns sequence< string >
```

The ADBCHelper Application Programming Interface

Setting context

By default the ADBCHelper API sends requests to an internal service monitor running in the main context with the EPL `route` command. However, if your application uses parallel processing and spawns to multiple contexts, you have to add code that identifies the main context so the ADBCHelper API can determine whether to send an event with `route` or with `enqueue`.

In applications with multiple contexts, use the following action to specify the main context before spawning.

```
setPrespawnContext( context c )
```

The ADBCHelper Application Programming Interface

Logging

This action specifies whether or not to log all SQL queries, commands, and commit operations to the correlator's log file.

```
action setLogQueries( boolean logQueries )
```

The default is `false`, which disables logging.

The ADBCHelper Application Programming Interface

Examples

The code listings in this section are adapted from the `api-helper-example.mon` application. The actual code can be found in the Apama installation's `samples\adbc\api-helper-example` directory.

Opening and closing a database and executing SQL commands

```
monitor ADBCHelper_Example
{
  com.apama.database.DBUtil db;
  action onload {
    db.openQuickODBC( "MySQL", "fred", "fred-123", handleError );
    db.doSQLCmd( "insert into NetworkInformation values (
        'Vodafone', 'FR', 104 );");
    db.doSQLCmd( "insert into NetworkInformation values (
        'O2', 'FR', 101 );");
    db.doSQLCmd( "insert into NetworkInformation values (
        'Three', 'FR', 102 );");
    db.doSQLCmdAck( "insert into NetworkInformation values (
        'Orange', 'FR', 103 );", 100, false);
    com.apama.database.DBAcknowledge ack;
    on com.apama.database.DBAcknowledge(ackNum = 100) : ack {
      if ack.error.length() = 0 then {
         log "Action complete" at INFO;
         // Other success handling code ...
      }
      else {
         log "Action failed: " + ack.error at ERROR;
         // Other failure handling code ...
      }
    db.close( false);
  }
  action handleError( string reason ) {
    log "DB Error: " + reason at ERROR;
  }
}
```

Executing SQL queries

```
monitor ADBCHelper_Example
{
  com.apama.database.DBUtil db;
  action onload {
    db.openQuickODBC( "DBName", "user", "password", handleError );
    db.doSQLQuery(
        "SELECT * FROM NetworkInformation", handleResult );
    db.close( false);
  }
  action handleResult( dictionary< string, string > data ) {
    log "Network: " + data[ "network" ] +
    " CountryCode: " + data[ "countrycode" ] +
    " NIC: " + data[ "networkidentificationcode" ] at INFO;
  }
  action handleError( string reason ) {
    log "DB Error: " + reason at ERROR;
  }
}
```

The ADBCHelper Application Programming Interface

The ADBC Event Application Programming Interface

The Apama Database Connector Event Programming Interface (API) provides operations for more complex, lower level interactions with databases than the operations included with the ADBCHelper API. The ADBC Event API is implemented with the following Apama event types and actions associated with those events.

- `Discovery` — This event type provides actions to obtain the names of data sources, databases, and named queries. Discovery actions are not necessary if your application knows the names of data sources, databases, and query templates.
- `Connection` — This event type provides actions for all operations on a database except for those involving queries.

- Query — This event type provides actions for creating and executing queries on databases.
- PreparedQuery — This event type provides actions for creating prepared query statements that are, in turn, used in queries.

The above events and associated actions are defined in the ADBCEvents.mon file.

In addition, some of the actions for Discovery events use the following event types, which are defined in the ADBCAdapterEvents.mon file.

- DataSource
- Database
- QueryTemplate

Using the Apama Database Connector

Discovering data sources

If your application needs to find available data sources, implement the following steps:

1. Create a new Discovery event.
2. Use the Discovery event's findAvailableServers action.
3. Create a handler action to perform callback actions on the results of the findAvailableServers action.
4. In the handler action, declare a variable for a DataSource event.

   The definitions for the two forms of the findAvailableServers action are:

   ```
   action findAvailableServers(
     float timeout,
     action < string, sequence<DataSource> > callback )
   ```

   and

   ```
   action findAvailableServersFull(
     float timeout,
     dictionary<string,string> extraParams,
     action < string, sequence<DataSource>  > callback )
   ```

   The definition of the DataSource event is:

   ```
   event DataSource
   {  string serviceId;
     string name;
     dictionary<string,string> extraParams;
   }
   ```

   - serviceID — The serviceID to talk to this DataSource.
   - name — The name of the DataSource such as ODBC, JDBC, or Sim.
   - extraParams — Optional parameters.

   The relevant code in the samples\adbc\api-example\ADBC_Example.mon file is similar to this:

   ```
   com.apama.database.Discovery adbc :=
       new com.apama.database.Discovery;
   adbc.findAvailableServers(TIME_TO_WAIT, handleAvailableServers);
   action handleAvailableServers(string error,
       sequence<com.apama.database.DataSource> results)
     {
   ```

```
        if error.length() != 0 then {
          log "Error occurred getting available data sources: " +
               error at ERROR;
        }
        else {
          if results.size() > 0 then {

            // Save off first service ID found.
            // Assumes first data source has at least one db
            if getDbServiceId() = "" then {
               dbServiceId := results[0].serviceId;
            }
            com.apama.database.DataSource ds;
            log "  DataSources: " at INFO;
            for ds in results {
                log "    " + ds.name + " - " + ds.serviceId at INFO;
            }
            log "Finding Databases ..." at INFO;
            // ... other logic ...
          }
          else {
            log "  No DataSources found" at INFO;
          }
        }
      }
```

The ADBC Event Application Programming Interface

Discovering databases

If your application needs to find available databases, implement the following steps:

1. Given a `Datasource` event, call the event's `getDatabases` action.
2. Create a handler action to perform callback actions on the results of the `getDatabases` action.
3. In the handler action, declare a variable for a `Database` event.

The definitions for the two forms of the `getDatabases` action are:

```
action getDatabases(
  string serviceId,
  string user,
  string password,
  action< string, sequence<Database> > callback)
```

and

```
action getDatabasesFull(
  string serviceId,
  string locationURL,
  string user,
  string password,
  dictionary<string,string> extraParams,
  action < string, sequence<Database> > callback)
```

Note: JDBC data sources will usually require user and password values.

The definition of the `Database` event is:

```
event Database
{
  string shortName;
  string dbUrl;
  string description;
  dictionary<string,string> extraParams;
}
```

- `shortName` — A short display name
- `dbUrl` — The complete URL of the database, for example, `jdbc:sqlserver://localhost/ApamaTest`.
- `extraParams` — Optional parameters.

The relevant code in the `samples\adbc\api-example\ADBC_Example.mon` file is similar to this:

```
action handleAvailableServers(string error,
    sequence<com.apama.database.DataSource> results)
  {
    if error.length() != 0 then {
      log "Error occurred getting available data sources: " +
           error at ERROR;
    }
    else {
      if results.size() > 0 then {

        // Save off first service ID found.
        // Assumes first data source has at least one db
        if getDbServiceId() = "" then {
          dbServiceId := results[0].serviceId;
        }
        com.apama.database.DataSource ds;
        log "  DataSources: " at INFO;
        for ds in results {
          log "    " + ds.name + " - " + ds.serviceId at INFO;
        }
        log "Finding Databases ..." at INFO;
        for ds in results {
          adbc.getDatabases(ds.serviceId, USER, PASSWORD,
             handleAvailableDatabases);
        }
      }
      else {
        log "  No DataSources found" at INFO;
      }
    }
  }
string dbName;
action handleAvailableDatabases(string error,
    sequence<com.apama.database.Database> results)
  {
    if error.length() != 0 then {
      log "Error occurred getting available databases: " +
           error at ERROR;
    }
    else {
      if results.size() > 0 then {
        // Save name of first db found
        if getDbName() = "" then {
          dbName := results[0].shortName;
        }
        com.apama.database.Database db;
        log "  Databases: ";
        for db in results {
          log "    " + db.shortName + " - " +
              db.description + " - " + db.dbUrl at INFO;
        }
        // ... other logic...
      }
      else {
        log "  No Databases found" at INFO;
      }
    }
  }
```

The ADBC Event Application Programming Interface

Opening a database

In order to open a database, your application should implement the following steps:

1. Create a new `Connection` event.
2. Call the `Connection` event's `openDatabase` action with the database's serviceID, database URL, autocommit preference, and the name of the callback action.
3. Create the handler action for the `openDatabase` callback action.

The definitions for the different forms of the `openDatabase` actions are:

```
action openDatabase(
  string serviceId,
  string databaseURL,
  string user,
  string password,
  string autoCommit,
  action <Connection, string> callback)
```

and

```
action openDatabaseFull (
  string serviceId,
  string databaseURL,
  string user,
  string password,
  string autocommit,
  boolean readOnly,
  dictionary<string,string> extraParams,
  action <Connection, string> callback)
```

In addition to these open actions you can also open a database using an already open matching connection if one exists using the `openDatabaseShared` action. If an existing connection is not found, the action opens a new connection.

```
action openDatabaseShared (
  string serviceId,
  string databaseURL,
  string user,
  string password,
  string autocommit,
  boolean readOnly,
  dictionary<string,string> extraParams,
  action <Connection, string> callback)
```

The value for the `autocommit` parameter is a combination of the `AutoCommit` and `StoreCommitInterval` properties. For information on these properties, see "Configuring an ADBC adapter" on page 184. The value for the `autocommit` parameter can be one of the following modes:

- `""` — An empty string specifies that the value set in the configuration file should be used.
- `true` — Use the data source's value as determined by the ODBC or JDBC driver.
- `false` — Disable `autocommit`.
- *x.x* — Use time auto commit interval in seconds.

The `readOnly` parameter specifies if the connection should be read-only. If the connection is read-only an error will be reported for any API action that requires writes (`Store`, `Commit`, or `Rollback`). Most databases do not prevent writes from a connection in read-only mode so it is still possible to perform writes using the `Command` actions.

Specifying parameter values in the open actions overrides the property values set in the configuration file.

The relevant code in the `samples\adbc\api-example\ADBC_Example.mon` file is similar to this:

```
com.apama.database.Connection conn :=
    new com.apama.database.Connection;
action handleAvailableDatabases(string error,
    sequence<com.apama.database.Database> results)
  {
    if error.length() != 0 then {
      log "Error occurred getting available databases: " +
           error at ERROR;
    }
    else {
      if results.size() > 0 then {
        // Save name of first db found
        if getDbName() = "" then {
          dbName := results[0].shortName;
        }
        com.apama.database.Database db;
        log "  Databases: " at INFO;
        for db in results {
          log "    " + db.shortName + " - " +
              db.description + " - " + db.dbURL at INFO;
        }
        log "Opening Database " +  dbName + " ..." at INFO;
        string serviceId := getDbServiceId();
        conn.openDatabase(serviceId, results[0].dbUrl, USER,
               PASSWORD, "", handleOpenDatabase);
        }
        else {
          log "  No Databases found" at INFO;
        }
     }
  }
```

The ADBC Event Application Programming Interface

Closing a database

In order to close a database your application should implement the following steps:

1. Call the `closeDatabase` action of the `Connection` event (for the open database) with the name of the callback action.
2. Create a handler action for the `closeDatabase` callback action.

The definitions for the two forms of the `closeDatabase` action are:

```
action closeDatabase(
  action <Connection, string> callback)
```

and

```
action closeDatabaseFull(
  boolean force,
  dictionary<string,string> extraParams,
  action<Connection,string> callback)
```

The relevant code in the `samples\adbc\api-example\ADBC_Example.mon` file is similar to this:

```
com.apama.database.Connection conn :=
new com.apama.database.Connection;
// ...
```

```
    conn.openDatabase(serviceId, results[0].dbUrl, "",
            handleOpenDatabase);
// ...
    conn.closeDatabase(handleCloseDatabase);
action handleCloseDatabase(com.apama.database.Connection conn,
    string error)
  {
    if error.length() != 0 then {
      log "Error closing database " + getDbName() + ": " +
           error at ERROR;
    }
    else {
      log "Database " + getDbName() + " closed." at INFO;
    }
  }
```

The ADBC Event Application Programming Interface

Storing event data

In order to store an event in a database, your application needs to use the `Connection` event's `storeEvent` action. The definition of the `storeEvent` action is:

```
action storeEvent(
  float timestamp,
  string eventString,
  string tableName,
  string statementName,
  string timeColumn,
  dictionary<string,string> extraParams) returns integer
```

The `getTime()` call on the event is used to set the `timestamp` value.

Similarly, the `toString()` call on an event sets the `eventString` field.

The `tableName` parameter specifies the name of the database table where you want to store the data.

The `statementName` parameter specifies the name of a `storeStatement` that references a prepared statement or stored procedure. The `storeStatement` is created with the `Connection` event's `createStoreStatement` action. See "Creating and deleting store events" on page 208 for more information on creating a `storeStatement`. If you do not want to specify a prepared statement or stored procedure, the `statementName` parameter should be set to `""` (an empty string).

The `timeColumn` parameter specifies the column in the database where you want the event timestamp to be stored.

The `storeEvent` action returns an integer value, which is the identifier for the event being stored. The `setStoreErrorCallback` action is used to specify an action to be used when an error is reported.

To store an event and provide acknowledgement, implement the `storeEventWithAck` action and a callback handler. The definition of the `storeEventWithAck` action is:

```
action storeEventWithAck(
  float timestamp,
  string eventString,
  string tableName,
  string statementName,
  string timeColumn,
  string token,
  dictionary<string,string> extraParams,
  action <Connection, string, string> callback)
```

In addtion to the parameters used with the `storeEvent` action, the `storeEventWithAck` action includes `token` and `callback` parameters. The `token` parameter specifies a user-defined string to be passed in that will be returned in the callback action. This allows the callback to perform different operations depending on the token value. In this way, a single callback action can perform different operations, eliminating the need to create separate callbacks for each operation. If the `token` parameter is not needed for the callback, it should be set to `""` (an empty string).

The `callback` parameter specifies the callback action that handles the success or failure of the `storeEventWithAck` action.

If you want to avoid the overhead of receiving acknowledgements each time event data is added to a database table, use the `storeEvent` action. If your application needs to handle a failure during a call to the `storeEvent` action, it should call the `setStoreErrorCallback` action; for more information, see "Handling data storing errors" on page 209.

The ADBC Event Application Programming Interface

Storing non-event data

In order to store non-event data in a database, your application needs to use the `Connection` event’s `storeData` action. The definition of the `storeData` action is:

```
action storeData(
  string tableName,
  string statementName,
  dictionary<string,string> fields,
  dictionary<string,string> extraParams) returns integer
```

The `tableName` parameter specifies the name of the database table where you want to store the data.

The `statementName` parameter specifies the name of a `StoreStatement` that references a prepared statement or stored procedure. The `storeStatement` is created with the `Connection` event's `createStoreStatement` action. See "Creating and deleting store events" on page 208 for more information on creating a `storeStatement`. If you do not want to specify a prepared statement or stored procedure, the `statementName` parameter should be set to `""` (an empty string).

The `fields` parameter specifies the column values to be stored.

To store an event and provide acknowledgement, implement the `storeDataWithAck` action and a callback handler. The definition of the `storeDataWithAck` action is:

```
action storeDataWithAck(
  string tableName,
  string statementName,
  dictionary<string,string> fields,
  string token,
  dictionary<string,string> extraParams,
  action <Connection, string, string> callback)
```

In addtion to the parameters used with the `storeData` action, the `storeDataWithAck` action includes `token` and `callback` parameters. The `token` parameter specifies a user-defined string to be passed in that will be returned in the callback action. This allows the callback to perform different operations defending on the token value. In this way, a single callback action can perform different operations, eliminating the need to create separate callbacks for each operation. If the `token` parameter is not needed for the callback, it should be set to `""` (an empty string).

The `callback` parameter specifies the callback action that handles the success or failure of the `storeDataWithAck` action. The acknowledgement `callback` string contains any errors reported as well as the returned `token`, an empty acknowledgement string indicates success.

If you do not have to take additional action each time a row of data is added to a database table, you can avoid the overhead of receiving acknowledgements by using the `storeData` action. If your application needs to handle a failure during a call to the `storeData` action, it should call the setStoreErrorCallback action; for more information, see "Handling data storing errors" on page 209.

The ADBC Event Application Programming Interface

Creating and deleting store events

If your application will use a prepared statement or a stored procedure in a store action (such as storeData or storeEvent) you need to first create a storeStatement with createStoreStatement action.

The createStoreStatement is defined as:

```
action createStoreStatement(
    string name,
    string tableName,
    string statementString,
    sequence<string> inputTypes,
    dictionary<integer,string> inputToNameMap,
    dictionary<string,string> extraParams,
    action<Connection,string,string> callback)
```

The arguments for this action are:

- name - The name of the storeStatement instance that will be used in a store action. The name must be unique. Specifying a value for name is optional and if omitted, one will be created in the form Statement_1.
- tableName - The name of the database table where the data will be written when the store action that uses the storeStatment is called.
- statementString - The SQL string that will be used as a template when the store action that uses the storeStatement is called. You can use question mark characters to indicate replacable parameters in the statement. For example, "insert into myTable(?,?,?) values(?,?,?)".

  If you want to use a stored procedure, in the statementString enclose the name of the database's stored procedure in curly brace characters {} and use question mark characters ? to indicate replaceable parameters. For example, "{call myStoredProcedure(?,?,?)}". Stored procedures used in this way can only take input parameters. The stored procedure must exist in the database.
- inputTypes - Specifies the types that will be used as replacable parameters in the statementString.
- inputToNameMap - Specifies what data item should be used for each input parameter of the store statement. If storing data it would be the name from the dictionary of data to be stored. If storing events it would be the event field name. When you specify the dictionary, the integer is the position and the string is the data name. For example, you might specify the inputToNameMap parameter as follows:

  ```
  inputToNameMap :=
        {1:"timefield",2:"strfield",3:"intfield",4:"floatfield",5:"boolfield"};
  ```
- extraParams - Not required
- callback - The action's callback handler. The definition of the callback action should take the error message as the first string parameter followed by the storeStatement name.

The deleteStoreStatement is defined as:

```
action deleteStoreStatement(
    string statementName,
```

```
    string tableName,
    dictionary<string,string> extraParams,
    action<Connection,string,string> callback)
```

The ADBC Event Application Programming Interface

Handling data storing errors

If your application uses the storeData or storeEvent actions, you can use the setStoreErrorCallback action to handle failures. This is useful for applications that make a large number of store calls where high performance is important and acknowledgement for an individual store operation call is not required. A single setStoreErrorCallback action can handle the failure of multiple store calls. The setStoreErrorCallback action is defined as follows:

```
action setStoreErrorCallback(
   action<Connection, integer, integer, string> callback)
 {
```

Calls to storeData and storeEvent actions return unique integer identifiers; use these identifiers in the setStoreErrorCallback action. The first integer specifies the indentifier of the first store action where an error occurred; the second integer specifies the indentifier of the last store action error. callback specifies the name of the user-defined error handling action.

The ADBC Event Application Programming Interface

Committing transactions

By default, the auto-commit behavior assumes the AutoCommit and StoreCommitInterval properties specified in the adapter's configuration file and the open action are using the default values. If you want more control over when changes are committed to a database, set the openDatabase action's autoCommit parameter to false and in your EPL code, manually commit data using the Connection event's commitRequest action.

1. Create a callback action to handle the results of the commitRequest action.
2. Call the commitRequest action of the Connection event (for the open database) with the name of the callback action.

The definitions for the two forms of the commitRequest action are:

```
action commitRequest(
 action<Connection, integer, string, string> callback) returns integer
action commitRequestFull(
 string token,
 dictionary<string, string> extraParams,
 action<Connection, integer, string, string> callback) returns integer
```

The ADBC Event Application Programming Interface

Rolling back transactions

To rollback a database transaction, your application should use the Connection event's rollbackRequest action. If you want to use rollback actions, you need to turn autocommit off.

1. Create a callback action to handle the results of the rollbackRequest action.
2. Call the rollbackRequest action of the Connection event (for the open database) with the name of the callback action.

The definitions for the two forms of the `rollbackRequest` action are:

```
action rollbackRequest(
 action<Connection, integer, string, string> callback) returns integer
action rollbackRequestFull(
 string token,
 dictionary<string, string> extraParams, string token,
 action<Connection, integer, string, string> callback) returns integer
```

The ADBC Event Application Programming Interface

Running commands

To execute database commands, such as creating a table or SQL operations such as Delete and Update, use the `Connection` event's `runCommand` action.

1. Call the `runCommand` action of the `Connection` event (for the open database) with the a string containing the SQL command to execute and the name of the callback action.

2. Create a handler action for the `runCommand` callback action.

The definitions for the two forms of the `runCommand` are:

```
action runCommand(
  string commandString,
  string token,
  action <Connection, string, string> callback)
```

and

```
action runCommandFull(
  string commandString,
  string token,
  dictionary<string, string> extraParams,
  action<Connection, string, string> callback)
```

The ADBC Event Application Programming Interface

Executing queries

An Apama application can execute three types of SQL queries on databases:

- Standard query — An SQL query that you write in your EPL code. This is typically a simple query provided as a string when your EPL code initializes the query. The query string is used when the query is submitted to the database when your EPL code calls the action that starts the query. See "Executing standard queries" on page 211.

- Prepared query — An SQL query that uses a *prepared statement* or *stored procedure*, both of which are stored in the database. Because they are stored in the database, prepared queries are more efficient than standard and named queries as they do not need to be compiled and destroyed each time they are run. Input parameters for prepared queries are not set during initialization. They are set after initialization, but before the query is submitted to the database when the query start action is called. See "Prepared statements" on page 213 and "Stored procedures" on page 214.

- Named query — An SQL query that you write in an XML file as part of the Apama Studio project. Typically, you use a named query if you plan to use the query multiple times (as a template, supplying parameterized values). If the query is relatively complex, it is useful to separate it from your EPL code for readability. Your EPL code specifies the query template name and the template parameter names and values to use when it initializes the query. The template name and parameters are used when the query is submitted to the database when your EPL code calls the action that starts the query. See "Named queries" on page 216.

The ADBC Event Application Programming Interface

Executing standard queries

In order to execute a standard query, your application needs to implement the following steps:

1. Create a new `Query` event.
2. Initialize the query by calling the `Query` event's `initQuery` action passing in the name of the database's `Connection` event and the query string.
3. Call the `Query` event's `setReturnType` action to specify the return type. Apama recommends specifying the return type using one of the following constants:

   - `Query.RESULT_EVENT`
   - `Query.RESULT_EVENT_HETERO`
   - `Query.NATIVE`
   - `Query.HISTORICAL`

   See "Return Types" on page 211 below for more information on return types.
4. If the return type is `Native`, indicate the event type to be returned by specifying it with the `Query` event's `setEventType` action.

   The `setEventType` action is defined as:

   ```
   action setEventType(string eventType)
   ```

   In addition, you need to add mapping rules to the ADBC adapter's configuration file for the event type being returned.
5. In addition, if the return type is `Native`, specify the database table column that stores the event's timestamp with the `Query` event's `setTimeColumn` action.

   The `setTimeColumn` action is defined as:

   ```
   action setTimeColumn(string timeColumn)
   ```
6. If the query will return a large number of results, call the `Query` event's `setBatchSize` action passing in an integer setting the batch size.
7. If you set a batchsize, also use the `Query` event's `setBatchDoneCallback` action passing in values for the `token` and `callback` parameters.

   ```
   action setBatchDoneCallback(
      string token,
      action<Query,string,integer,float,string,string> callback)
   ```
8. If the application needs to know the query's result set schema, call the `Query` event's `setSchemaCallback` action passing in the name of the handler action.
9. Call the `Query` event's `start` action passing in the name of the handler action that will be called when the query completes.

Return Types

- NATIVE — This return type is most commonly used for playback. When a query is run, each row of the query will be passed through the IAF mapping rules and the matching event will be sent as-is to the correlator. The Native return type would not be used for general database queries.

  In addition to specifying the Native return type, your query needs to specify the event type to be returned and the name of the database table's column that contains the event's time stamp. Specify the event by using the Query event's setEventType action; specify the time column by using the Query event's setTimeColumn action. You also need to add mapping rules for this event type to the ADBC adapter's configuration file.

- HISTORICAL — This return type is also used for playback. When a query is run, each row of the query will be passed through the IAF mapping rules and then the matching event will be “wrapped” in a container event. The container event will have a name based on that of the event name. For example a Tick event would be wrapped in a HistoricalTick event. Event wrapping allows events to be sent to the correlator without triggering application listeners. A separate user monitor can listen for wrapped events, modify the contained event, and reroute it such that application listeners can match on it. The Wrapped return type would not be used for general database queries.

- RESULT_EVENT — This return type is used for general database queries. When a query is run, each row in the result set will be mapped to a dictionary in a generic ResultEvent. The ADBC adapter will generate a SchemaEvent containing the schema (name and type) of the fields in the result set of the query. The SchemaEvent will be sent first, before any ResultEvents.

The definition for ResultEvent is:

```
event ResultEvent {
  integer messageId;  // Unique id of query
  string serviceId;
  integer schemaId;   // ResultSchema event schemaId to use with this ResultEvent
  dictionary <string, string> row; // Data
}
```

The definition for ResultSchema is:

```
event ResultSchema {
  integer messageId; // Unique id of query
  string serviceId;
  integer schemaId;
  sequence <string> fieldOrder;
  dictionary <string, string> fieldTypes;
  sequence <string> indexFields;
  dictionary<string,string> extraParams;
}
```

- RESULT_EVENT_HETERO — This return type is intended for advanced database queries. It is not applicable to SQL databases. Some market databases support queries which can, in essence, return multiple tables. For example a market database might allow queries which return streams of both Tick and Quote data. For such databases multiple SchemaEvents would be generated indexed by id.

Executing queries

Stopping queries

The following action cancels all outstanding queries in the queue.

```
action stopAllQueries(
   action<Connection,string> callback)
```

Executing queries

Preserving column name case

In order to provide compatibility for a wide number of database vendors, the ADBC adapter normally converts column names to lower case. However, if you want to execute complex queries where the `_ADBCType` or `_ADBCTime` are returned as part of the query rather than being specified using the `setEventType` and `setTimeColumn` actions on the query, you need to set the `ColumnNameCase` property in the ADBC adapter's configuration file to `unchanged`.

Setting the `ColumnNameCase` property is done by manually editing the `ColumnNameCase` property to the configuration file.

1. In the **Project Explorer**, in the project's `Adapters` node, expand the ODBC or JDBC adapter, and double-click the adapter instance to open it in the ADBC adapter editor.
2. Display the ADBC adapter editor's **XML source** tab.
3. In the `<transport>` element, edit the `ColumnNameCase` property as follows:

   ```
   <property name="ColumnNameCase" value="unchanged"/>
   ```

4. Save the ADBC adapter instance's configuration.

When the `ColumnNameCase` property is set to `unchanged`, you can specifiy a query string in the form

```
string queryString := "SELECT *, 'Trade' AS _ADBCType FROM TradeTable
                            WHERE symbol = "ADL";
```

The other values for the `ColumnNameCase` property can be `lower`, (the default) and `upper`.

Executing queries

Prepared statements

Apama applications can use prepared statements when executing queries. Prepared statements have the following performance advantages over standard queries:

- The query does not need to be re-parsed each time it is used.
- The query allows for replaceable parameters.

The ADBC Event Application Programming Interface

Using a prepared statement

To use a prepared statement:

1. Create a new `Query` event.
2. Create a new `PreparedQuery` event.
3. Call the new `PreparedQuery` event’s `init` action, passing in the database connection, the query string, the input types if using replaceable parameters and the output types if it will be used as a stored procedure.

   The definition for the `init` action is:

   ```
   action init (
      Connection conn,
      string queryString,
      sequence<string> inputTypes,
      sequence<string> outputTypes)
   ```

The arguments for the init action are:

- conn — The name of the database's Connection event.
- queryString — The SQL query string; you can use question mark characters ? to indicate replaceable parameters.
- inputTypes — This is optional, but if you use replaceable parameters in the queryString, you need to specify the types that will be used in the query.
- outputTypes — This is optional, but if the PreparedQuery event is to be used for a stored procedure and it uses output parameters, you need to specify the output types.

For example:

```
sequence<string> inputTypes := ["INTEGER","INTEGER"];
myPreparedQuery.init (
   myConnection,
   "SELECT * FROM mytable WHERE inventory > ? and inventory <?",
   inputTypes, new sequence<string>);
```

4. Call the new PreparedQuery event's create action, passing in the name of the callback action.
5. In the callback action's code, call the Query event's initPreparedQuery action (instead of the initQuery action), passing in the name of the PreparedQuery event. See "Executing standard queries" on page 211.
6. Call the Query event's setInputParams action, passing in the values to be used for the replaceable parameters. The definition of the setInputParams action is:

   ```
   setInputParams(sequence<string> inputParams)
   ```

   If you want to use NULL for the value of a replaceable parameter, use ADBC_NULL.
7. If necessary, call any of the other Query actions, such as setBatchSize, as required.
8. Call the Query event's start action as you would when executing any other query. See "Executing standard queries" on page 211.

Prepared statements

Stored procedures

Apama applications can use stored procedures when executing queries. Using stored procedures is similar to using prepared statements. The difference is that a stored procedure needs to specify the name of the stored procedure and the output types returned by the query.

The ADBC Event Application Programming Interface

Using a stored procedure

Queries in Apama application use stored procedures by specifying the name of the stored procedure in a prepared statement's query string.

To use a stored procedure:

1. Create a new Query event.
2. Create a new PreparedQuery event.

3. Call the new `PreparedQuery` event's `init` action, passing in the database connection, the query string, the input types, and the output types.

   The definition for the `init` action is:

   ```
   action init (
     Connection conn,
     string queryString,
     sequence<string> inputTypes,
     sequence<string> outputTypes)
   ```

   The arguments for the `init` action are:

   - `conn` — The name of the database's `Connection` event.
   - `queryString` — The SQL query string; enclose the name of the database's stored procedure in curly brace characters `{}` and use question mark characters `?` to indicate replaceable parameters.
   - `inputTypes` — Specify the types that will be used for the replaceable parameters in the `queryString`.
   - `outputTypes` — Specify the types that will be used for the replaceable parameters in the result.

   For example:

   ```
   sequence<string> inputTypes := ["INTEGER", "NULL", "INTEGER"];
   sequence<string> outputTypes := ["NULL", "INTEGER", "INTEGER"];
   myPreparedQuery.init (
       myConnection,
       "{call myprocedure(?,?,?)}",
       inputTypes,
       outputTypes);
   ```

   - If a parameter is used as both an input and output type, it must be specified in both places.
   - If it is only an input type it must be specified as NULL in `outputType`.
   - If it is only an output type it must be specified as NULL in `inputType`.

   Therefore, in the example above, the first parameter is just an input type; the second parameter is just an output type; and the third parameter is both an input and output type.

4. Call the new `PreparedQuery` event's `create` action, passing in the name of the callback action.

5. In the callback action's code or once the callback action has been called, call the `Query` event's `initPreparedQuery` action instead of the `initQuery` action, passing in the name of the `PreparedQuery` event. An error will be reported if the Query event's `initPreparedQuery` is called before the `PreparedQuery` create callback has been called. See "Executing standard queries" on page 211.

6. Call the `Query` event's `setInputParams` action, passing in the values to be used for the replaceable parameters. The definition of the `setInputParams` action is:

   ```
   setInputParams(sequence<string> inputParams)
   ```

   If you want to use NULL for the value of a replaceable parameter, use `ADBC_NULL`.

7. If necessary, call any of the other `Query` actions, such as `setBatchSize`, as required.

8. Call the `Query` event's `start` action as you would when executing any other query. See "Executing standard queries" on page 211.

Stored procedures

Named queries

Apama applications can use named queries. Named queries are templates with parameterized values and are stored in Apama projects. Queries of this type provide advantages for queries that will be used multiple times. They also serve to keep the SQL query strings separate from the application's EPL code.

To use a named query, your EPL code needs to specify the query template name and the template parameter names and values to use when it initializes the query. The template name and parameters are used when the query is submitted to the database.

You define a named query as a query template in the ADBC adapter's `ADBC-queryTemplates-SQL.xml` file. This file contains some pre-built named queries:

- `findEarliest` — Get the row with the earliest time (based on the stored event's timestamp).
- `findLatest` — Get the row with the latest time.
- `getCount` — Get the number of rows in a table.
- `findAll` — Get all the rows from a table.
- `findAllSorted` — Get all the rows from a table ordered by column.

The ADBC Event Application Programming Interface

Using named queries

To use a named query:

1. Create a new `Query` event.
2. Initialize the query by calling the `Query` event's `initNamedQuery` action, passing the name of the database's `Connection` event, the name of the query template, and a `dictionary<string, string>` containing the names and values of the named query's parameters.
3. Call the `Query` event's `setReturnType` action to specify the return type to be `ResultEvent`. When a query is run, each row in the result set will be mapped to a dictionary event field in a `ResultEvent` event.
4. Call the `Query` event's `setReturnEventCallback` action to specify the callback action that will handle the results returned by the query.
5. If the query will return a large number of events (on the order of thousands):
   a. Call the `Query` event's `setBatchSize` action passing an integer that sets the batch size. The query returns results in batches of the specified size.
   b. Call the `Query` event's `setBatchDoneCallback` action passing the name of the handler action.
   c. Define the `setBatchDoneCallback` action to define what to do when a batch is complete. You must call the `Query` event's `getNextBatch` action to continue receiving the query results. The batch size for the next batch is set by passing an integer parameter for the batch size. You could also call the stop action to stop the query, rather than continuing to receive batches of data.
6. Call the `Query` event's `start` action passing the name of the handler action that will be called when the query completes.
7. Create the callback action that you specified in Step 4, to handle the results returned by the query.

8. Each row of data that matches the query results in a call to the callback action, returning the row results in a parameter of `ResultEvent` type. The `ResultEvent` type contains a dictionary field that contains the row data.

9. Create the action that specifies what to do when the query completes (when all results are returned).

The following example uses the `initNamedQuery` action call to initialize the query, specifying the `findEarliest` named query and `stock_tables` as the value for the named query's `TABLE_NAME` parameter.

```
using com.apama.database.Connection;
using com.apama.database.Query;
using com.apama.database.ResultEvent;

monitor ADBCexample {
    Connection conn;
    Query query;

    string serviceId := "com.apama.adbc.JDBC_INSTANCE_1";
    string dbUrl := "jdbc:mysql://127.0.0.1:3306/exampledb";
    string user := "root";
    string password := "mysql";
    string queryString := "SELECT * FROM sys.tables";
    string tableName := "stock_table";
    dictionary<string,string> paramTable :=
         {"TABLE_NAME":tableName,"TIME_COLUMN_NAME":"tbd"};

  action onload() {
    conn.openDatabase(serviceId, dbUrl, user, password, "",
      handleOpenDatabase);
  }
  action handleOpenDatabase (Connection conn, string error){
    if error.length() != 0 then {
      log "Error opening database : " + error at ERROR;
    }
    else {
      log "Database is open." at INFO;
      runQuery();
    }
  }
  action runQuery {
    query.initNamedQuery(conn, "findEarliest", paramTable);
    query.setReturnType("ResultEvent");
    query.setResultEventCallback(handleResultEvent);
    query.start(handleQueryComplete);
  }
  action handleResultEvent(Query q, ResultEvent result) {
  log result.toString() at INFO;
  }
action handleQueryComplete(Query query, string error,
    integer eventCount, float lastEventTime) {
      if error.length() != 0 then {
         log "Error running query '" + queryString + "': " +
            error at ERROR;
      }
      else {
      log "  Query '" + queryString + "' successfully run." at INFO;
      log "    Total events: " + eventCount.toString() at INFO;
        if lastEventTime > 0.0 then {
          log "    Last Event Time: " + lastEventTime.toString()
             at INFO;
        }
      }
    conn.closeDatabase(handleCloseDatabase);
  }
  action handleCloseDatabase(Connection conn, string error) {
    if error.length() != 0 then {
      log "Error closing database : " + error at ERROR;
    }
    else {
      log "Database closed." at INFO;
    }
  }
}
```

Named queries

Creating named queries

Each named query in the ADBC-queryTemplates-SQL.xml file is defined in an XML <query> element. Each <query> element has the following attributes:

- name — The name of the query.
- description — A short description of the query.
- implementationFunction — The substitution function that the adapter uses to process the named query. The substitution function allows you to specify tokens that are replaced by parameters with matching names.
- inputString — A string that contains the substitution tokens you want to replace with values specified as parameters.

A <query> element can also have one or more optional <parameter> child elements. Each <parameter> element has the following attributes:

- description — A short description of the parameter.
- name — The name of the parameter.
- type — The data type of the parameter.
- default — The default value of the parameter.

As an example, the following XML code in the ADBC-queryTemplates-SQL.xml file defines the pre-built findEarliest named query. The query returns the row with the earliest time.

```
<query
    name="findEarliest"
    description="Get the row with the earliest time."
    implementationFunction="substitution"
    inputString="select * from ${TABLE_NAME} order by ${TIME_COLUMN_NAME}
           asc limit 1">
      <parameter
        description="Name of a table to query"
        name="TABLE_NAME"
        type="String"
        default=""/>
      <parameter
        description="Name of the time column"
        name="TIME_COLUMN_NAME"
        type="String"
        default="time"/>
</query>
```

To create a named query:

1. In the Project Explorer, expand the project's Adapters node and open the adapter folder.
2. Double-click the instance configuration file to open it in the adapter editor.
3. In the adapter editor, select the Advanced tab.

4. Click the `ADBC-queryTemplates-SQL.xml` file to open it.
5. Select the Design tab.
6. On the Design tab, right-click the `namedQuery` element and select Add Child > New Element.
7. In the **New Element** dialog, type `query`, then click OK. A new query row is added to the list.
8. For each of the four attributes (`name`, `description`, `implementationFunction`, `inputString`):
   a. Right-click the `query` element you added in Step 4, and select Add Attribute > New Attribute.
   b. In the **New Attribute** dialog, provide a Name and a Value for the attribute.
9. If you want the query to use input parameters, for each parameter:
   a. a. Right-click the `query` element and select Add Child > New Element.
   b. b. In the **New Element** dialog, type `parameter`, then click OK.
   c. c. Create the following attributes for each parameter:
      - `description`
      - `name`
      - `type`
      - `default`
10. Save the project's version of the query template file.

Named queries

The Visual Event Mapper

When you add or open an instance of the ADBC Adapter, the adapter editor provides a Visual Event Mapper. The Event Mapper is available by selecting the Event Mapping tab. With the Event Mapper you specify an Apama event type and a table in an existing ODBC or JDBC database. When you save the adapter configuration file, Apama Studio creates the rules that provide the mapping between the fields in the event and the columns in the database. The mapping rules are stored in the adapter instance's configuration file.

The Generate Store Monitors option in the Visual Event Mapper specifies whether or not Apama Studio generates all the necessary EPL code for monitors that listen for events of the specified types as well as for the EPL code that interacts with the database -- opening the database, checking the adapter status, storing event data, etc. This is the default setting. If you turn this option off, you need to write the EPL code for event listeners and for interacting with the database.

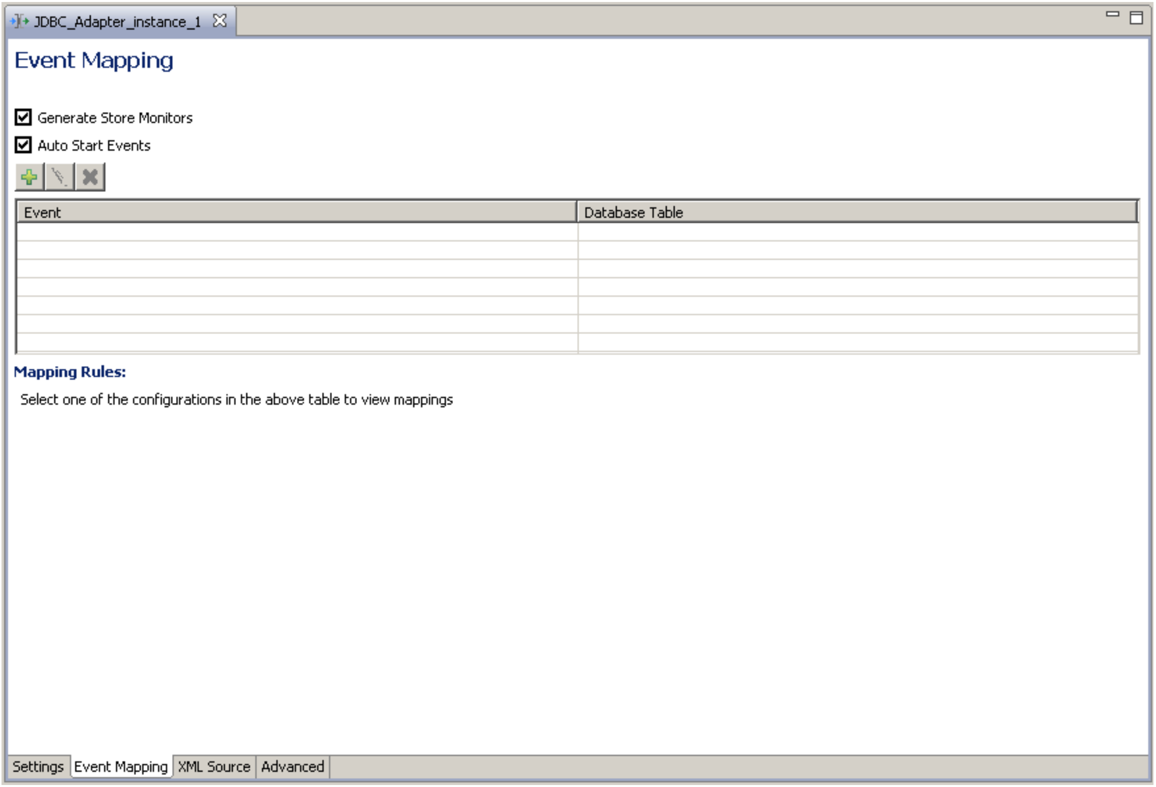
The Auto Start Events option in the Event Mapper specifies whether or not Apama Studio generates events that cause Apama Studio to automatically start saving event data when the application is launched. If you turn this option off, your application needs to manually send a `StartStoreConfiguration` event in order to start saving data.

Using the Apama Database Connector

Using the Visual Event Mapper

To map an Apama event to a table in a database:

1. Add a new instance of the ADBC Adapter or open an existing instance and select the adapter editor's Event Mapping tab.

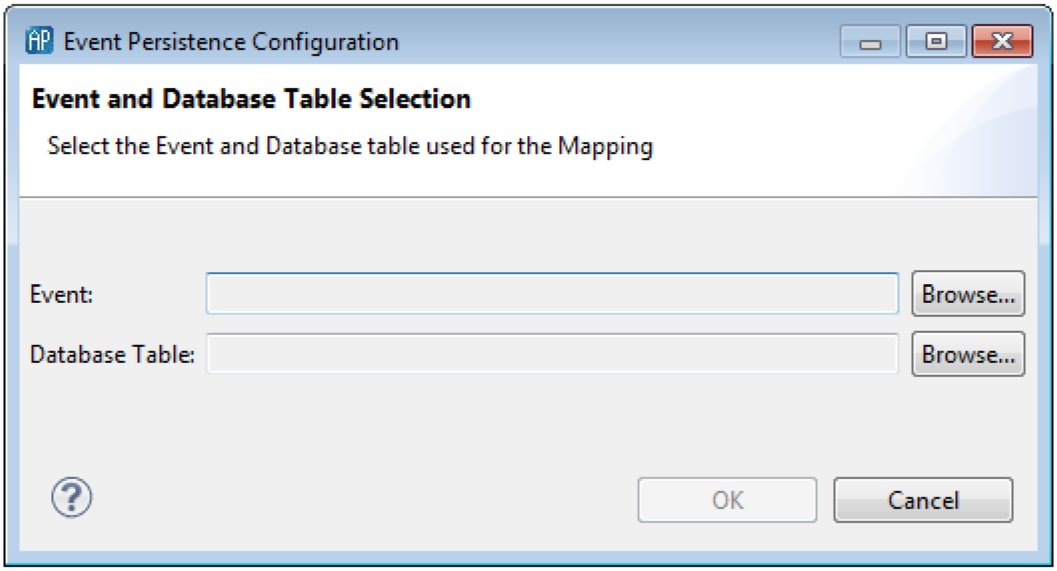


2. If you want Apama Studio to automatically generate an EPL monitor to listen for events of the specified type, make sure the Generate Store Monitors option is enabled; this is the default setting. In addition to generating all the necessary EPL code for monitors that listen for events of the specified types, Apama Studio generates all the EPL code that interacts with the database -- opening the database, checking the adapter status, storing event data, etc. This setting is useful if your application does not need to guarantee that each event is persisted. The generated monitor provides a best effort storage implementation suitable for storing data to be analyzed in tools like Analyst Studio. The generated monitor does not perform any filtering so all events of the type specified will be stored.

   If your application needs to perform filtering of the events or needs to guarantee that each event will be persisted, you should disable Generate Store Monitors option and manually write the required code for the EPL monitors and for interacting with the database.

3. Make sure there is a check mark in the Auto Start check box (this is the default) if you want to start saving event data immediately when you launch the project. If you clear the check mark in the Auto Start check box, your application will need to manually send a `StartStoreConfiguration` event in order to start storing events.

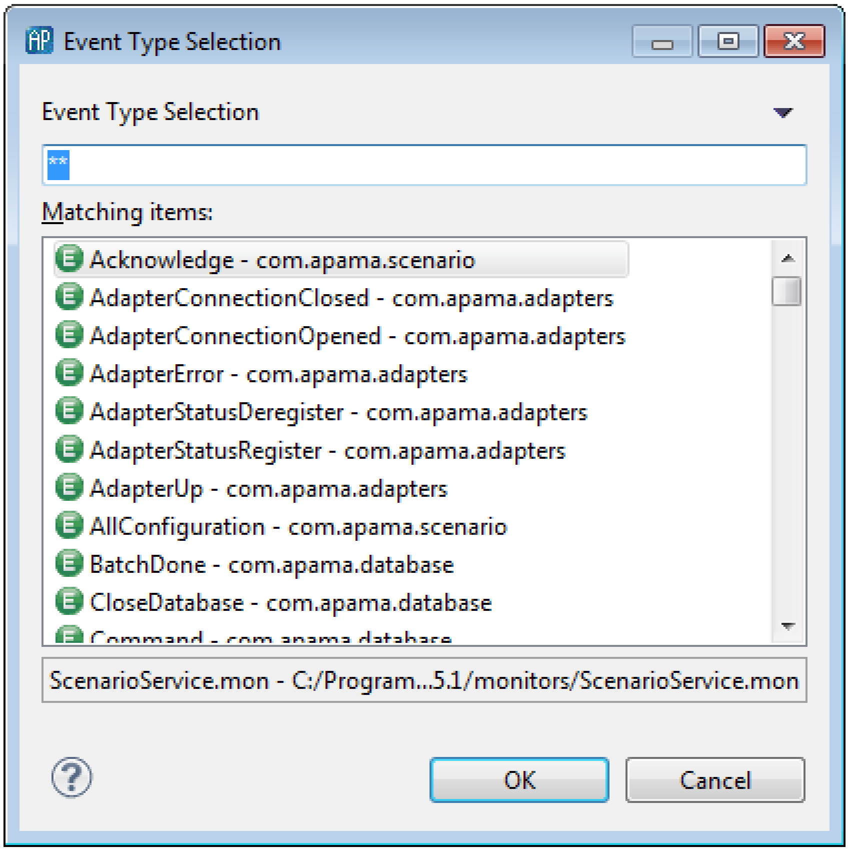
4. In the adapter editor, click the Add button. The **Event Persistence Configuration** dialog opens.



Event and Database Table Selection

Select the Event and Database table used for the Mapping

5. In the **Event Persistence Configuration** dialog, click the Browse button next to the Event field. The **Event Type Selection** dialog opens displaying the available event types you can select from. Only events that can be emitted are shown; events that contain fields with contexts or actions are not displayed.



**

Acknowledge - com.apama.scenario
AdapterConnectionClosed - com.apama.adapters
AdapterConnectionOpened - com.apama.adapters
AdapterError - com.apama.adapters
AdapterStatusDeregister - com.apama.adapters
AdapterStatusRegister - com.apama.adapters
AdapterUp - com.apama.adapters
AllConfiguration - com.apama.scenario
BatchDone - com.apama.database
CloseDatabase - com.apama.database
Command - com.apama.database

ScenarioService.mon - C:/Program...5.1/monitors/ScenarioService.mon

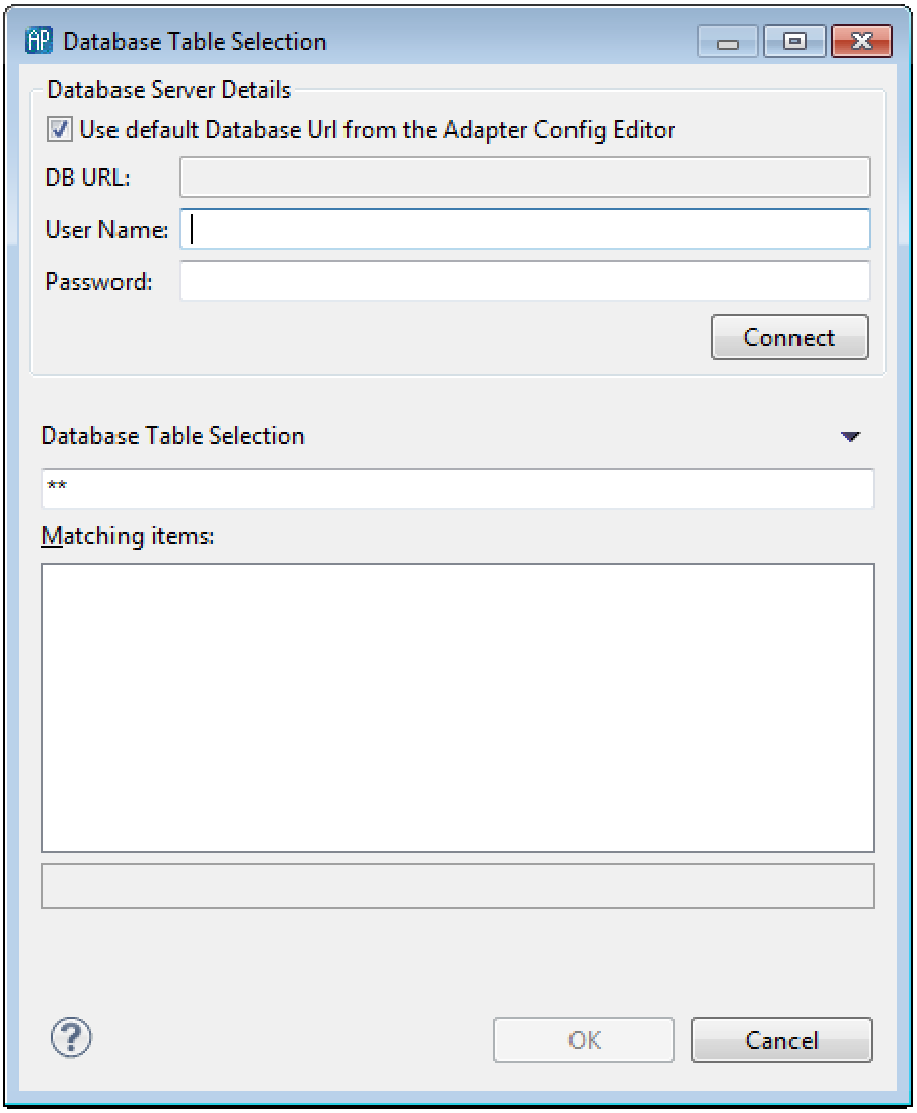
6. In the **Event Type Selection** dialog, select the event type you want to map as follows:

   a. In the Choose an Event Type field, enter the name of the event. As you type, event types that match what you enter are shown in the Matching Items list. When you select an event, the full name is shown on the dialog's status line. You can turn off this display with the dialog's Down Arrow menu icon (▾).

b. In the Matching Items list, select the name of the event type you want to map. The name of the EPL file that defines the selected event is displayed in the status area at the bottom of the dialog.

c. Click OK.

7. In the **Event Persistence Configuration** dialog, click the Browse button next to the Database table field. The **Database Table Selection** dialog opens.



☑ Use default Database Url from the Adapter Config Editor

Database Table Selection: **

8. In the **Database Table Selection** dialog, select the database table to which you want to map the event's fields as follows:

a. In the Database Server Details section, specify the DB URL, User Name, and Password. By default, the DB URL uses the value used in the adapter configuration settings. You can change the name of the database by un-checking the check box and entering a new name. (Note, you cannot change the type of database.)

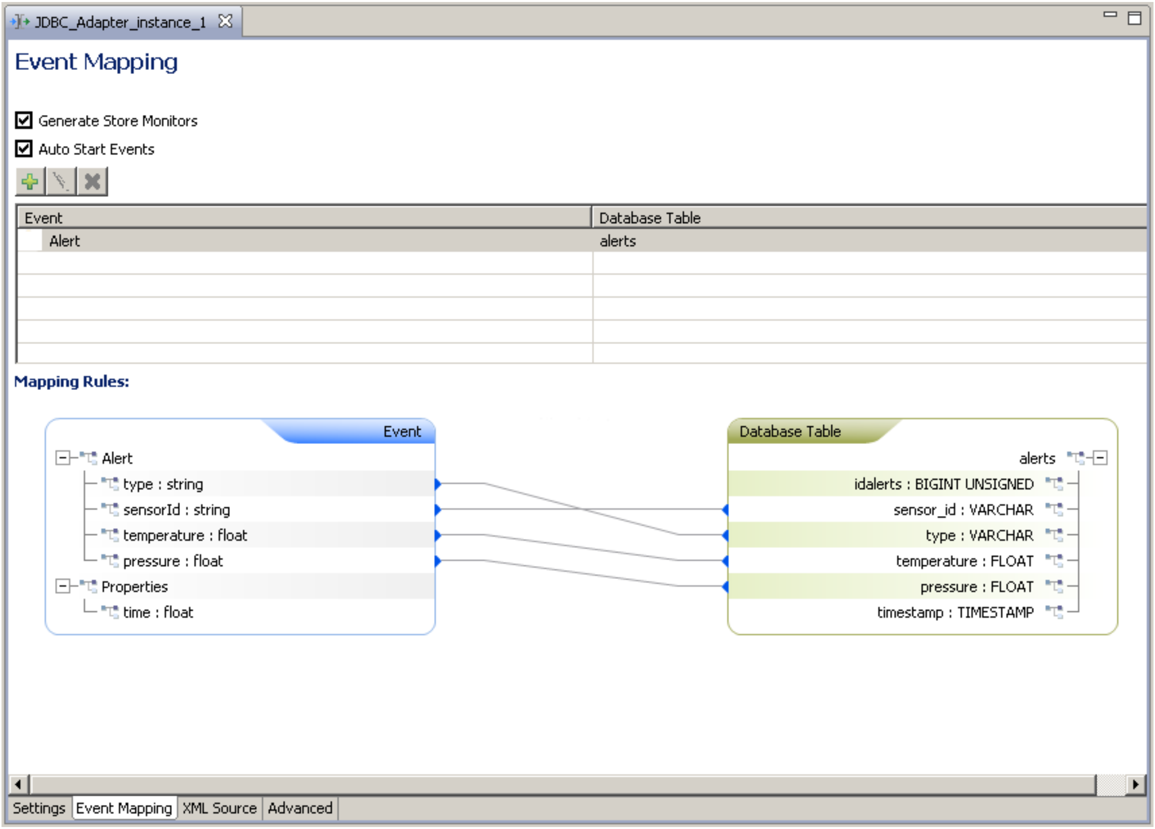
b. Click Connect to access the database.

c. Select the name of the table from the Matching Items list or enter text in the Database Table Selection field. As you type, table names that match what you enter are shown in the Matching Items list. When you select a table, its name is also shown on the dialog's status line. You can turn off this display with the dialog's Down Arrow menu icon (▾).

d. In the Matching Items list, select the name of the database table where you want to store the event data.

e. Click OK.

9. In the **Event Persistence Configuration** dialog, click OK. The adapter editor display is updated to show the name of the event type and the database table in the Event section. The Mapping Rules section displays lists for Event and Database Table.



JDBC_Adapter_instance_1

Event Mapping

☑ Generate Store Monitors

☑ Auto Start Events

Event	Database Table
Alert	alerts

Mapping Rules:

10. For each event field you want to store in the Event list click on the field and draw a line to the desired column in the Database Table list.

When you save the adapter instance configuration, Apama Studio generates mapping rules that specify the associations between event fields and database columns. Apama Studio also generates a monitor that listens for events of the specified type. The monitor allows the Apama application to manage when the events are written to the database.

The Visual Event Mapper

Playback

If event data is stored in a database, you can play back the events through the correlator using Apama Studio’s Data Player. The Data Player consists of the Query Editor and the Data Player control. In the Query Editor you create and modify queries in order to specify what event data you want to play back. The Data Player control allows you to specify what query to use and how fast to play back the event data.

For full information on the Data Player, see "Using the Data Player" in *Using Apama Studio*.

Command line utilities

When you have stored event data in a database and created queries in Apama Studio, you can also launch a playback session using the Data Player command line utility, `adbc_management`. The `adbc_management` utility is described in "Using the data player command-line interface" on page 165.

Using the Apama Database Connector

Sample applications

Several sample applications in the Apama installation illustrate the use of the ADBCHelper and ADBC Event APIs. The samples are located in the `samples\adbc` directory of the Apama installation. The `api-helper-example` uses the ADBCHelper API; the other examples use the ADBC Event API. The samples include:

- `api-helper-example` — An EPL application that shows how to open and close a database and execute SQL commands and queries using the ADBCHelper API.
- `api-example` — An EPL application that uses the ADBC Event API to show how to use all API operations except those for storing data. Included is code for discovering data sources and databases, opening and closing databases, and executing queries.
- `store-data` — An EPL application that shows how to open a database, create a table, and store non-event data using the ADBC Event API.
- `store-events` — An EPL application that shows how to open a database, create a table, and store event data using the ADBC Event API.

Using the Apama Database Connector

Format of events in .sim files

In Apama 4.1 and earlier, Apama captured data streaming through the correlator into proprietary `.sim` files. These files consist of comma-delimited values. You can use the Apama Studio Data Player to play back event data from existing `.sim` files. Note, however, that the ADBC does not write data in `.sim` format.

Apama `.sim` files contain string versions of events and can also contain an optional header that specifies the default timezone for the series. The timezone identifiers can be any supported by Java 1.7. The format of the events contained in a `.sim` file is:

- timestamp — a float specifying UTC seconds since 01/01/1970.
- event origin — a string specifying whether the event is an internal or external event.
- event — a stringified version of the event itself.

Elements of the exported event are separated by commas.

The following is an example of an external event from a `.sim` file (each event is stored on a single line, here they are shown on separate lines for clarity):

```
1161287634.200,
    external,
    com.apama.backtest.RawTick(
        com.apama.marketdata.Tick("RACK",34.97,11,{}))
```

The following is an example of an internal event from a `.sim` file:

```
1161287629.600,
    internal,
    com.apama.backtest.RawTick(
        com.apama.marketdata.Tick("RACK",34.96,64,{}))
```

The events in the example are `RawTick` events with embedded `Tick` events.

The following is an example of the optional header containing a specified default timezone:

```
#
# <Timezone=America/New_York>
#
```

Comments in .sim files

You can add comments when you edit `.sim` files. Introduce lines containing comments with either `#` or `//`.

Using the Apama Database Connector

11 The Apama Web Services Client Adapter

The Apama Web Services Client adapter is a SOAP-based adapter that allows Apama applications to invoke Web services. To use the Apama Web Services Client adapter in an Apama project, you need to do the following:

- Add the pre-packaged bundle of adapter resources for the Web Services Client adapter.
- Specify the location of the of the Web Service, using the URI of its Web Service Definition Language file (WSDL).
- Specify the Web Service operation or operations to invoke.
- Specify what Apama events will interact with the Web Service operations.
- Create mapping rules that associate the fields in the Apama events with the Web Service operations' parameters.

When you add and configure an instance of the Web Services Client adapter, Apama Studio automatically generates the configuration files, service monitors, and adapter artifacts that are necessary to deploy and run the Apama project's adapter instances.

You can add multiple instances of the Web Services Client adapter to an Apama project. The files generated by Apama Studio are specific to each adapter instance.

Note that Apama applications only invoke Web Service operations in the consume use case; it is not possible to expose actions in Apama applications as Web Services operations.

Web Services Client adapter overview

The process of adding a Web Services Client adapter to an Apama project involves the following:

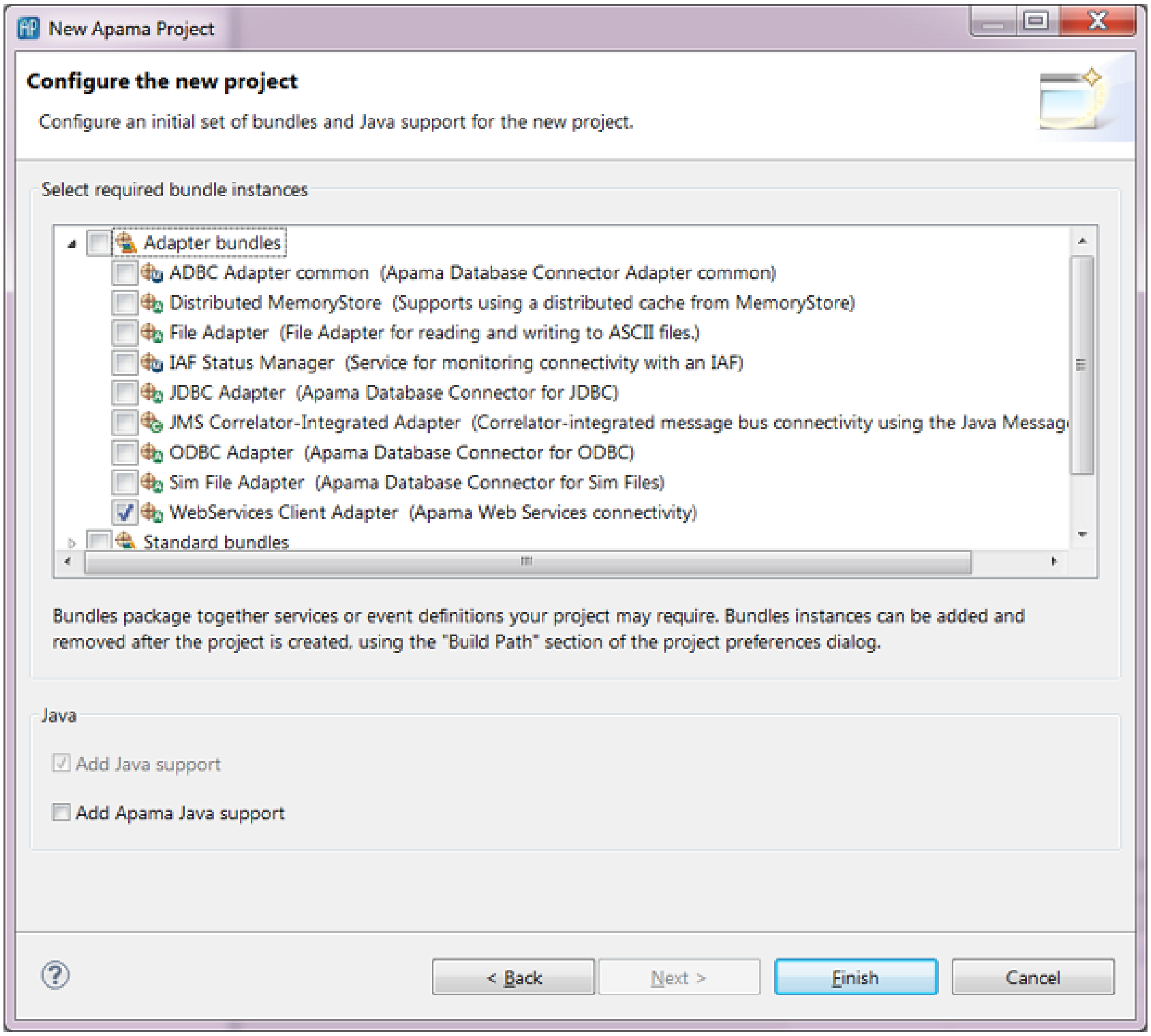
1. Add the Web Services Client adapter resource bundle to an Apama project.
2. Configure the Web Services Client adapter, which involves:
    a. Specifying the Web Service Definition File (WSDL) that points to the Web Service.
    b. Specifying a Web Service operation or operations your application will invoke.
    c. Mapping Web Service Client adapter parameters.

The Apama Web Services Client Adapter

Adding a Web Services Client adapter to an Apama Studio project

To add a Web Services Client adapter to a project:

1. Select File > New > Apama Project from the Apama Studio menu. This launches the **New Apama Project** wizard.
2. In the **New Apama Project** wizard, give the project a name, and click Next. The second page of the wizard is displayed, listing the available Apama resource bundles.



New Apama Project

Configure the new project

Configure an initial set of bundles and Java support for the new project.

Select required bundle instances

- □ Adapter bundles
  - □ ADBC Adapter common (Apama Database Connector Adapter common)
  - □ Distributed MemoryStore (Supports using a distributed cache from MemoryStore)
  - □ File Adapter (File Adapter for reading and writing to ASCII files.)
  - □ IAF Status Manager (Service for monitoring connectivity with an IAF)
  - □ JDBC Adapter (Apama Database Connector for JDBC)
  - □ JMS Correlator-Integrated Adapter (Correlator-integrated message bus connectivity using the Java Messag
  - □ ODBC Adapter (Apama Database Connector for ODBC)
  - □ Sim File Adapter (Apama Database Connector for Sim Files)
  - ☑ WebServices Client Adapter (Apama Web Services connectivity)
- □ Standard bundles

Bundles package together services or event definitions your project may require. Bundles instances can be added and removed after the project is created, using the "Build Path" section of the project preferences dialog.

Java

☑ Add Java support

□ Add Apama Java support

3. In the **New Apama Project** wizard, in the Select required bundle instances field, select the WebService Client Adapter bundle. When you select this adapter, Apama Studio automatically enables the Add Java Support setting.

4. Click Finish.

Apama Studio generates the basic resources for Web Services Client adapter in the project. From here you need to configure the adapter instance in order to specify the Web Service your application will access. See "Configuring a Web Services Client adapter" on page 229 for information on configuring the adatper instance.

The Apama Web Services Client Adapter

Configuring a Web Services Client adapter

After you add the Web Service Client adapter's resource bundle to an Apama project, you need to configure the adapter to interact with the external Web Service. This involves the following:

1. Specify the Web Service Definition File (WSDL) that points to the Web Service.

2. Specify a Web Service operation or operations your application will invoke.

3. Map the parameters of the Web Service operations to Apama events.

When you save the information that you add to a Web Service Client adapter, Apama Studio generates all the run-time support resources necessary for the adapter. For details on these resources, see "Web Services Client adapter artifacts" on page 268.

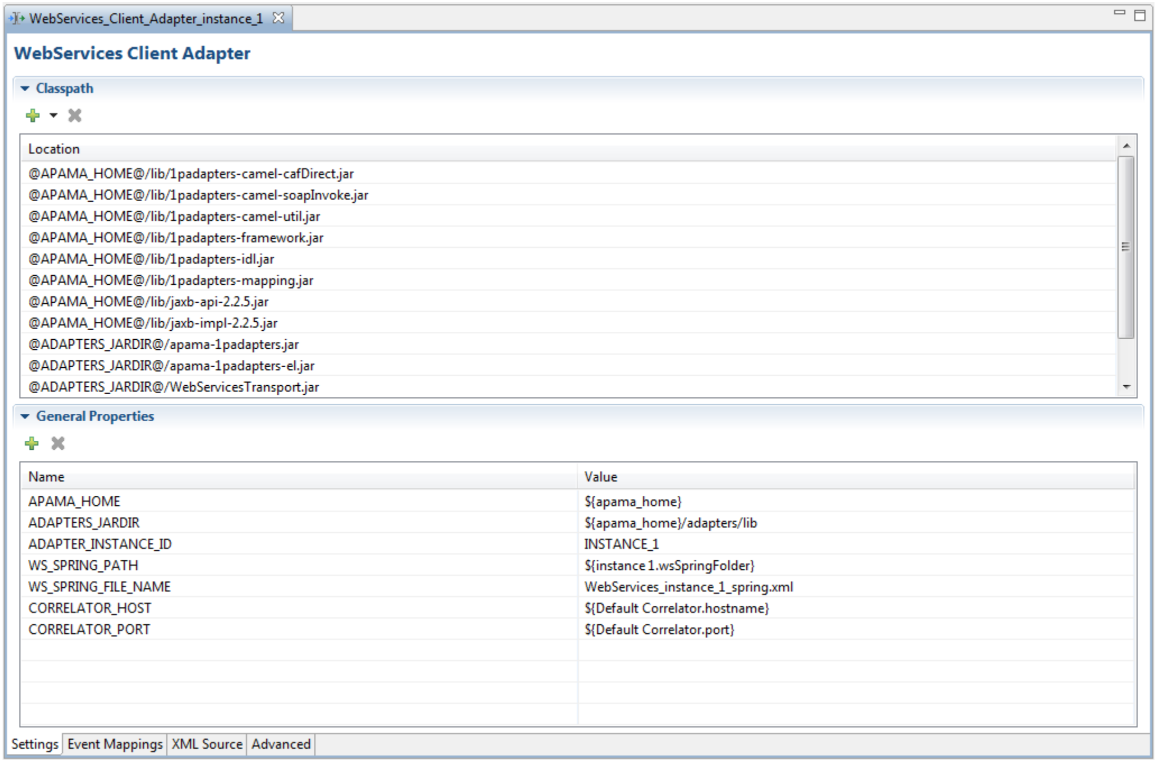
The Apama Web Services Client Adapter

Specify the Web Service and operation to use

After you add an instance of the Web Services Client adapter to a project you need to specify the Web Service that the application will use and which Web Service operation the application will invoke.

To specify the Web Service to use:

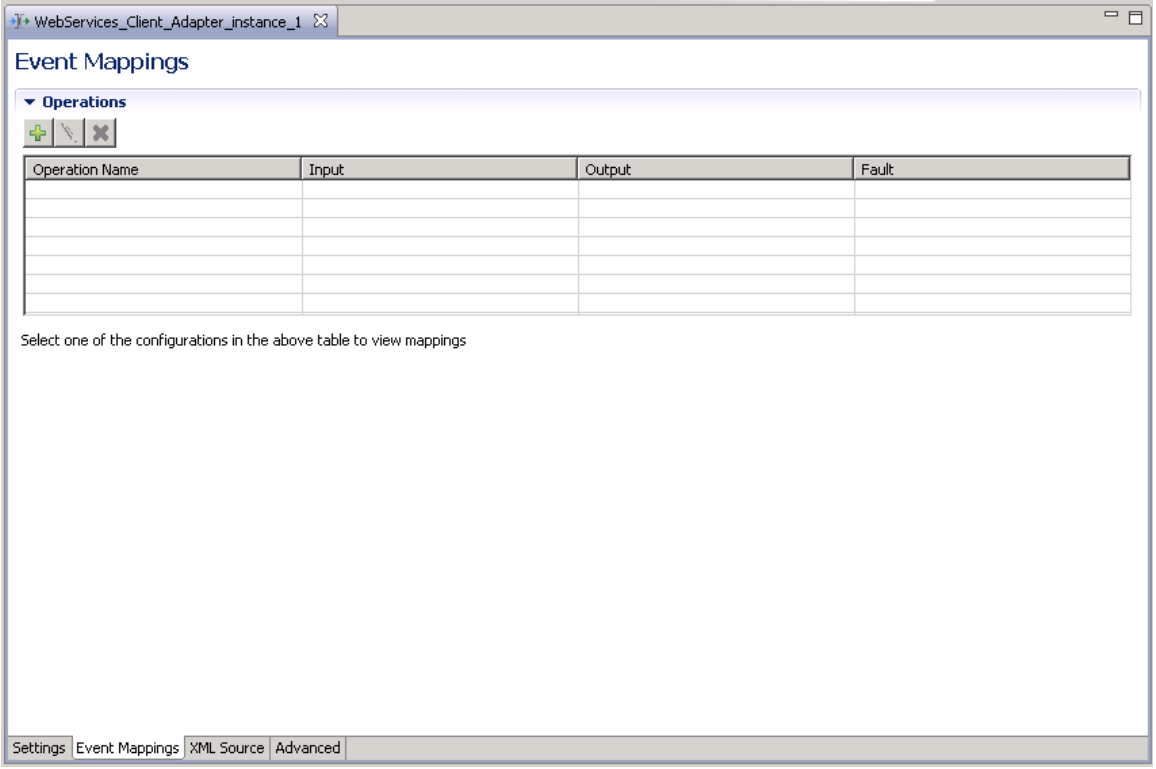
1. In the Project Explorer, expand the project's `Adapters` node and then expand the `WebServices Client Adapter` node.
2. Double-click the entry for the adapter instance you want to configure. This opens the adapter instance configuration in the Web Service adapter editor, showing `CLASSPATH` and other standard properties provided by the adapter's resource bundle.



Location
@APAMA_HOME@/lib/1padapters-camel-cafDirect.jar
@APAMA_HOME@/lib/1padapters-camel-soapInvoke.jar
@APAMA_HOME@/lib/1padapters-camel-util.jar
@APAMA_HOME@/lib/1padapters-framework.jar
@APAMA_HOME@/lib/1padapters-idl.jar
@APAMA_HOME@/lib/1padapters-mapping.jar
@APAMA_HOME@/lib/jaxb-api-2.2.5.jar
@APAMA_HOME@/lib/jaxb-impl-2.2.5.jar
@ADAPTERS_JARDIR@/apama-1padapters.jar
@ADAPTERS_JARDIR@/apama-1padapters-el.jar
@ADAPTERS_JARDIR@/WebServicesTransport.jar

Name	Value
APAMA_HOME	${apama_home}
ADAPTERS_JARDIR	${apama_home}/adapters/lib
ADAPTER_INSTANCE_ID	INSTANCE_1
WS_SPRING_PATH	${instance 1.wsSpringFolder}
WS_SPRING_FILE_NAME	WebServices_instance_1_spring.xml
CORRELATOR_HOST	${Default Correlator.hostname}
CORRELATOR_PORT	${Default Correlator.port}

3. In the Web Service adapter editor, select the Event Mappings tab.



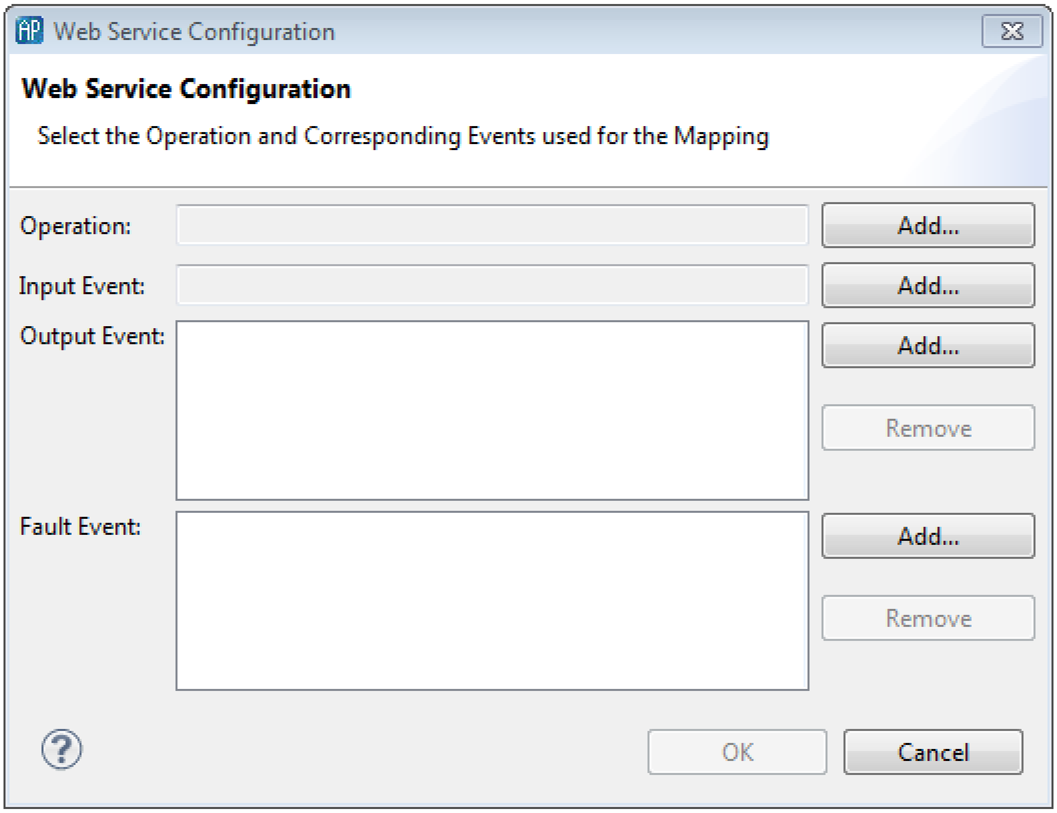
Event Mappings

Operations

Operation Name	Input	Output	Fault

Select one of the configurations in the above table to view mappings

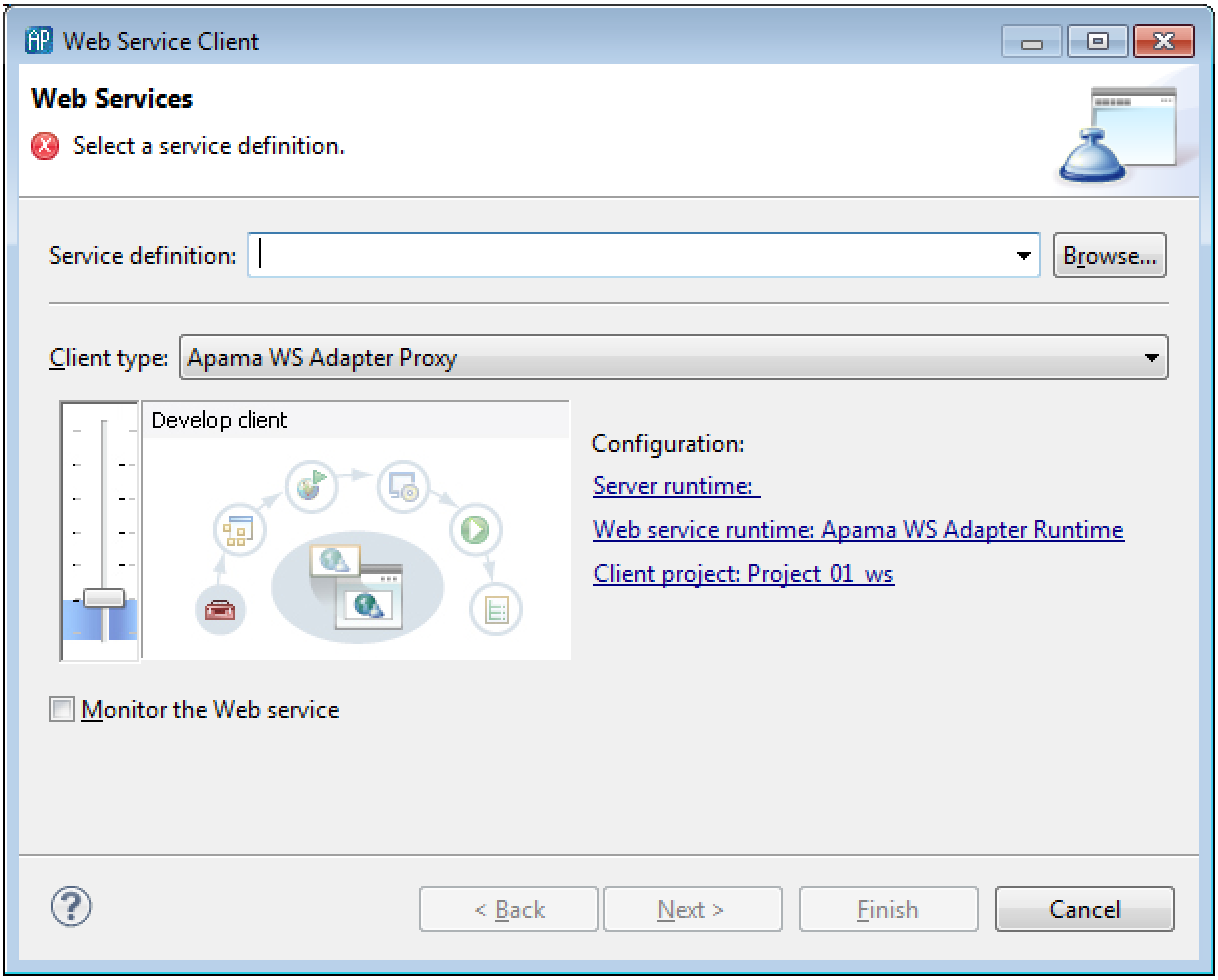
4. On the adapter editor's Event Mappings tab, click the Add button (+). This displays the **Web Service Configuration** dialog.



Web Service Configuration

Select the Operation and Corresponding Events used for the Mapping

5. In the **Web Service Configuration** dialog, to the right of the Operation field, click the Add button and select Create New. This displays the **Web Service Client** wizard.



Web Service Client

Web Services

Select a service definition.

Service definition:

Client type: Apama WS Adapter Proxy

Configuration:

Server runtime:

Web service runtime: Apama WS Adapter Runtime

Client project: Project_01_ws

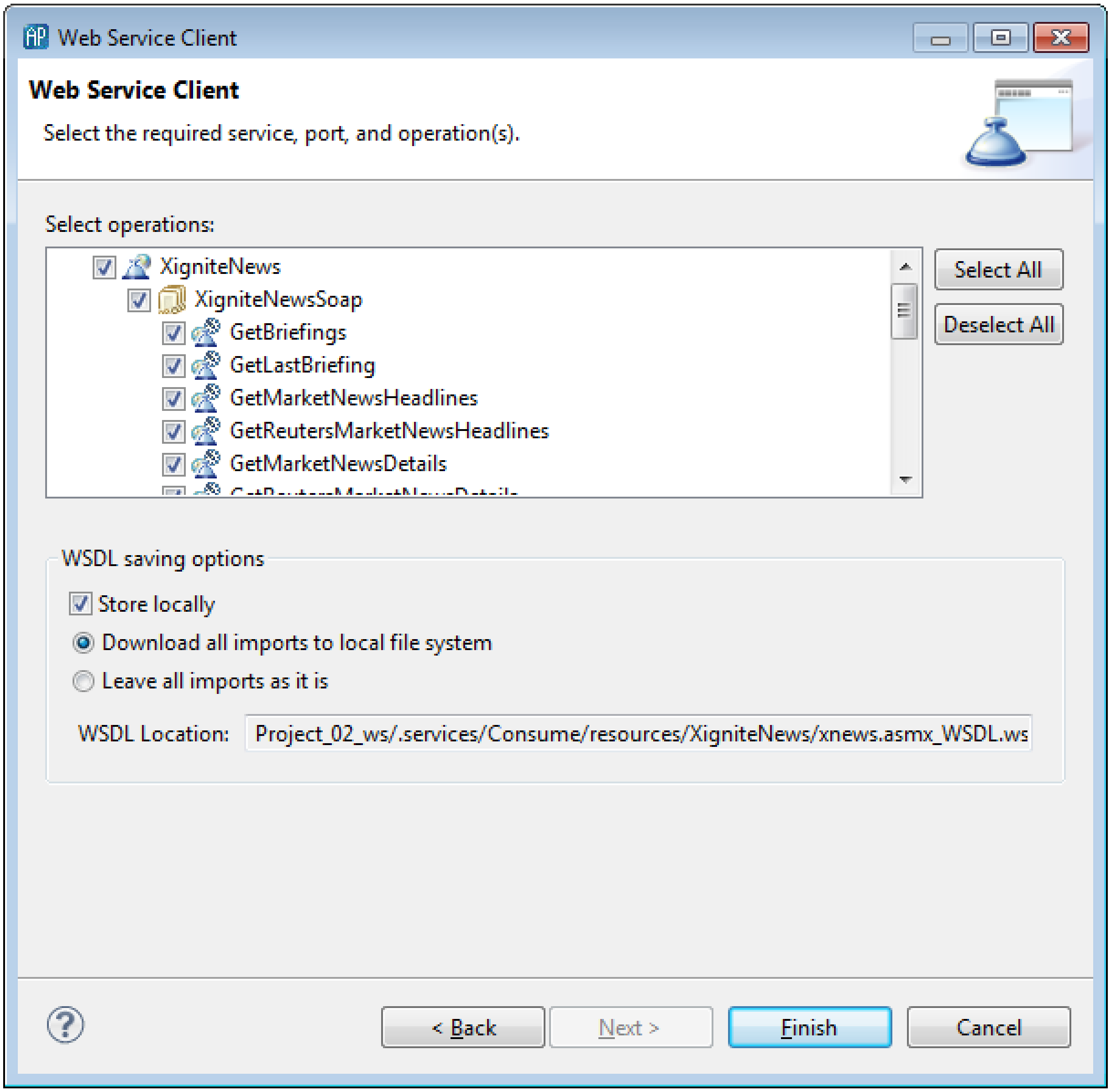
☐ Monitor the Web service

Note, if you already configured a Web Service in your project, when you click the Add button, you can select Choose from existing instead; for more information, see "Editing Web Services Client adapter configurations" on page 238.

6. In the **Web Service Client** wizard, in the Service definition field, specify a valid URL for the Web Service Definition File (WSDL) that defines the Web Service. This can be a local file or a file at a remote location. Be sure to use the complete syntax when specifying the location for the WSDL file including `file:///` or `http://`. Note, if you click the Browse button, you can select a WSDL file located in your Workspace.

7. In the Client type field, make sure that `Apama Adapters Proxy` is selected from the drop-down list (this is the default).

8. Click Next. The second page of the **Web Service Client** wizard opens, showing all the operations that are defined in the WSDL file. The operations are shown with SOAP and SOAP 1.2 bindings depending on how they are defined in the WSDL file.

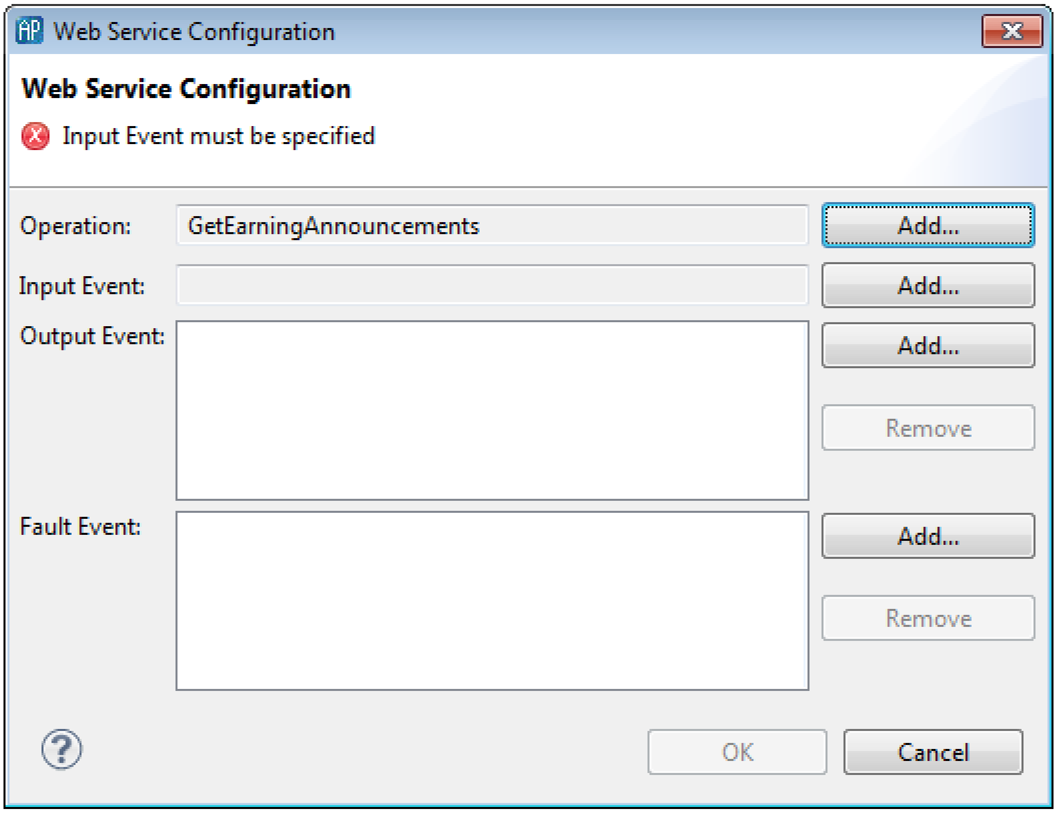

Web Service Client
Web Service Client
Select the required service, port, and operation(s).
Select operations:
☑ XigniteNews
☑ XigniteNewsSoap
☑ GetBriefings
☑ GetLastBriefing
☑ GetMarketNewsHeadlines
☑ GetReutersMarketNewsHeadlines
☑ GetMarketNewsDetails
Select All
Deselect All
WSDL saving options
☑ Store locally
● Download all imports to local file system
○ Leave all imports as it is
WSDL Location: Project_02_ws/.services/Consume/resources/XigniteNews/xnews.asmx_WSDL.ws
< Back
Next >
Finish
Cancel

If you want your application to invoke all operations, click Finish instead of Next and skip the next two steps. You are done with this part of the configuration.

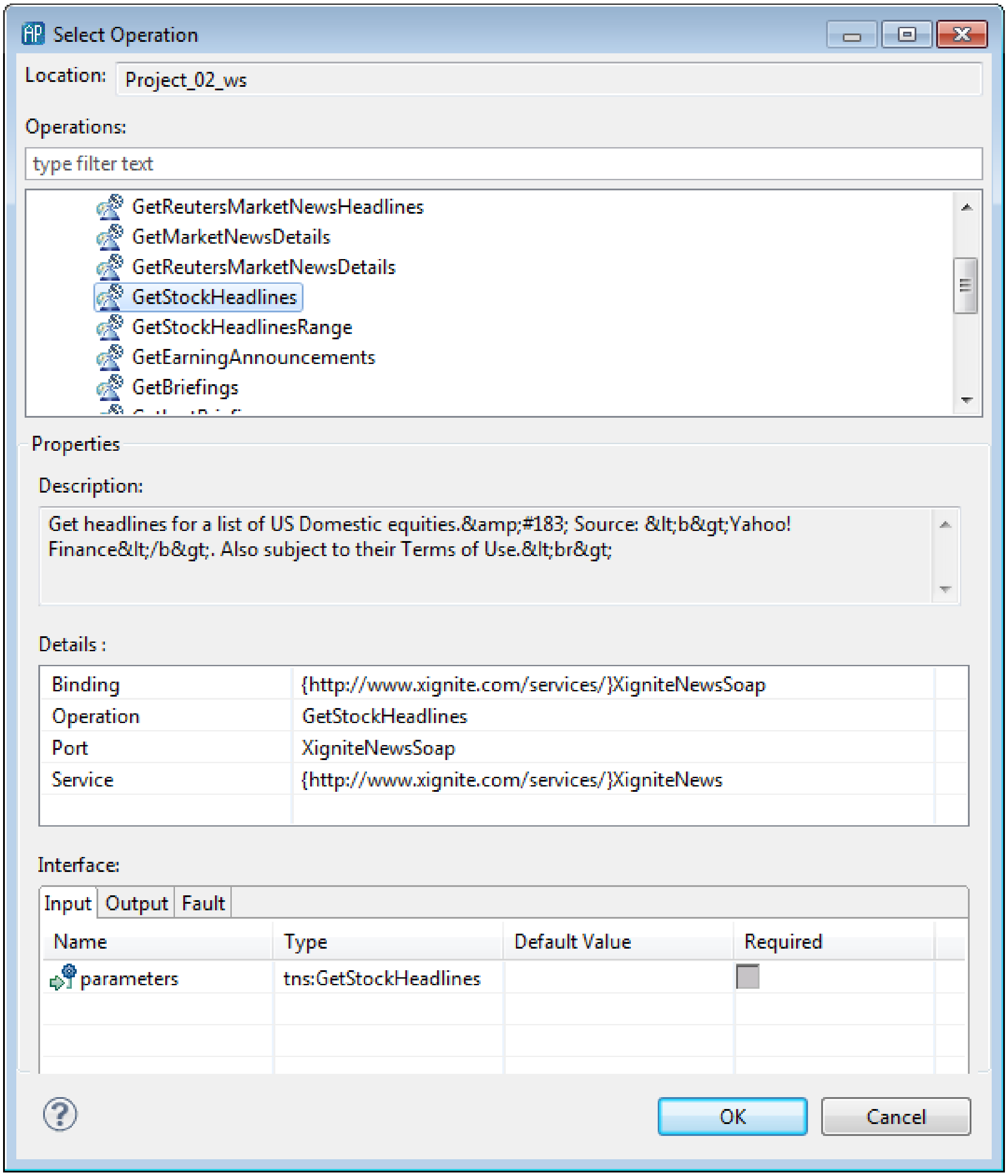
9. In the second page of the **Web Service Client** wizard, select the service operation(s) you want your application to invoke.

10. Click Finish.

    If you selected one operation the wizard adds it to the Operation field of the **Web Service Configuration** dialog.



If you selected more than one operation the wizard displays the Select Operation dialog, which lets you specify the Apama events that will be mapped to Web Service messages.



Location: Project_02_ws

GetReutersMarketNewsHeadlines
GetMarketNewsDetails
GetReutersMarketNewsDetails
GetStockHeadlines
GetStockHeadlinesRange
GetEarningAnnouncements
GetBriefings

Description:

Get headlines for a list of US Domestic equities.&#183; Source: <b>Yahoo! Finance</b>. Also subject to their Terms of Use.

Details :

Binding	{http://www.xignite.com/services/}XigniteNewsSoap
Operation	GetStockHeadlines
Port	XigniteNewsSoap
Service	{http://www.xignite.com/services/}XigniteNews

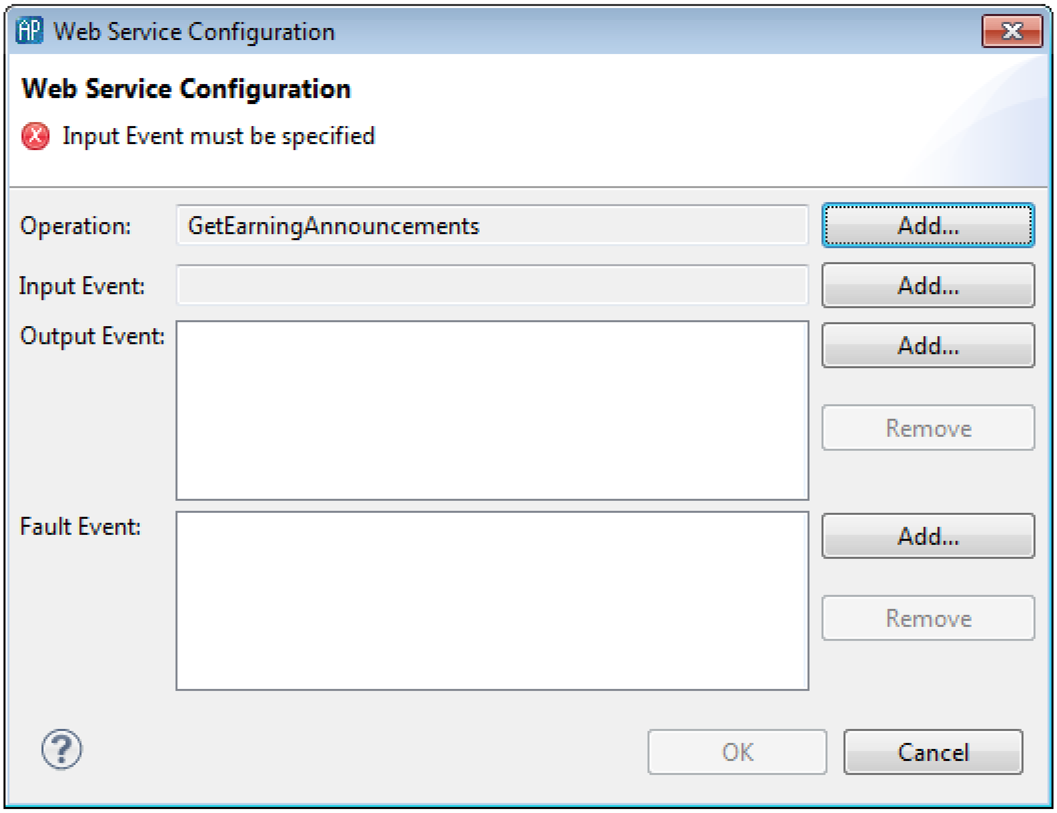
Name	Type	Default Value	Required
parameters	tns:GetStockHeadlines		□

Whether you selected one operation or multiple operations, the next step is to specify the Apama events that will be mapped to Web Service messages; for more information, see "Specifying Apama events for mapping" on page 235.

Configuring a Web Services Client adapter

Specifying Apama events for mapping

After you specify the Web Service operation your application will invoke, you need to indicate the Apama events with which your application will interact with the messages used by the Web Service operation.



Web Service Configuration

Web Service Configuration

Input Event must be specified

Operation: GetEarningAnnouncements

Input Event:

Output Event:

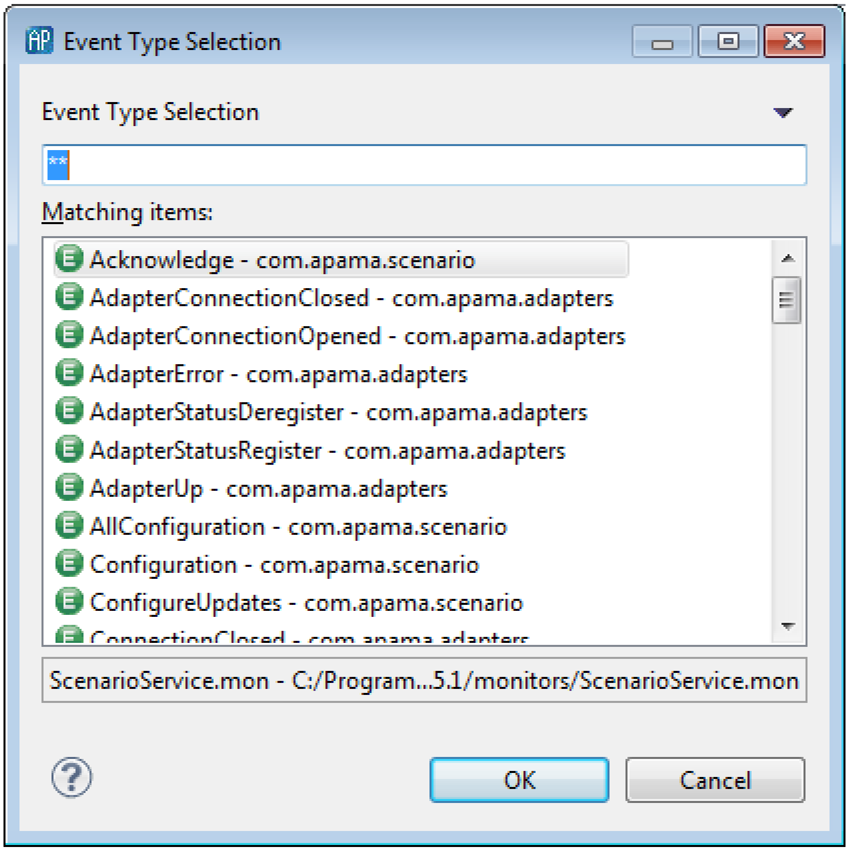
Fault Event:

For each operation that you add:

- You must specify an input event. You will map the fields in this input event to the parameters in the input message required to invoke the operation. An Apama application sends an input event to invoke a Web Service operation. You can add the same operation to the adapter more than once. Each time you add the same operation you can specify a different input event or the same input event.
- You can optionally specify output events. If you specify output events, you will map the parameters of the Web Service operation response message to the fields of the events. An Apama application receives an output event when a Web Service sends a response message as the result of an operation request.
- You can optionally specify fault events. If you specify fault events, you will map the parameters of the error message the Web Service might send to the fields of the events. An Apama application receives a fault event if there is an error during invocation of a Web Service operation.

To specify Apama events for mapping:

1. In the **Web Service Configuration** dialog, to the right of the Input Event field, click the Browse button. This displays the **Event Type Selection** dialog.



**

Acknowledge - com.apama.scenario
AdapterConnectionClosed - com.apama.adapters
AdapterConnectionOpened - com.apama.adapters
AdapterError - com.apama.adapters
AdapterStatusDeregister - com.apama.adapters
AdapterStatusRegister - com.apama.adapters
AdapterUp - com.apama.adapters
AllConfiguration - com.apama.scenario
Configuration - com.apama.scenario
ConfigureUpdates - com.apama.scenario
ConnectionClosed - com.apama.adapters

ScenarioService.mon - C:/Program...5.1/monitors/ScenarioService.mon

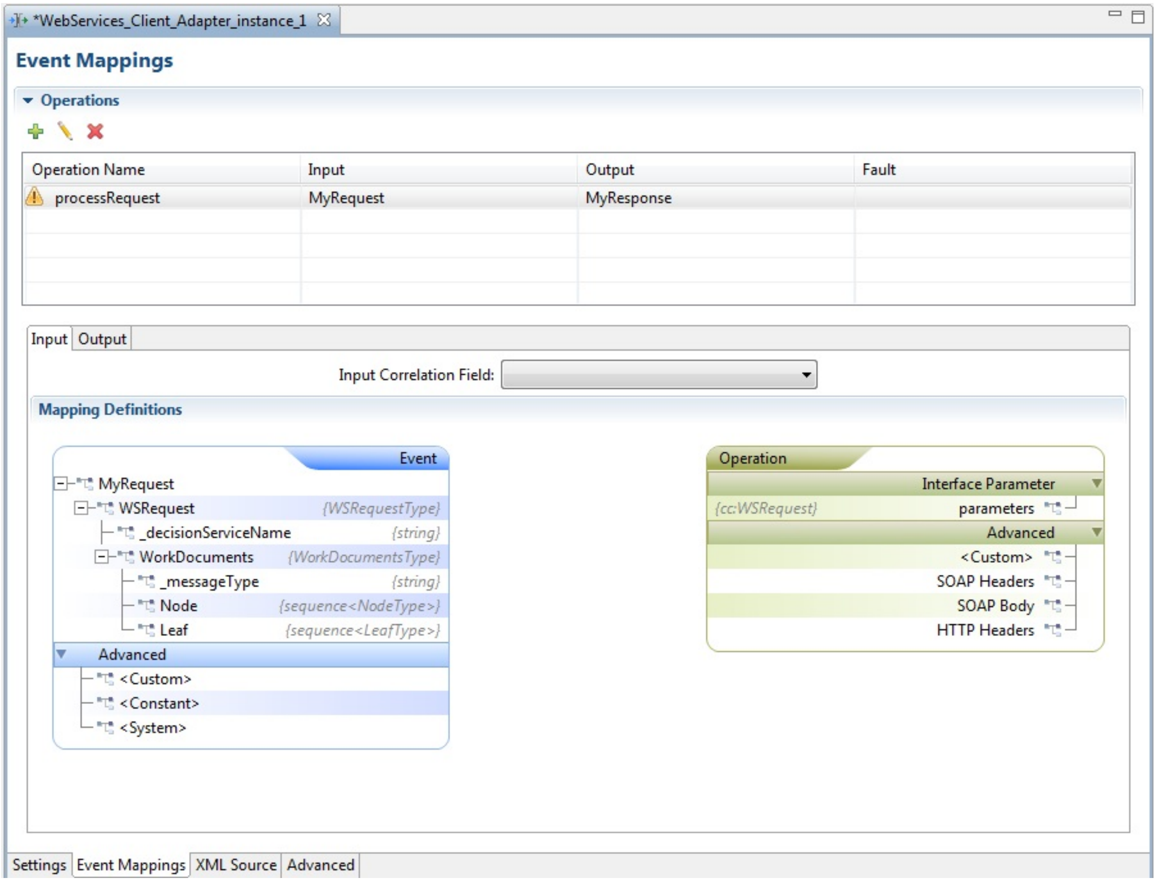
a. In the **Event Type Selection** dialog's Event Type Selection field, enter the name of the event. As you type, event types that match what you enter are shown in the Matching Items list.

b. In the Matching Items list, select the name of the event type you want to use as the Input Event. The name of the EPL file that defines the selected event is displayed in the status area at the bottom of the dialog.

c. Click OK. The **Web Service Configuration** dialog is again displayed, showing the event you selected in the Input Event field

2. If your application will use output messages or fault messages from the Web Service, specify the Apama events that will be associated with those message types in the Output Event and Fault Event fields of the **Web Service Configuration** dialog. For output and fault messages, you can add multiple Apama event types to those fields by selecting the event type and clicking Add.

3. In the **Web Service Configuration** dialog, click OK. Information about the specified Web Service operations and the associated Apama events is displayed in the Operations section of the adapter editor. In the Mapping Definitions section, the editor displays an Input tab and, if you have specified output events or fault events, Output or Fault tabs.



Operation Name	Input	Output	Fault
processRequest	MyRequest	MyResponse	

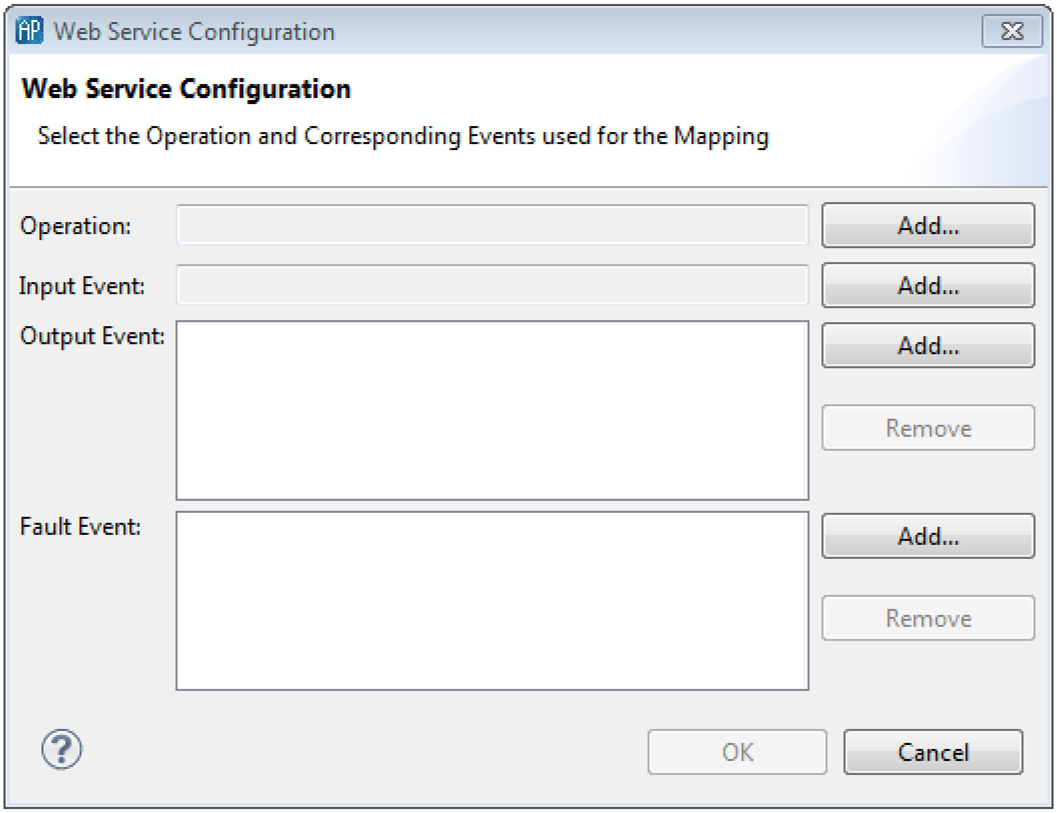
The next step is to map the fields of the Apama events to the parameters in the Web Service operations; for more information, see "Mapping Web Service message parameters" on page 242.

Configuring a Web Services Client adapter

Editing Web Services Client adapter configurations

If you have already configured an instance of the Web Services Client adapter and you want to configure another operation for the application to invoke, add the operation to the adapter configuration as follows:

1. In the Project Explorer, expand the project’s `Adapters` node and then expand the `WebServices Client Adapter` node.
2. Double-click the entry for the adapter instance you want to configure. This opens the adapter instance configuration in the Web Service adapter editor.
3. In the Web Service adapter editor, select the Event Mappings tab.
4. On the adapter editor's Event Mappings tab, click the Add button (✚). This displays a blank **Web Service Configuration** dialog.



Web Service Configuration

Web Service Configuration

Select the Operation and Corresponding Events used for the Mapping

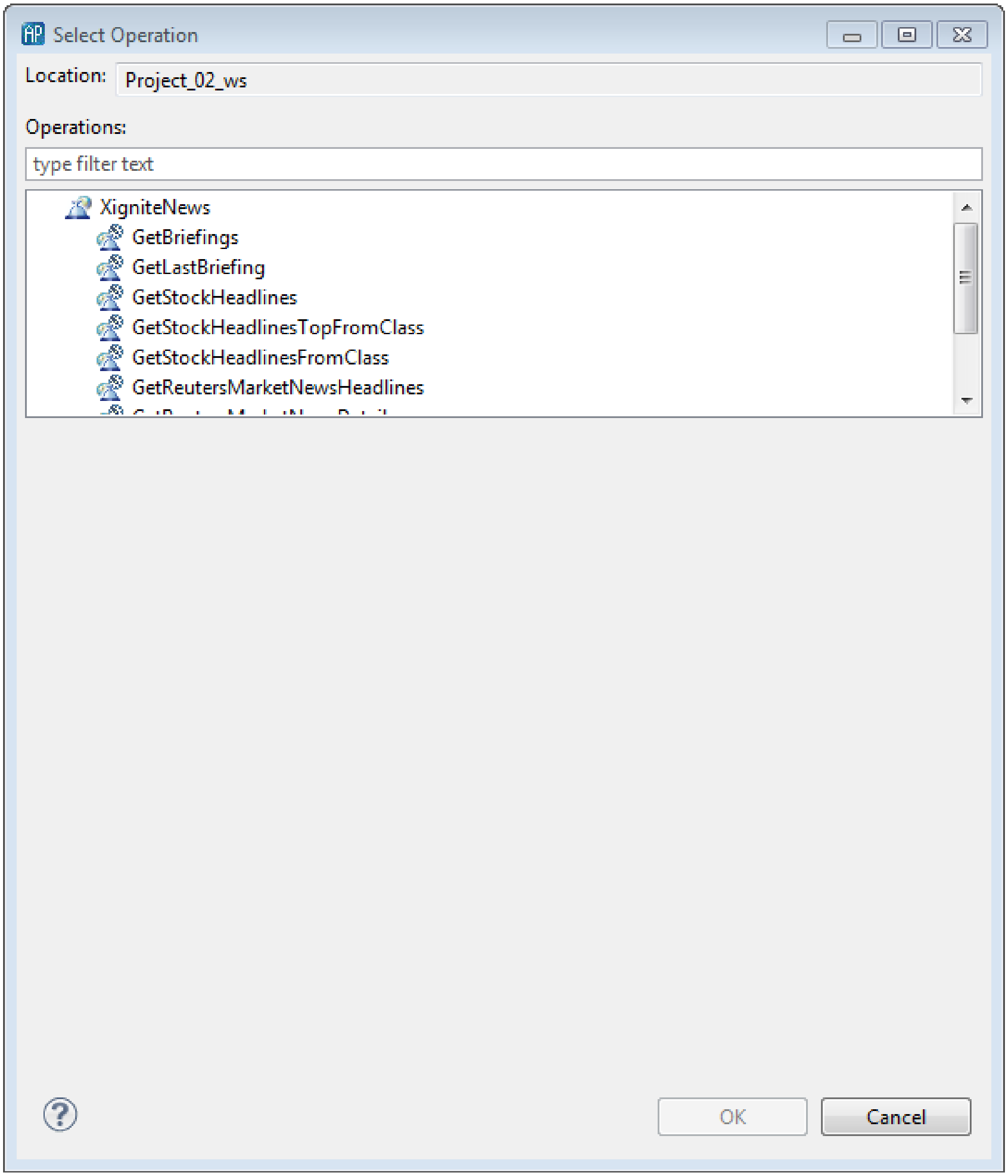
Operation:

Input Event:

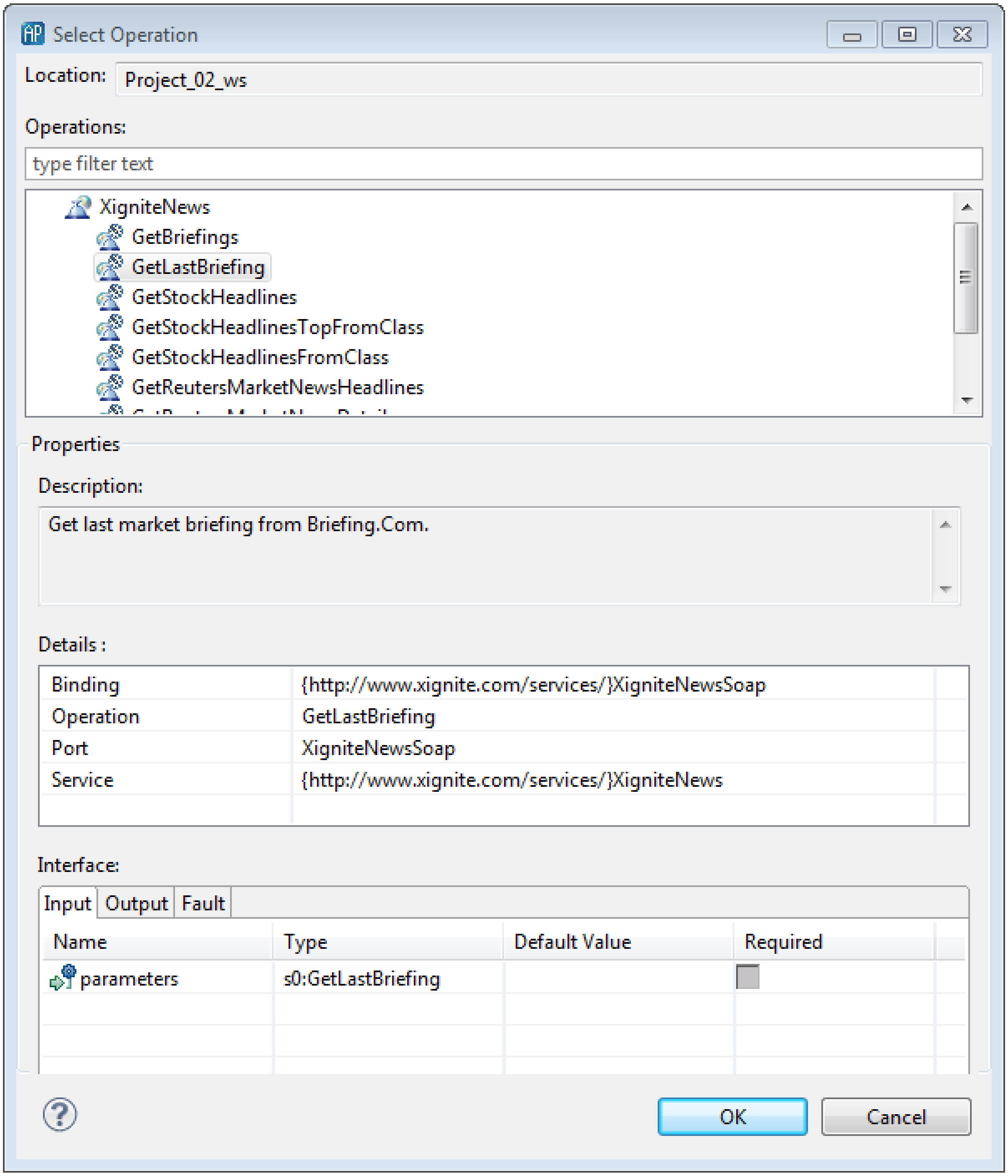
Output Event:

Fault Event:

5. In the **Web Service Configuration** dialog, to the right of the Operation field, click the Add button and select Choose from existing. This displays the **Select Operation** dialog.



6. In the **Select Operation** dialog, in the Operations field, select the operation to invoke. The display in the dialog is updated to show the properties associated with the operation.



Select Operation

Location: Project_02_ws

Operations:

type filter text

- XigniteNews
  - GetBriefings
  - GetLastBriefing
  - GetStockHeadlines
  - GetStockHeadlinesTopFromClass
  - GetStockHeadlinesFromClass
  - GetReutersMarketNewsHeadlines

Properties

Description:

Get last market briefing from Briefing.Com.

Details :

Binding	{http://www.xignite.com/services/}XigniteNewsSoap
Operation	GetLastBriefing
Port	XigniteNewsSoap
Service	{http://www.xignite.com/services/}XigniteNews

Interface:

Input | Output | Fault

Name	Type	Default Value	Required
parameters	s0:GetLastBriefing		□

7. In the **Select Operation** dialog, click OK.

   The operation you selected is added to the Operations field of the **Web Service Configuration** dialog.

8. Specify the Apama events to use to interact with the operation's Input messages and, if desired, Output and Fault messages as described in "Specifying Apama events for mapping" on page 235.

The Apama Web Services Client Adapter

Adding multiple instances of the Web Services Client adapter

You can add multiple Web Services Client adapter instances to an Apama project. Apama Studio generates service monitors and configuration files that are specific to each adapter instance. Different instances can be used, for example, to invoke different Web Service operations and map different Apama events to Web Service input, output, and fault messages. The generated service monitors contain event listeners for the events used in the mappings of the specific adapter instance. If the same event is used in as an input event for multiple adapter instances, multiple events will be emitted to the corresponding channels of each adapter.

To add another adapter instance to a project:

1. In the Project Explorer, expand the project's `Adapters` node.
2. Right-click the `WebServices Client Adapter` node and select Add Instance from the pop-up menu. The **Add Instance** dialog opens.
3. In the **Add Instance** dialog, accept the default adapter instance name or give it a new one and click OK. The instance is added to the `WebServices Client Adapter` node.

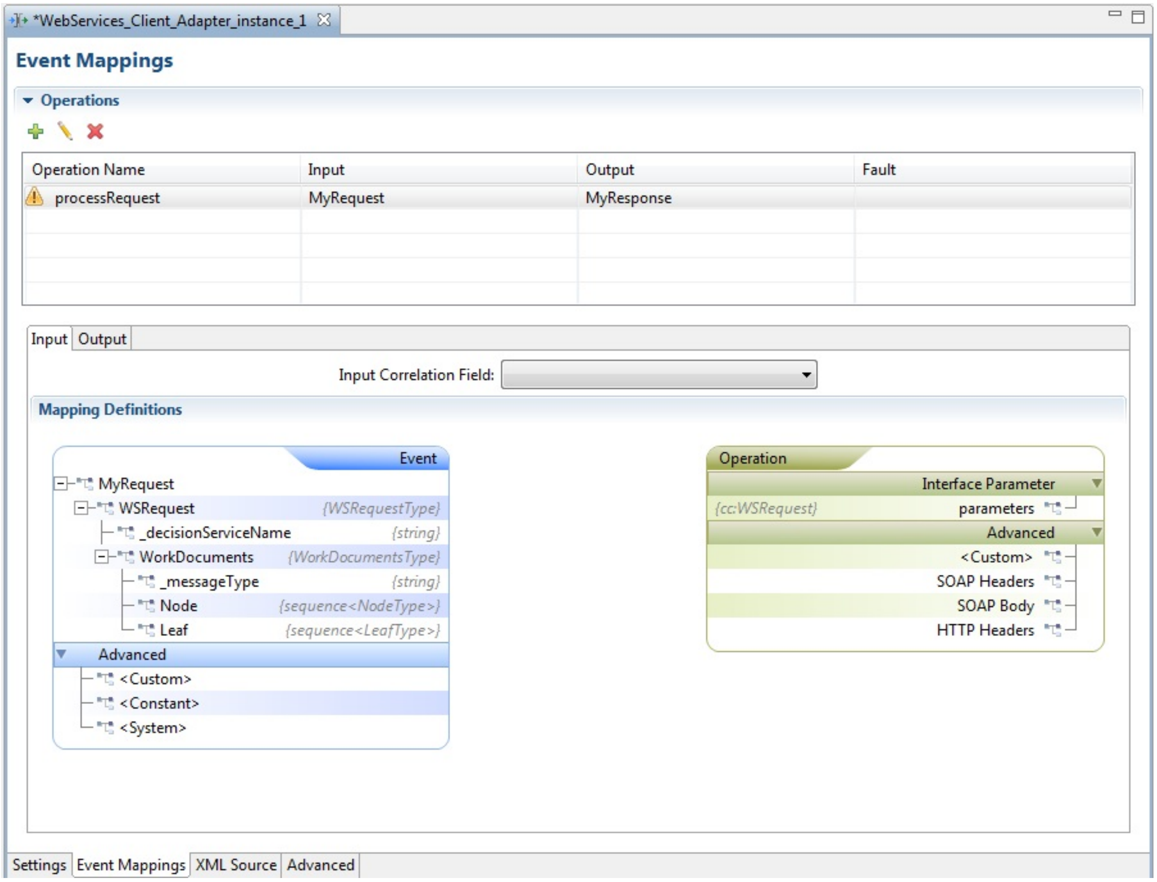
The Apama Web Services Client Adapter

Mapping Web Service message parameters

After you specify which Apama events you want to map to the Web Service messages, you need to create mapping rules that associate Apama event fields with parameters in the Web Service messages. The Apama Studio adapter editor provides a visual mapping tool to create the mapping rules. Web Service messages fall into three categories, each of which can be mapped to Apama events.

- Input mapping specifies how fields in an Apama event are connected to the parameters in a Web Service input message. These are also known as request messages.
- Output mapping specifies how parameters in a Web Service output message are connected to the fields in an Apama event. These are also known as response messages.
- Fault mapping specifies how Apama event fields and Web service parameters are connected when a fault message is delivered.

When you select an operation in the adapter editor's Event Mappings tab, in the Operations Name column, the Mapping Definitions section of the editor displays hierarchical representations of the Apama event and the Web Service operation. In Apama Studio, the source tree is on the left and the target on the right. For input event messages, the Apama event is the source and the Web Service message is the target; for output and fault messages the Web Service is the source and the Apama event is the target.



*WebServices_Client_Adapter_instance_1

Event Mappings

Operations

Operation Name	Input	Output	Fault
processRequest	MyRequest	MyResponse	

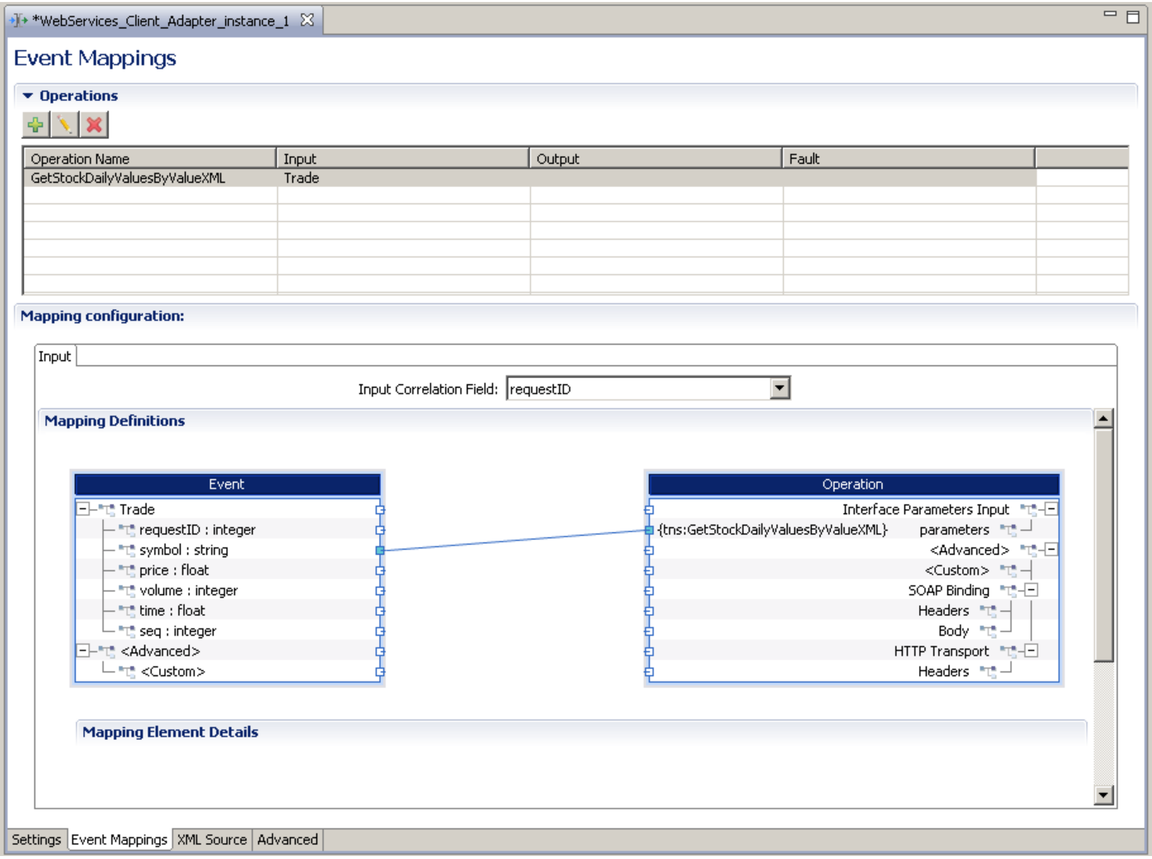
Input | Output

Input Correlation Field:

Mapping Definitions

There is a tab for each type of Apama event specified: input, output, and fault events. You must specify an input event for each operation that you add. You can add the same operation more than once. If you do, you can specify a different input event or the same input event for each additional instance of the operation. Optionally, you can specify one output event and/or one fault event for each operation instance that you add.

On the adapter editor's Event Mappings tab, in the Mapping Definitions section, you specify the mapping rules by clicking on an entity in one tree and dragging a line to the entity in the other tree.



Operation Name	Input	Output	Fault
GetStockDailyValuesByValueXML	Trade		

Input Correlation Field: requestID

In the Mapping Definitions section, for input messages, the Apama event fields are displayed to the left and the Web Services operation parameters on the right. To create a mapping rule, click on the event field and drag a line to the desired operation parameter.

There are several approaches for how to map Apama event fields to the parameters in Web Services messages, depending on the complexity of the events and the message parameters.

- Simple - Use this approach when a simple Apama event field can be associated with a corresponding type in the Web Service message. A simple Apama event field is of type `integer`, `float`, `boolean`, `decimal`, or `string`. For example, you can use the simple approach to map a `string` event field to a message parameter of type `xsd:string`.
- Convention-based - Use this approach when the structure of an Apama event corresponds directly with the XML structure of the Web Service message. The Apama event must have been defined by following the conventions described in "Convention-based XML mapping" on page 286. In this case, Apama (at runtime) automatically converts the event instance to the request XML structure of the Web Service, and also converts the Web Service response XML to an event instance.
- Template-based - Use this approach when the XML definition of the Web Service message contains a complex type. You must supply an XML template file that defines the complex type. In the template file, you use variables that will be replaced with values from the Apama event fields. You then map event fields to the variables in the template.
- Combination - This approach combines the convention-based and template-based approaches. Use this approach when the adapter can automatically convert at least one event field (`event` or `sequence` type) to XML that models a message parameter and when at least one of the values of the converted fields serves as a value for a variable in a template.

You can also specify if a mapping rule requires a an XPath or XSLT transformation. For more information on how to specify a transformation type, see "Specifying transformation types" on page 256.

The visual mapping tool allows you to add custom entries to the SOAP Headers, HTTP Headers, and SOAP Body if the Web Service requires it. For more information on customizing mapping rules, see "Customizing mapping rules" on page 257.

When you save a Web Service configuration, Apama Studio generates the XML files used to interact with the Web Services and the appropriate service monitors. See "Web Services Client adapter artifacts" on page 268 for more information.

The Apama Web Services Client Adapter

Simple mapping

When creating a rule for mapping an Apama event field that contains a simple type (`integer`, `float`, `decimal`, `boolean`, or `string`) to a Web Services parameter that contains a similar type, you can drag a line between the elements as follows:

1. In the Web Services Client adapter editor, display the **Event Mapping** tab.
2. For each mapping rule, click on the entity you want to map and drag a line to the entity you want to map it to. You must ensure that the types of the two entities match. For example, an Apama `string` type field must map to an XML `xsd:string` field.

Apama Studio represents each rule with a blue line between entities.

Mapping Web Service message parameters

Convention-based XML mapping

Convention-based mapping allows XML documents to be created or parsed, based on a document structure encoded in the definition of the source or target Apama event type.

The first stage when using convention-based mapping is to examine the structure of the XML document, and create an event definition to represent its root element, with fields for each attribute, text node, sub-element or sequence (of attributes, text nodes or sub-elements). The actual names of the event types are not important, but the event field names and types must follow the following conventions:

- XML attributes can be represented by any EPL simple type such as `string`, `integer`, etc. The name used should be preceded by an underscore,for example `boolean _flag;`.
- XML text nodes are represented by either:
  - A field inside an Apama event representing the parent of the element containing the text, named after the element that encloses the text such as `string myelement;`. This avoids the need to create an event type to represent the element in cases where the element only contains a text node, and no attributes or children. The field type may be any primitive EPL type (for example `string`, `integer`, etc.).
  - A field inside an Apama event representing the element that directly contains the text, named `xmlTextNode`. This is necessary in cases where an Apama event type is needed to represent the

element so that attributes and/or child elements can also be mapped. The field type may be any primitive EPL type (for example `string`, `integer`, etc.).

- XML elements containing attributes or sub-elements of interest are represented by a field of an event type which follows these same conventions. The event type can have any name, but the field must be named after the element, for example, `MyElementEventType myelement`.
- XML attributes, text nodes or elements which may occur more than once in the document are represented by a sequence field of the appropriate primitive or event type, named after the element, for example, `sequence<string> myelement` or `sequence<MyElement> myelement`.

Some special cases to be aware of when naming fields to match element/attribute names are:

- XML nodes which are inside an XML namespace are always referenced by their local name only (the namespace or namespace prefix is ignored).
- XML node names that are Apama EPL keywords (such as `<return>`) must be escaped in the event definition using a hash character, for example, `string #return;`. When generating an XML document, each field in the event will be processed in order and used to build up the output document. When parsing an XML document, each field in the event will be populated with whatever XML content matches the field name and type (based on the conventions above); any XML content that is not referenced in the event definition will be silently ignored.
- XML node names containing any character that is not a valid EPL identifier character (anything other than than a-z, A-Z, 0-9 and _) must be represented using a $hexcode escape sequence. Of the characters that are not valid EPL identifier characters, only the hyphen and dot are supported. Note that the hexcode based escape sequences are case sensitive. For representing the hyphen or dot use the following:
  - Hyphen '-' is represented as `$002d`.
  - Dot '.' is represented as `$002e`.

Limitations of convention-based XML mapping

In this release it is not possible to generate documents that contain elements in different XML namespaces (although when parsing this is not a problem).

The following limitations apply to the Apama event definitions that can be used to generate XML:

- Dictionary event field types are not supported
- If an event field is of type `sequence`, the sequence can contain simple types or events. The sequence cannot contain sequences of sequences or sequences of dictionaries

Mapping Web Service message parameters

Convention-based Web Service message mapping example

As an example of using convention-based mapping, consider the following XML documents, which define a request message and a response message:

```
<WSRequest decisionServiceName="dsName"
  xmlns="urn:WSService">
  <WorkDocuments messageType="FLAT">
    <Node id="idAttrValue1">
      <id>id1</id>
      <isLeaf>true</isLeaf>
      <isRoot>false</isRoot>
      <child href="href1"></child>
      <child href="href2"></child>
      <parent href="href1"></parent>
      <parent href="href2"></parent>
      <leaf href="href1"></leaf>
      <leaf href="href2"></leaf>
    </Node>
    <Node id="idAttrValue2">
      <id>id2</id>
      <isLeaf>true</isLeaf>
      <isRoot>false</isRoot>
      <child href="href1"></child>
      <child href="href2"></child>
      <parent href="href1"></parent>
      <parent href="href2"></parent>
      <leaf href="href1"></leaf>
      <leaf href="href2"></leaf>
    </Node>
    <Leaf id="idValue">
      <node href="hrefValue1"></node>
    </Leaf>
  </WorkDocuments>
</WSRequest>

<WSResponse decisionServiceName="dsName"
xmlns="urn:WSService">
  <WorkDocuments messageType="FLAT">
    <Node id="idAttrValue1">
      <id>id1</id>
      <isLeaf>true</isLeaf>
      <isRoot>false</isRoot>
      <child href="href1"></child>
      <child href="href2"></child>
      <parent href="href1"></parent>
      <parent href="href2"></parent>
      <leaf href="href1"></leaf>
      <leaf href="href2"></leaf>
    </Node>
    <Node id="idAttrValue2">
      <id>id2</id>
      <isLeaf>true</isLeaf>
      <isRoot>false</isRoot>
      <child href="href1"></child>
      <child href="href2"></child>
      <parent href="href1"></parent>
      <parent href="href2"></parent>
      <leaf href="href1"></leaf>
      <leaf href="href2"></leaf>
    </Node>
    <Leaf id="idValue">
      <node href="hrefValue1"></node>
    </Leaf>
  </WorkDocuments>
  <Messages version="versionX.Y">
    <Message>
      <severity>Info</severity>
      <text>text1</text>
      <entityReference href="erHref1"></entityReference>
    </Message>
    <Message>
      <severity>Info</severity>
      <text>text2</text>
      <entityReference href="erHref2"></entityReference>
    </Message>
  </Messages>
</WSResponse>
```

Following are event definitions that follow the conventions of these XML documents. It is important to understand that the EPL field types directly correspond to the types defined in the XML Schema document used by the request and response documents. Likewise, event field names directly correspond to the element names defined in the XML Schema document.

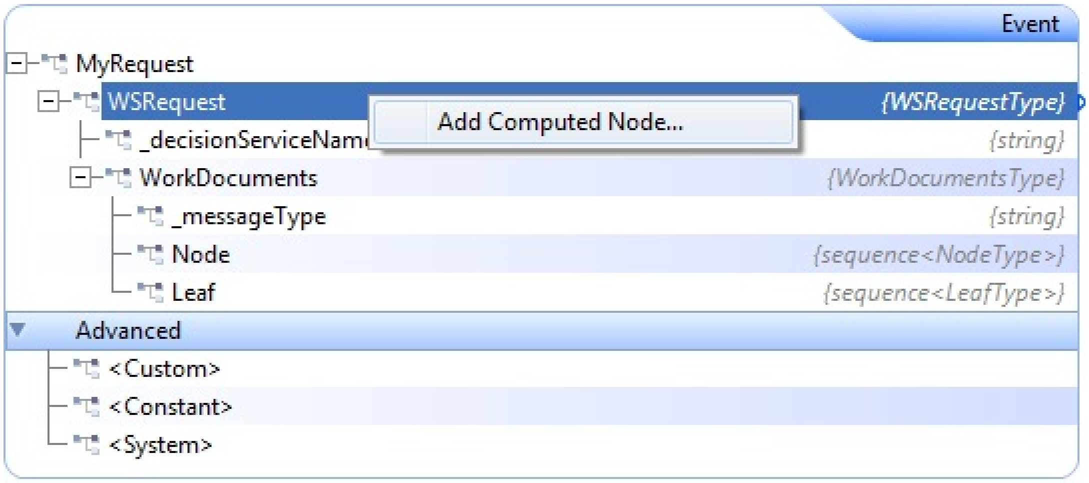
```
event NodeRef {
    string _href;
}
event NodeType {
    string  _id;
    string  id;
    boolean isLeaf;
    boolean isRoot;
    sequence<NodeRef> child;
    sequence<NodeRef> parent;
    sequence<NodeRef> leaf;
}
event LeafType {
    string  _id;
    NodeRef node;
}
event WorkDocumentsType {
    string _messageType;
    sequence< NodeType > Node;
    sequence< LeafType > Leaf;
}
event WSRequestType {
    string _decisionServiceName;
    WorkDocumentsType WorkDocuments;
}
event MessageType {
    string severity;
    string text;
    NodeRef entityReference;
}
event MessagesType {
    sequence<MessageType> Message;
    string _version;
}
event WSResponseType {
    string _decisionServiceName;
    WorkDocumentsType WorkDocuments;
    MessagesType Messages;
}
event MyRequest {
    WSRequestType WSRequest;
}
event MyResponse {
    WSResponseType WSResponse;
    }
```

To use the convention approach to mapping event fields to message parameters:

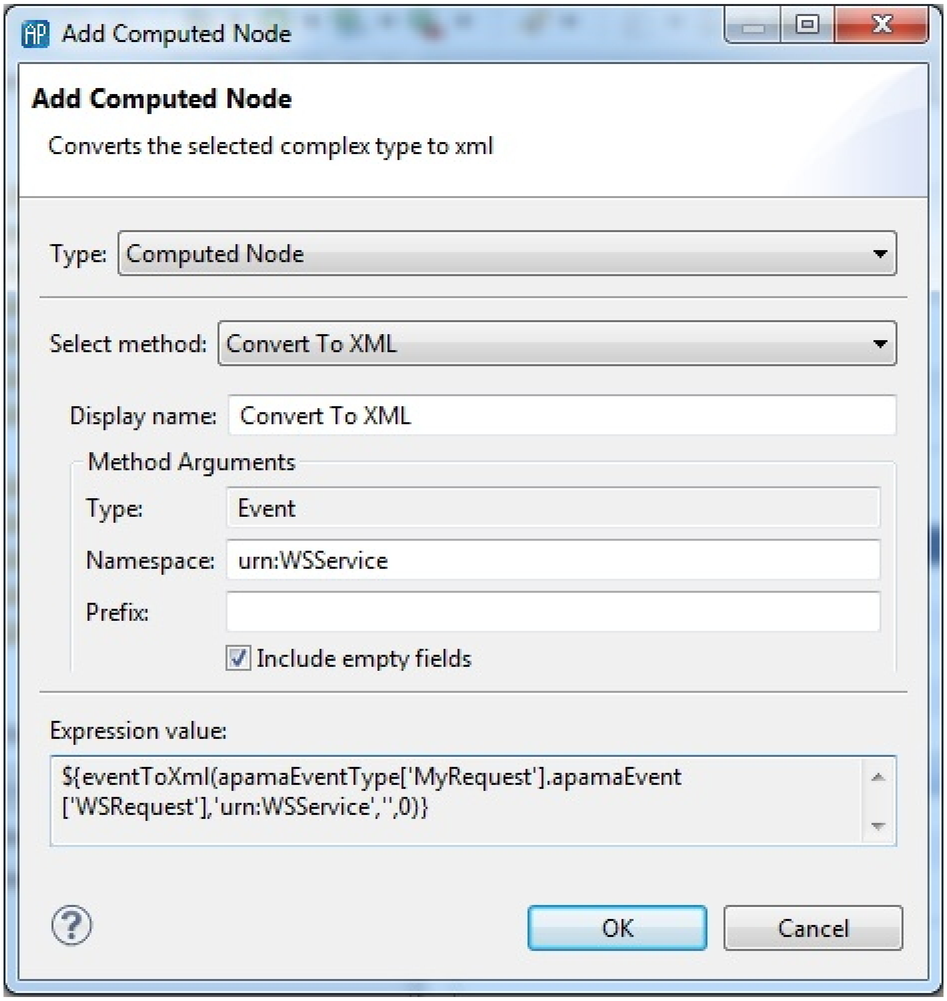
1. Create an Apama event with fields that correspond in type and order to the parameters of a Web Service operation.

2. Specify the event as the input, output, or fault event associated with a Web service operation. See "Specifying Apama events for mapping" on page 235.

3. With that operation selected, in the Event Mappings tab, right-click the Apama event and select Add Computed Node. For example, the following tree is in the Input tab:


Event
MyRequest
WSRequest {WSRequestType}
Add Computed Node...
_decisionServiceNam {string}
WorkDocuments {WorkDocumentsType}
_messageType {string}
Node {sequence<NodeType>}
Leaf {sequence<LeafType>}
Advanced
<Custom>
<Constant>
<System>

The **Add Computed Node** dialog appears:


Add Computed Node

Add Computed Node

Converts the selected complex type to xml

Type: Computed Node

Select method: Convert To XML

Display name: Convert To XML

Method Arguments

Type: Event

Namespace: urn:WSService

Prefix:

☑ Include empty fields

Expression value:

```
${eventToXml(apamaEventType['MyRequest'].apamaEvent
['WSRequest'],'urn:WSService','',0)}
```

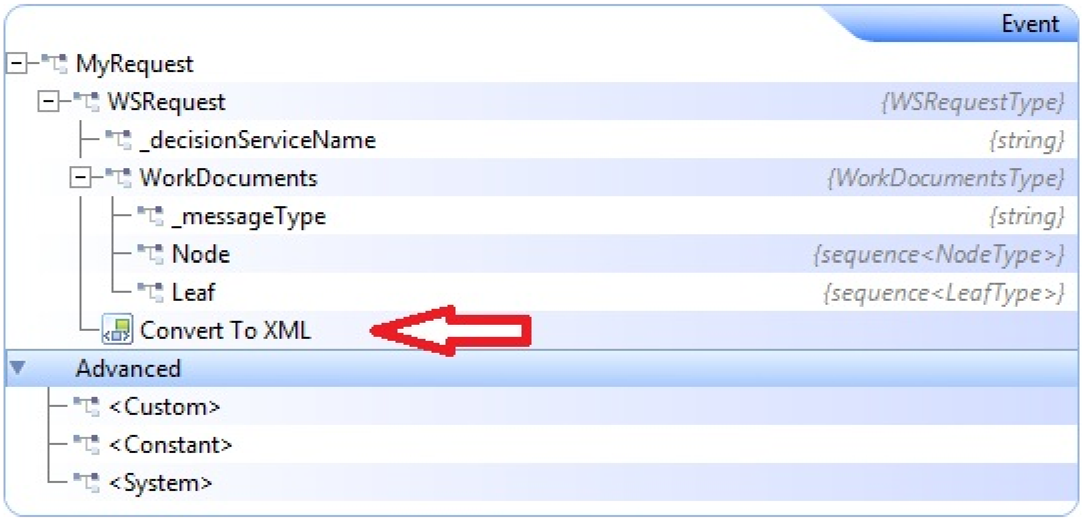
4. In the **Add Computed Node** dialog's Select Method field, select Convert to XML from the drop-down list. The dialog is updated to show more information.

   When the adapter generates the request XML structure you can customize the namespace and namespace prefix in the generated XML. If sub-elements in the request XML structure are in different namespaces, you cannot use only the convention mapping approach. You must combine it with the template approach. See "Combining convention and template mapping" on page 253. However, when sub-elements belong to more than one namespace you can use the convention approach without a template to convert the XML structure to an event.

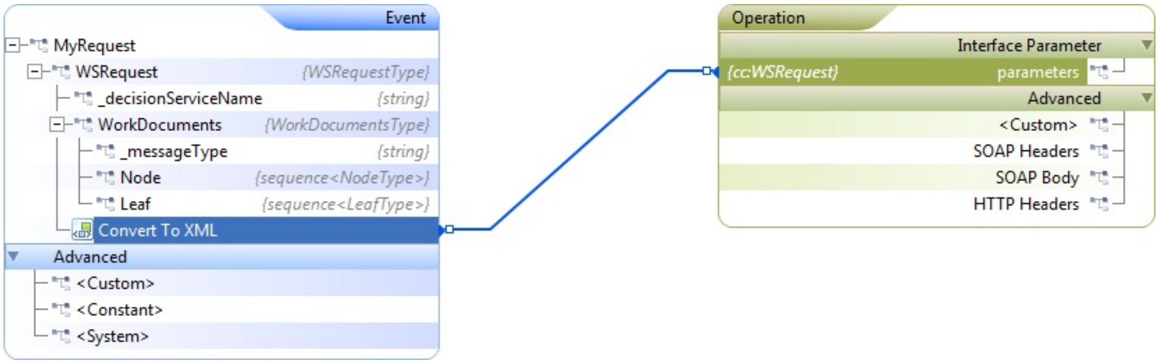
By default the Include empty fields option is enabled. This specifies that empty XML nodes will be generated when empty EPL string fields are encountered within an Apama event. This option does not affect empty strings within a sequence of EPL strings. If you clear the check box to disable the option, empty XML nodes will not be generated.

5. Click OK.

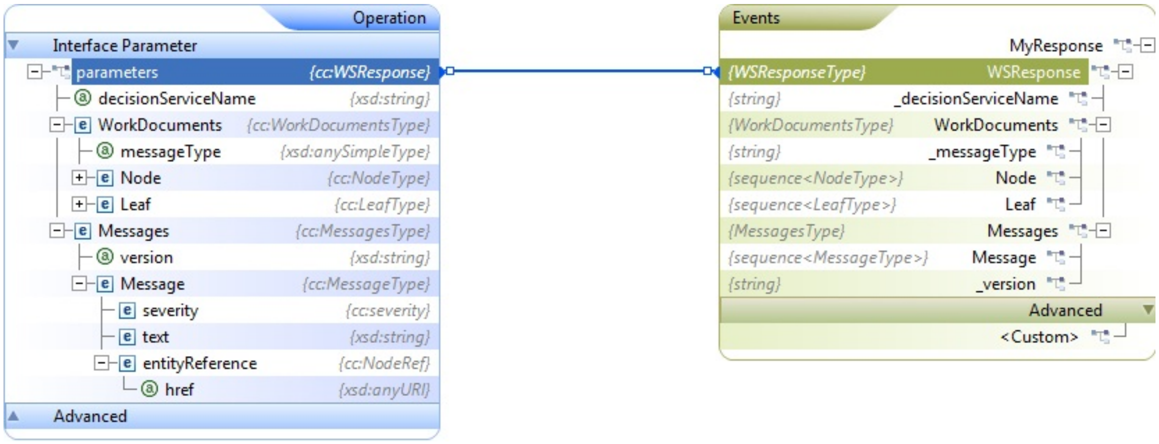
   In the mapping tree, Apama Studio adds an entry of type `Convert To XML` to the selected event node. For example:


Event
MyRequest
WSRequest {WSRequestType}
_decisionServiceName {string}
WorkDocuments {WorkDocumentsType}
_messageType {string}
Node {sequence<NodeType>}
Leaf {sequence<LeafType>}
Convert To XML
Advanced
<Custom>
<Constant>
<System>

6. Drag a line from the `Convert To XML` entry to the `parameters` entry. For example:


Event
MyRequest
WSRequest {WSRequestType}
_decisionServiceName {string}
WorkDocuments {WorkDocumentsType}
_messageType {string}
Node {sequence<NodeType>}
Leaf {sequence<LeafType>}
Convert To XML
Advanced
<Custom>
<Constant>
<System>
Operation
Interface Parameter
{cc:WSRequest} parameters
Advanced
<Custom>
SOAP Headers
SOAP Body
HTTP Headers

For the output mapping, map the output Web Service parameter to an event field. If the event field is of an `event` type that models the output XML (per the convention), the adapter automatically creates the event instance at runtime (implicitly) from the XML. Following is an example of mapping an operation's parameter to an output event:


Operation
Interface Parameter
parameters {cc:WSResponse}
decisionServiceName {xsd:string}
WorkDocuments {cc:WorkDocumentsType}
messageType {xsd:anySimpleType}
Node {cc:NodeType}
Leaf {cc:LeafType}
Messages {cc:MessagesType}
version {xsd:string}
Message {cc:MessageType}
severity {cc:severity}
text {xsd:string}
entityReference {cc:NodeRef}
href {xsd:anyURI}
Advanced
Events
MyResponse
{WSResponseType} WSResponse
{string} _decisionServiceName
{WorkDocumentsType} WorkDocuments
{string} _messageType
{sequence<NodeType>} Node
{sequence<LeafType>} Leaf
{MessagesType} Messages
{sequence<MessageType>} Message
{string} _version
Advanced
<Custom>

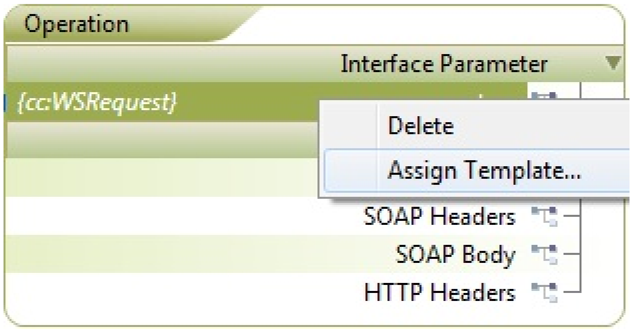
Convention-based XML mapping

Template-based mapping

The template-based approach to mapping lets you map fields in an Apama event to elements and attributes in complex XML structures.

To use the template-based approach:

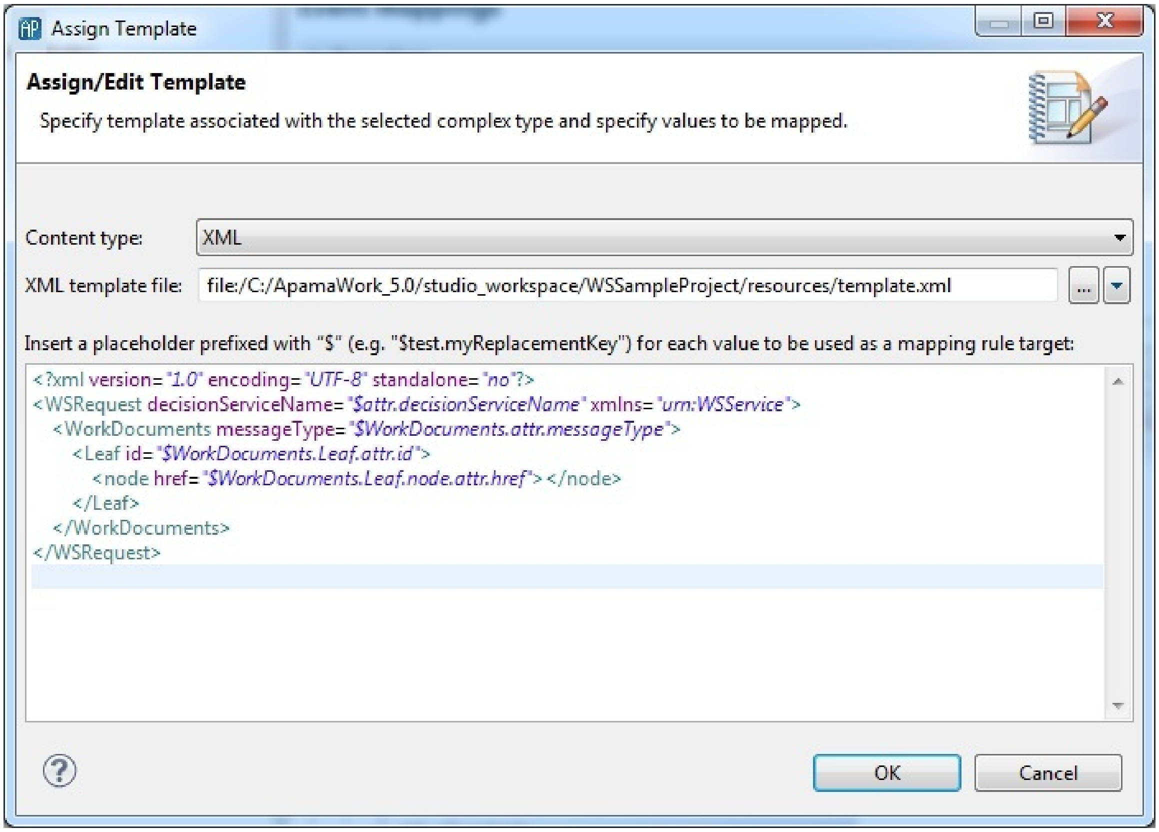
1. In the adapter editor's Event Mappings tab, right-click the operation's parameters entry and select Assign Template.

   The **Assign Template** dialog appears.

2. In the **Assign Template** dialog's XML Template file field, enter the name of the template file you want to use or use the Browse and Down Arrow buttons to locate the file.

   When you specify a template file, the contents of the file are added to the text field in the dialog. For example:



Assign Template

Assign/Edit Template

Specify template associated with the selected complex type and specify values to be mapped.

Content type: XML

XML template file: file:/C:/ApamaWork_5.0/studio_workspace/WSSampleProject/resources/template.xml

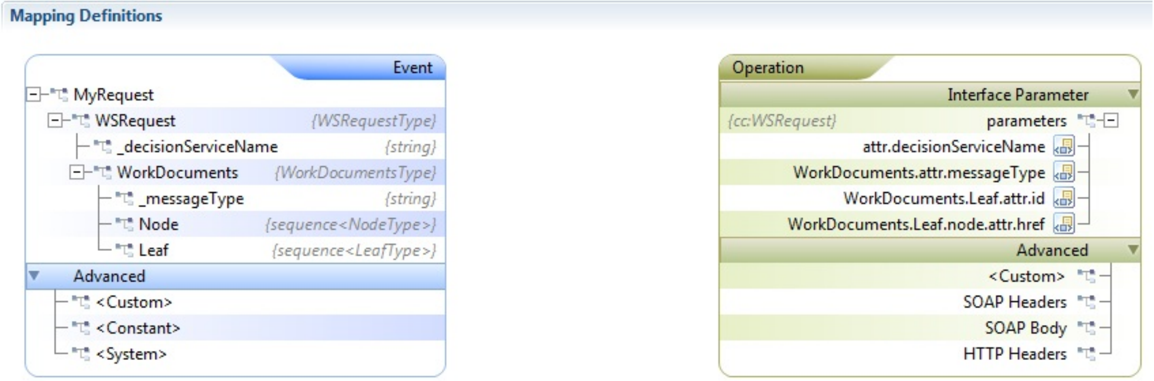
Insert a placeholder prefixed with "$" (e.g. "$test.myReplacementKey") for each value to be used as a mapping rule target:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<WSRequest decisionServiceName="$attr.decisionServiceName" xmlns="urn:WSService">
  <WorkDocuments messageType="$WorkDocuments.attr.messageType">
    <Leaf id="$WorkDocuments.Leaf.attr.id">
      <node href="$WorkDocuments.Leaf.node.attr.href"></node>
    </Leaf>
  </WorkDocuments>
</WSRequest>
```

You must have previously written the template file. In the template file, you define variables to represent field values that you want the adapter to obtain from the input event. To define a variable, insert a dollar sign ($) following by the variable name. After you click OK, the variable name appears as an element in the Operation mapping tree.

3. Edit the template as needed and click OK.

   The Operation hierarchical tree is re-displayed showing the various elements and attributes that are defined in the template.



Mapping Definitions

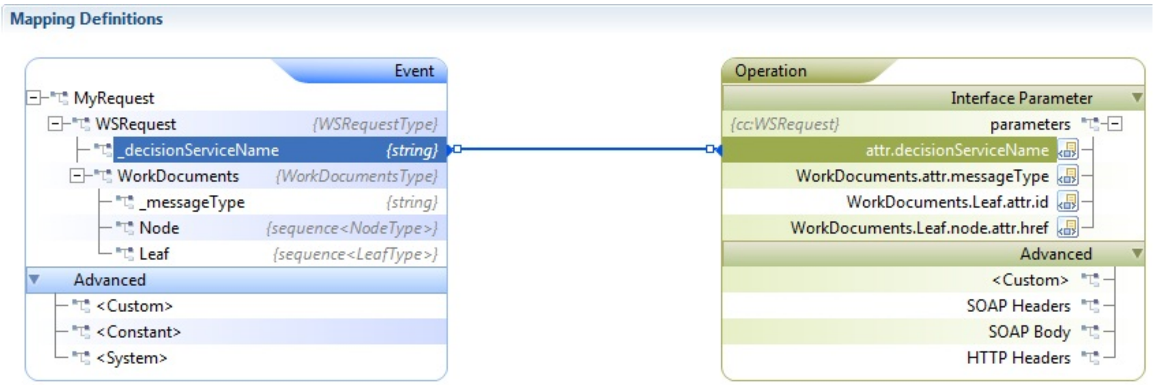
Event

```
MyRequest
  WSRequest                      {WSRequestType}
    _decisionServiceName         {string}
    WorkDocuments                {WorkDocumentsType}
      _messageType               {string}
      Node                       {sequence<NodeType>}
      Leaf                       {sequence<LeafType>}
Advanced
  <Custom>
  <Constant>
  <System>
```

Operation

```
Interface Parameter
{cc:WSRequest}                         parameters
              attr.decisionServiceName
          WorkDocuments.attr.messageType
              WorkDocuments.Leaf.attr.id
        WorkDocuments.Leaf.node.attr.href
Advanced
                                <Custom>
                                SOAP Headers
                                SOAP Body
                                HTTP Headers
```

4. In the Event hierarchical tree, click the Apama event field that you want to map to a particuler element or attribute and drag a line to that element or attribute in the Operation tree.


Mapping Definitions
Event
MyRequest
WSRequest {WSRequestType}
_decisionServiceName {string}
WorkDocuments {WorkDocumentsType}
_messageType {string}
Node {sequence<NodeType>}
Leaf {sequence<LeafType>}
Advanced
<Custom>
<Constant>
<System>
Operation
Interface Parameter
{cc:WSRequest} parameters
attr.decisionServiceName
WorkDocuments.attr.messageType
WorkDocuments.Leaf.attr.id
WorkDocuments.Leaf.node.attr.href
Advanced
<Custom>
SOAP Headers
SOAP Body
HTTP Headers

Mapping Web Service message parameters

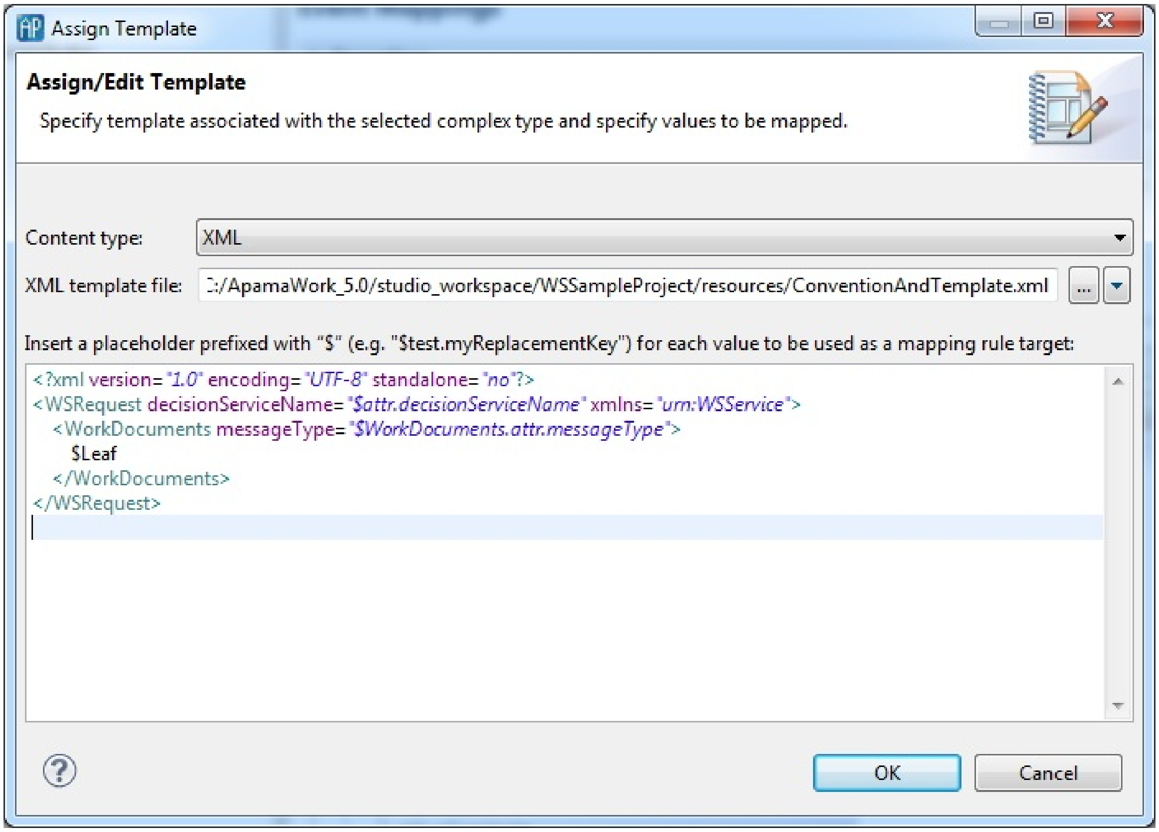
Combining convention and template mapping

You can combine the convention-based and template-based mapping approaches. In this case, you supply an XML template file and you also use the adapter to automatically convert one or more event fields (of `event` or `sequence` type) to XML. The value of one or more converted fields supplies the value for a variable in the template file.

To use the combined mapping approach:

1. In the adapter editor's Event Mappings tab, right-click the operation's parameters entry and select Assign Template. The **Assign Template** dialog appears.
2. In the **Assign Template** dialog's XML Template file field, enter the name of the template file you want to use or use the Browse and Down Arrow buttons to locate the file.

   When you specify a template file, the contents of the file are added to the text field in the dialog.
3. Edit the template file to create at least one variable that will get its value from a field in the input event. For example, suppose you are working with the `WSRequest` and `WSResponse` messages described in "Convention-based Web Service message mapping example" on page 246. You might edit the template to create a variable that will get its value from `Leaf` elements.



Assign Template

Assign/Edit Template

Specify template associated with the selected complex type and specify values to be mapped.

Content type: XML

XML template file: :/ApamaWork_5.0/studio_workspace/WSSampleProject/resources/ConventionAndTemplate.xml

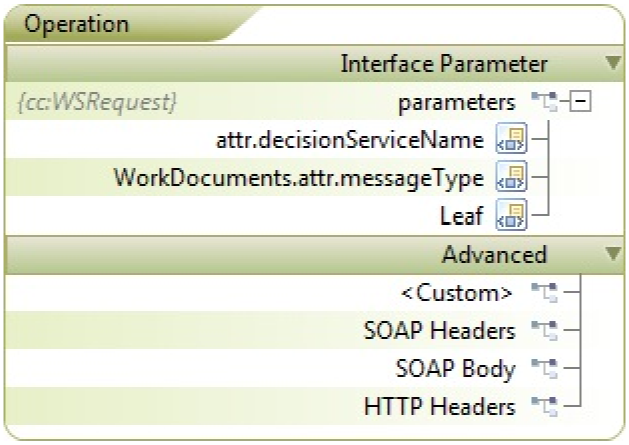
Insert a placeholder prefixed with "$" (e.g. "$test.myReplacementKey") for each value to be used as a mapping rule target:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<WSRequest decisionServiceName="$attr.decisionServiceName" xmlns="urn:WSService">
  <WorkDocuments messageType="$WorkDocuments.attr.messageType">
    $Leaf
  </WorkDocuments>
</WSRequest>
```

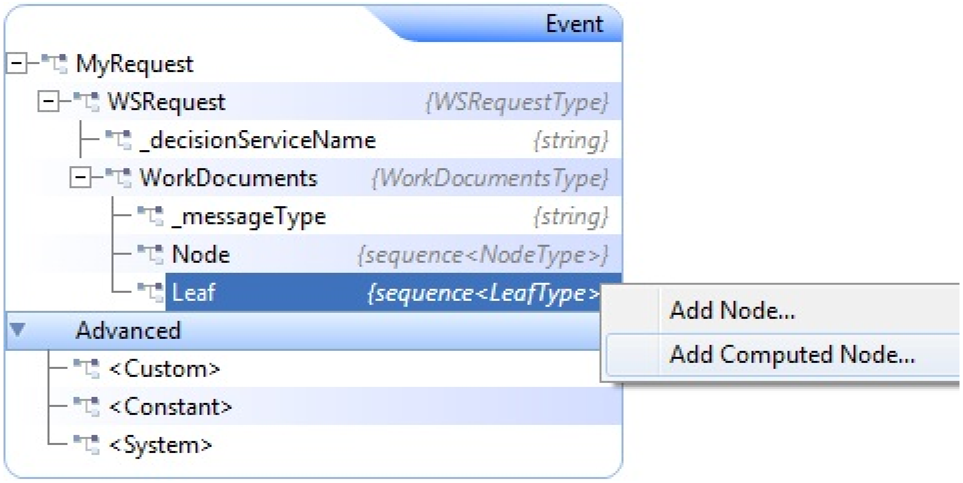
Remember that you must write or have previously written the template file.

4. Edit the template as needed and click OK.

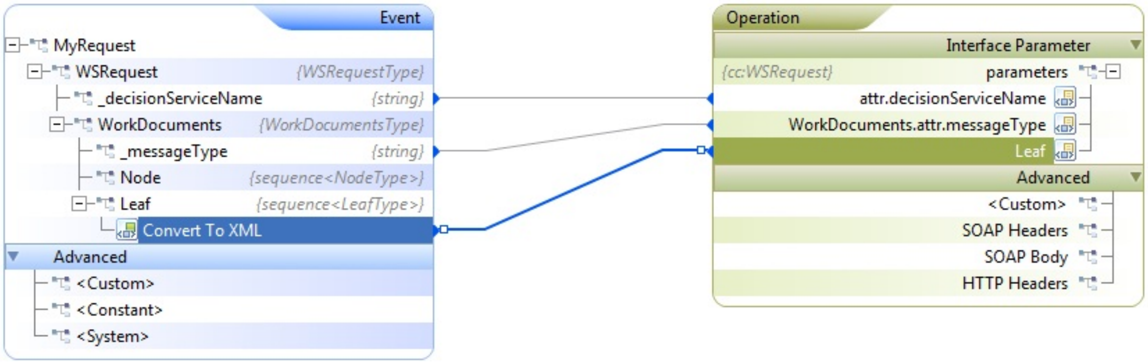
   The Operation hierarchical tree is re-displayed showing the various elements and attributes that are defined in the template, including any variables.



5. In the Event hierarchical tree, right-click the Apama event field that contains a field you want to map to a variable and click Convert Into XML. For example:



6. Right-click an event field in the converted field drag a line from that field to the template variable element in the Operation tree. The event field you select provides the value for the target variable. For example, map the converted `Leaf` sequence to the `Leaf` variable:


Event
MyRequest
WSRequest {WSRequestType}
_decisionServiceName {string}
WorkDocuments {WorkDocumentsType}
_messageType {string}
Node {sequence<NodeType>}
Leaf {sequence<LeafType>}
Convert To XML
Advanced
<Custom>
<Constant>
<System>
Operation
Interface Parameter
{cc:WSRequest} parameters
attr.decisionServiceName
WorkDocuments.attr.messageType
Leaf
Advanced
<Custom>
SOAP Headers
SOAP Body
HTTP Headers

Mapping Web Service message parameters

Mapping complex types

When mapping to a complex Web Service parameter, one option for specifying the input mapping is to provide the entire XML content in an Apama string field, and map that string field to the Web Service parameter. Similarly, in the output mapping, you can map the output Web Service parameter to an Apama string field.

Mapping Web Service message parameters

Difference between doc literal and RPC literal WSDLs

In the RPC literal style of WSDL, the XML document that forms the request and response in the SOAP body includes a parent tag of the operation name. All the message parts are provided under that parent tag. On the other hand, the doc literal style of WSDL uses only one message part in the SOAP body so the XML document contains just a single message part.

In light of the above, it is important to note that when mapping Apama entities to create the XML request, or mapping the response XML back to Apama, the mapping should always be done to/from the message parts, regardless of whether it is a doc literal or RPC literal WSDL. The operation name tag for RPC literal WSDLs will be accounted for automatically and should not be supplied in the request by Apama, and it will not be provided in the response to Apama.

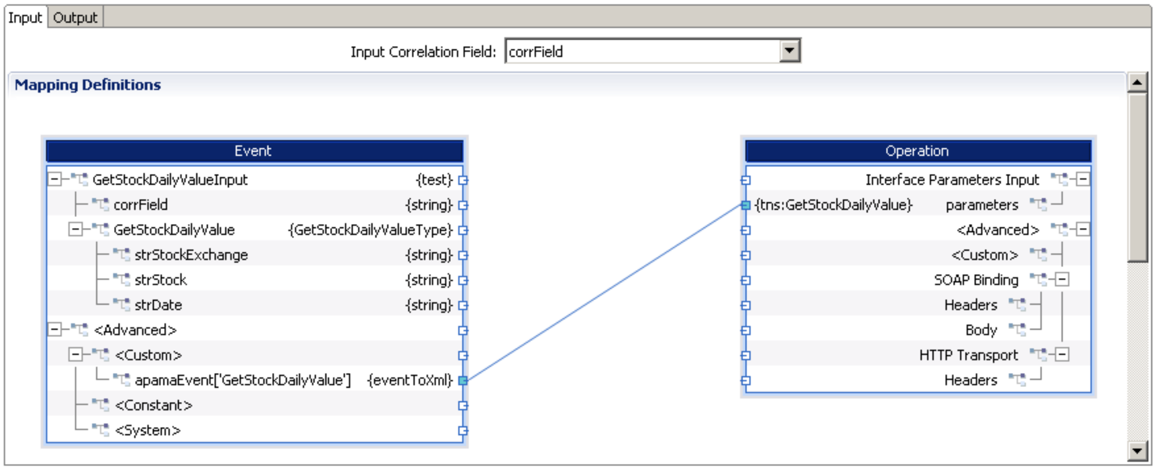
Mapping Web Service message parameters

Specifying a correlation ID field

Because Web Service request (input) and response (output) messages are asychronous, if an Apama application needs to correlate response messages with a specific request message, you need to specify a field in the Apama event that will contain the correlation ID information.

To use Correlation IDs to associate response messages from Web Services with request messages from Apama applications, specify the name of the event field that will contain the correlation ID by selecting the field in the Input Correlation Field, Output Correlation Field, and Fault Correlation Field drop-down lists. Do this for each event (input, output, and fault) that you associate with the operation. These drop-down lists are located on the InputOutput, and Fault tabs, respectively.

In the following image of an Input Mapping Events tab, you can see the Input Correlation Field at the top of the tab, and the corrField field in the input event tree. There is no need to map the correlation ID field. The adapter automatically takes care of this.



Input Correlation Field: corrField

Input
Output
Mapping Definitions
Event
GetStockDailyValueInput {test}
corrField {string}
GetStockDailyValue {GetStockDailyValueType}
strStockExchange {string}
strStock {string}
strDate {string}
<Advanced>
<Custom>
apamaEvent['GetStockDailyValue'] {eventToXml}
<Constant>
<System>
Operation
Interface Parameters Input
{tns:GetStockDailyValue} parameters
<Advanced>
<Custom>
SOAP Binding
Headers
Body
HTTP Transport
Headers

The Apama Web Services Client Adapter

Specifying transformation types

In the Mapping Element Details section, in the Transformation Type field select the desired type from the drop-down list. You can specify either an XSLT stylesheet or an XPath statement.

The Apama Web Services Client Adapter

Specifying an XSLT transformation type

If the mapping from an Apama event type to a Web Service operation's parameters requires an XSLT transformation, specify the transformation details as follows:

1. If necessary, in the adapter editor's Event Mappings tab, in the Mapping Element Details section, draw the line indicating the mapping from source to target.

2. In the Mapping Element Details section, click on the line that specifies the mapping rule you are interested in.
3. In the Transformation Type field select XSLT Transformation from the drop-down list. This displays the Stylesheet URL field.
4. In the Stylesheet URL field, click Browse [...] to locate the file of the stylesheet to use.

Specifying transformation types

Specifying an XPath transformation type

If the mapping from an Apama event type to a Web Service operation's parameters requires an XPath transformation, specify the transformation details as follows:

1. If necessary, in the adapter editor's Event Mappings tab, in the Mapping Element Details section, draw the line indicating the mapping from source to target.
2. In the Mapping Element Details section, click on the line that specifies the mapping rule you are interested in.
3. In the Mapping Element Details section, in the Transformation Type field select XPath statement.
4. In the XPath Expression field, specify a valid XPath expression. You can either enter the XPath expression directly or you can use the **XPath** builder tool to construct an expression.

   To use the **XPath Builder**:

   a. Click the Browse button [...] to the right of the XPath Expression field. The Select input for XPath helper dialog is displayed.

   b. In the Select input for XPath helper dialog, click Browse [...] and select the name of the file that contains a definition of the XML structure (the drop-down arrow allows you to select the scope of the selection process). Click OK. The **XPath Helper** opens, showing the XML structure of the selected file in the left-hand pane.

   c. In the **XPath Helper** build the desired XPath expression by double-clicking on nodes of interest in the left hand pane. The resultant XPath expression is displayed in the XPath tab in the upper right-hand pane.

   d. In the **XPath Helper**, click OK. The **XPath Builder** closes and the XPath Expression field displays the XPath expression you built.

Specifying transformation types

Customizing mapping rules

You can add custom entries to the SOAP Headers, HTTP Headers, and SOAP Body if the Web Service requires it. Create mapping rules for this additional data as described in the following topics.

The Apama Web Services Client Adapter

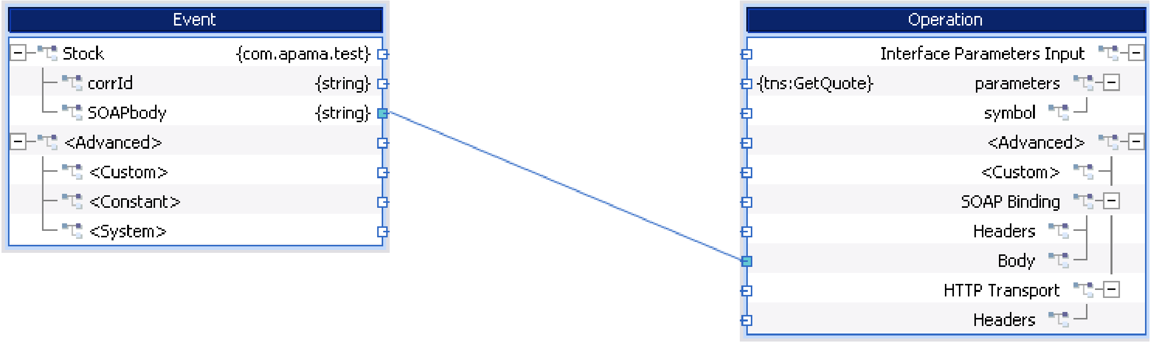
Mapping SOAP body elements

When you invoke an operation you can add custom entries to SOAP body elements if the Web Service requires it. Create mapping rules for SOAP body data as follows.

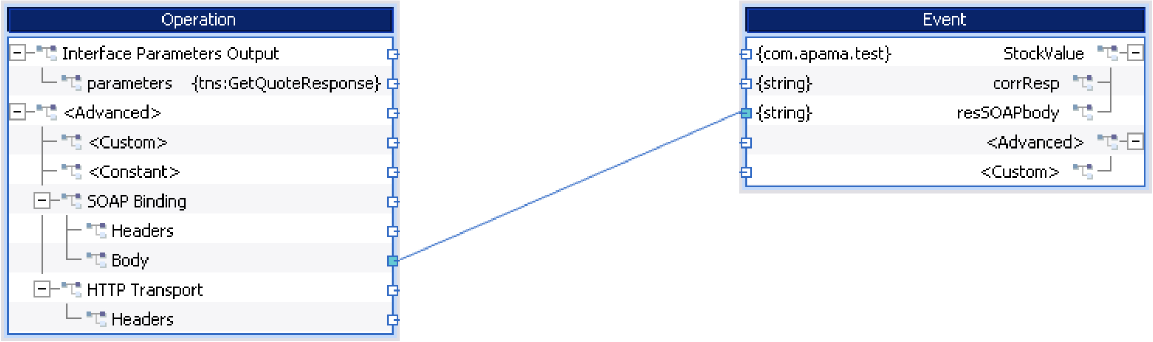
1. Define Apama events that contain `string` fields to hold the SOAP body data. For example:

```
event Stock{
   string corrId;
   string SOAPbody;
}

event StockValue{
   string corrResp;
   string resSOAPbody;
   }
```

2. In the adapter editor's Mapping Definitions tab, map the event field string that contains the SOAP body to the operation's SOAP body parameter. The following image shows an input mapping for a SOAP body:



The following image shows an output mapping for a SOAP body:



In this example, you would need to create the SOAP body in EPL or in some other fashion. To use the Web Services Client adapter to build the XML that makes up the SOAP body, you can use the convention-based approach shown in the next few steps. If you use convention-based mapping for SOAP body data then the event you use must correlate one-to-one to the expected SOAP body.

3. Define Apama events such as the following. In this example, the `_GetStockDailyValue` event contains the SOAP body data.

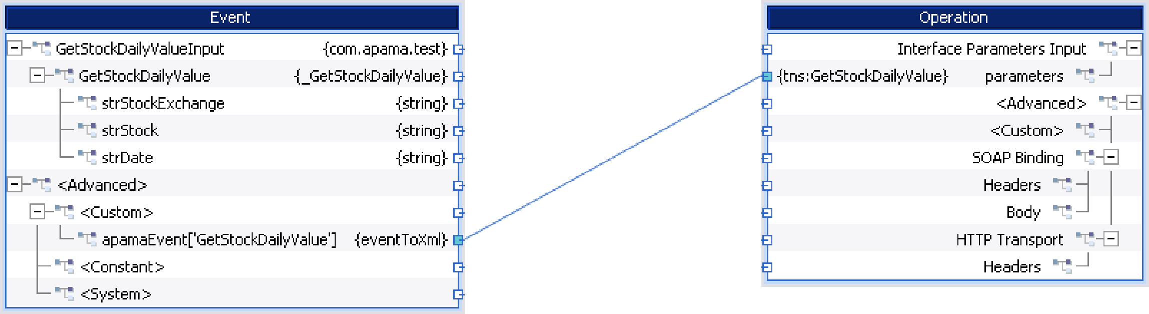
```
event _GetStockDailyValue{
   string strStockExchange;
   string strStock;
   string strDate;
}

event GetStockDailyValueInput{
   _GetStockDailyValue GetStockDailyValue;
}

event _GetStockDailyValueResponse{
   float GetStockDailyValueResult;
}
```

```
event GetStockDailyValueOutput{
   _GetStockDailyValueResponse GetStockDailyValueResponse;
   }
```

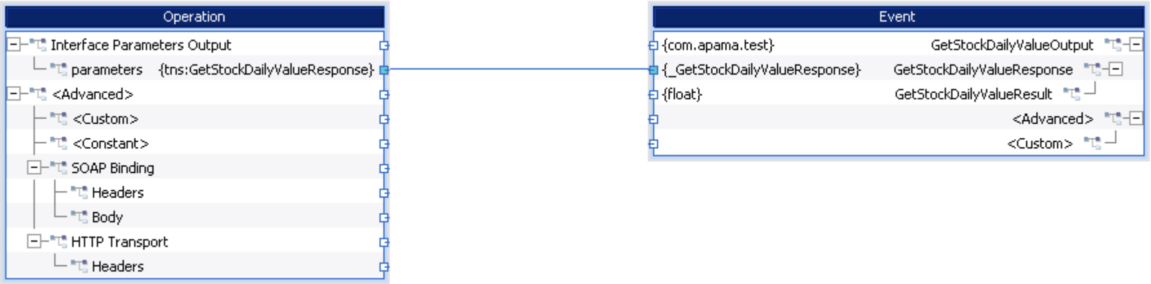
4. In the Input tab of the Event Mappings tab, right-click the event that contains the SOAP body data. In this example, this is `_GetStockDailyValue` field. Select `Add Computed Node` and click `OK`. The **Add Computed Node** dialog appears.

5. In the **Add Computed Node** dialog's Select Method field, select Convert to XML from the drop-down list and click OK.

6. Map the <Custom>apamaEvent that contains the SOAP body to the operation's parameters field. For example, the input mapping might look like this:


Event
GetStockDailyValueInput {com.apama.test}
GetStockDailyValue {_GetStockDailyValue}
strStockExchange {string}
strStock {string}
strDate {string}
<Advanced>
<Custom>
apamaEvent['GetStockDailyValue'] {eventToXml}
<Constant>
<System>
Operation
Interface Parameters Input
{tns:GetStockDailyValue} parameters
<Advanced>
<Custom>
SOAP Binding
Headers
Body
HTTP Transport
Headers

The output mapping might look like this:


Operation
Interface Parameters Output
parameters {tns:GetStockDailyValueResponse}
<Advanced>
<Custom>
<Constant>
SOAP Binding
Headers
Body
HTTP Transport
Headers
Event
{com.apama.test} GetStockDailyValueOutput
{_GetStockDailyValueResponse} GetStockDailyValueResponse
{float} GetStockDailyValueResult
<Advanced>
<Custom>

Customizing mapping rules

Mapping SOAP header elements

When you invoke an operation you can add custom entries to SOAP header elements if the Web Service requires it. Create mapping rules for SOAP header data as follows:

1. Define Apama events that contain `string` fields to hold the SOAP header data. For example:

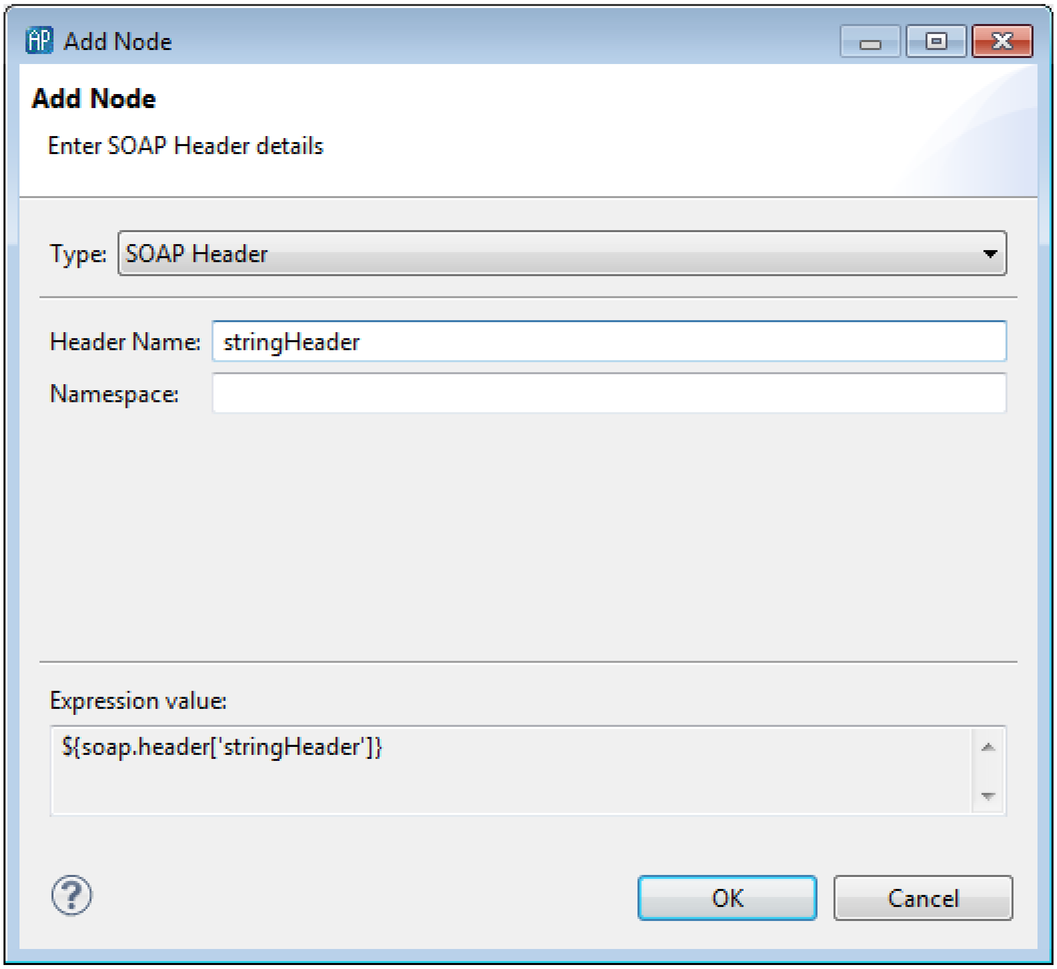
```
event Stock{
   string body;
   string header;
}

event StockValue{
   string resBody;
   string resHeader;
}
```

2. In the adapter editor's Mapping Definitions tab, right-click the operation entry to which you want to add a SOAP header and select Add. The **Add Node** dialog appears.

3. In the **Add Node** dialog's Type field select SOAP Header as the expression type.

4. In the **Add Node** dialog's Header Name field enter the name for the SOAP header and click OK.

   For example:



Add Node

Add Node

Enter SOAP Header details

Type: SOAP Header

Header Name: stringHeader

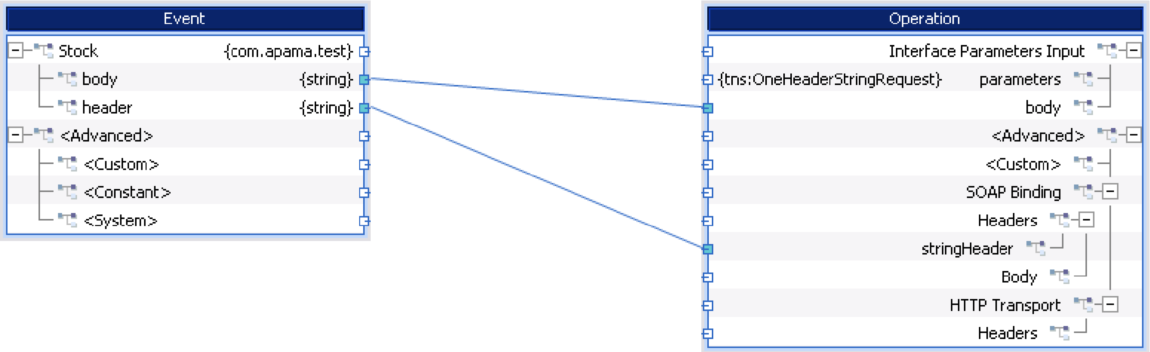
Namespace:

Expression value:

```
${soap.header['stringHeader']}
```

The new entry is added to the Web Service operation's parameters.

5. Map this entry to the event `string` field that will contain the SOAP header. Following is a sample input mapping:



Customizing mapping rules

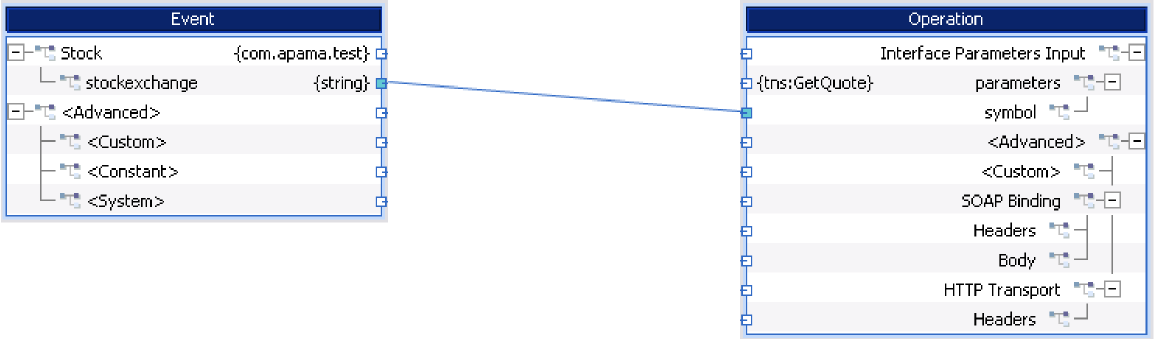
Mapping HTTP header elements

When you invoke an operation you can add custom entries to HTTP header data if the Web Service requires it. The following steps provide an example of obtaining HTTP header data from an operation's response message.

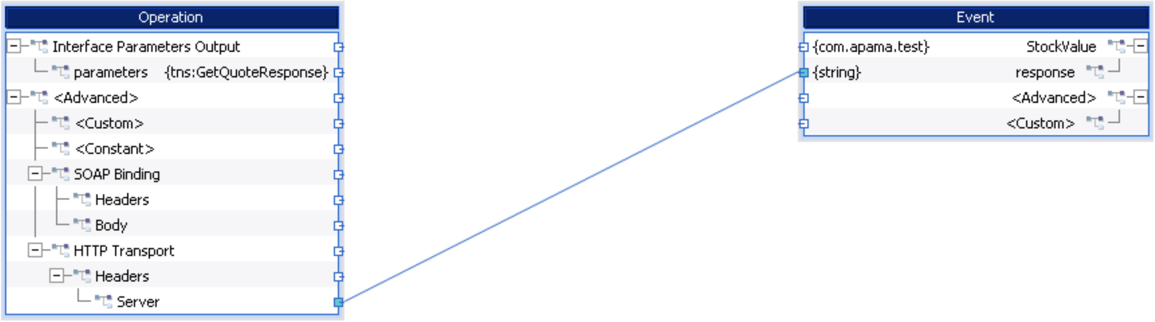
1. Define Apama events that contain `string` fields to hold the HTTP header data. For example:

```
event Stock{
   string stockexchange;
}

event StockValue{
   string response;
}
```

2. In the adapter editor's Input mapping tab for the operation you want to invoke, map event fields to operation parameters as needed to invoke the operation. For example:



3. In the output mapping, retrieve the HTTP header information by mapping it to a `string` field event. For example, to retrieve server details, the output mapping would look like the following:



Customizing mapping rules

Using EPL to interact with Web Services

Apama applications that interact with Web Services need to use specific Apama Event Processing Language (EPL) code to do so.

Define the Apama events that are to be used specifically to interact with the Web Services. If the application will use a convention-based approach to mapping, design Apama events such that the event fields correspond to the elements of the associated Web Service messages. Note that fields that correspond to attributes in the Web Service messages should be prefixed with an underscore character (" _ "). For more information on convention-based mapping, see "Convention-based XML mapping" on page 286.

For example:

```
event GetStockDailyValue{
   string strStockExchange;
   string strStock;
   string strDate;
}

event Stock{
   string stockexchange;
   string stock;
   string date;
}

event StockValue{
   string response;
}
```

The `com.apampa.ws` and `com.apama.statusreport` packages define events that are helpful when invoking Web Services. For convenience, specify `using` statements for the events you want to use so that you do not need to specify their full paths. For example:

```
using com.apama.statusreport.Status;
using com.apama.ws.WSError;
using com.apama.ws.SetupContextListeners;
using com.apama.statusreport.SubscribeStatus;
```

Define a listener for an `AdapterUp` event that indicates that the Web Services Client adapter instance is available to pass your application's service requests to the Web Service. In this listener, be sure so specify the name of the adapter instance as it appears in the EPL file that Apama Studio generates. Also, in the body of the listener, enqueue an event to subscribe to the Web Service. For example:

```
on com.apama.adapters.AdapterUp(adapterName = service_id) {
   enqueue SubscribeStatus(service_id,"","","");
   }
```

Define a listener for an event that indicates that the Web Service is available to respond to requests. When the Web Service is availabe set up a listener for an event that you specified as the output event for an operation, that is, a listener for response messages from the Web Service. If the application depends on a correlation ID, this listener needs to test the value of the field specified as the correlation field. You can then invoke the Web Service by routing an event you specified as the input event for a particular operation. For example:

```
Status stat;
on Status():stat{
   if (stat.available and stat.serviceID = service_id) then {
      listenResponse();
      route Stock("NASDAQ","MSFT","2011-07-12");
      route Stock("NASDAQ","APMA","2011-07-12");
   }
}
action listenResponse() {
   StockValue stockRes;
   on all StockValue():stockRes {
      log stockRes.toString() at INFO;
   }
   WSError wsError;
      on all WSError():wsError {
         print "At service " + wsError.extraParams["serviceId"];
            // wsError always contains service id
         print "For the requested event " + wsError.requestEvent ;
         print " got the error message as " + wsError.errorMsg ;
         print " Failure message is " + wsError.failureType ;
            // See WSConstants event for types of failure
      }
}
```

The following code provides a complete example of how a monitor can invoke a Web Service in a private context. In this example, the monitor sends requests and listens for responses from the Web Service in a private context.

```
using com.apama.statusreport.Status;
using com.apama.ws.WSError;
using com.apama.ws.SetupContextListeners;
using com.apama.ws.TerminateContextListeners;
using com.apama.statusreport.SubscribeStatus;

/*
 * Monitor that will spawn to a private context send requests,
 * and listen for Web Service responses in that private context.
 */
monitor UsingContexts {

   context privateContext;
   constant string service_id := "Webservice_INSTANCE_1";
       // ID that the Apama-Studio-generated EPL uses.

   action onload() {

      on com.apama.adapters.AdapterUp(adapterName = service_id) {
         enqueue SubscribeStatus(service_id,"","","");
            // Subscribe to Web Serivce.
      }

      Status stat;
      on Status():stat{
         if (stat.available and stat.serviceID = service_id) then {
            spawnPrivateContext();
         }
      }
   }

   action spawnPrivateContext() {
      privateContext :=  context("PrivateContext" , false);
         // Create a private context.

      // The application must send a SetupContextListeners event, which
      // contains details such as the service ID (based on generated EPL)
      // and information about the private context in which the monitor
      // listens for service responses.
      route SetupContextListeners(service_id,privateContext);

      spawn sendAndListen() to privateContext;
   }

   action sendAndListen() {
      print "Sending events to " + context.current().toString();
      enqueue Stock("NASDAQ","APMA","2011-07-12") to context.current();
      StockValue stockRes;

      on all StockValue():stockRes {
         log stockRes.toString() at INFO;
         print "Terminating Private context service monitor";
         terminatePrivateContext(); // Terminates service monitor in this context.
      }

      WSError wsError;
      on all WSError():wsError {
         print "At service " + wsError.extraParams["serviceId"];
            // WSError always contains service ID.
         print "For the requested event " + wsError.requestEvent ;
         print " got the error message as " + wsError.errorMsg ;
         print " Failure message is " + wsError.failureType ;
            // See WSConstants event for types of failures.
      }
   }

   action terminatePrivateContext() {
      // The application must send a TerminateContextListeners event,
      // which contains details such as the service ID and the private
      // context that contains the service monitor that contains
      // the service response listeners. When the response listeners
      // terminate, the service monitor that contained them also terminates.
      // The private context is still availabe to be used.
      route TerminateContextListeners(service_id,privateContext);

      //If a request is sent there should not be a response.
      print "Sending request....expecting no response" ;
      enqueue Stock("NASDAQ","APMA","2011-07-12") to privateContext;
   }
}
```

The Apama Web Services Client Adapter

Configuring logging for Web Services Client adapter

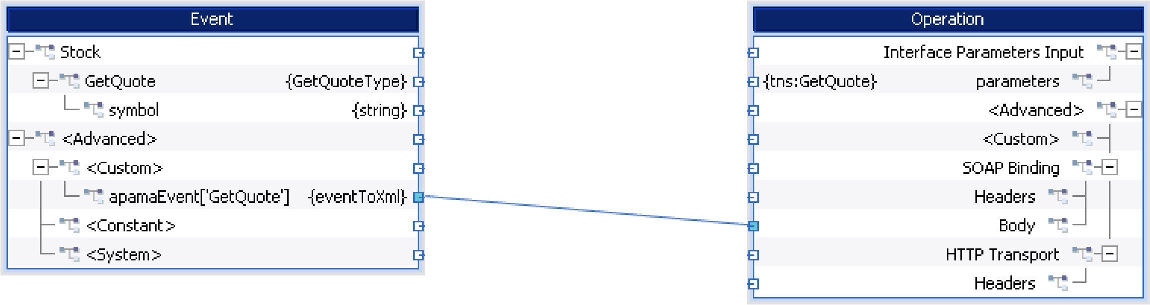
To configure logging for a Web Services Client adapter, modify the `APAMA_HOME/adapters/config/adapters-log4j.properties` file. The default logging level for the properties listed in this file is `INFO`. The `com.progress.el` package contains the mapping framework. Possible logging levels are `INFO`, `WARN`, `ERROR`, `DEBUG`, and `TRACE`.

If you want you can place this file in another location or use a different name for this file. If you do then be sure to update the value of the `log4j.configuration` property in the Web Services Client adapter's IAF configuration file. This is the line you need to change:

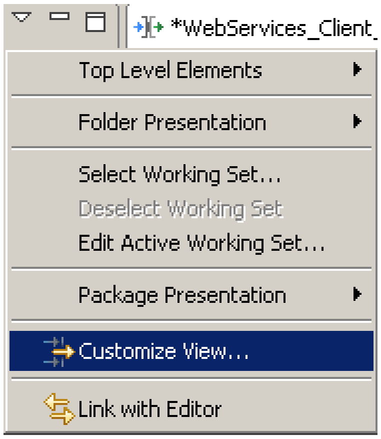
```
<property name="log4j.configuration" value="file:///@APAMA_HOME@/adapters/config/adapters-log4j.properties"/>
```

In addition to general adapter logging, you can configure payload logging. Payload logging can help you diagnose problems by indicating what the constructed request and responses looks like, which can point to whether a problem is in the tooling part of your application or in the runtime execution.

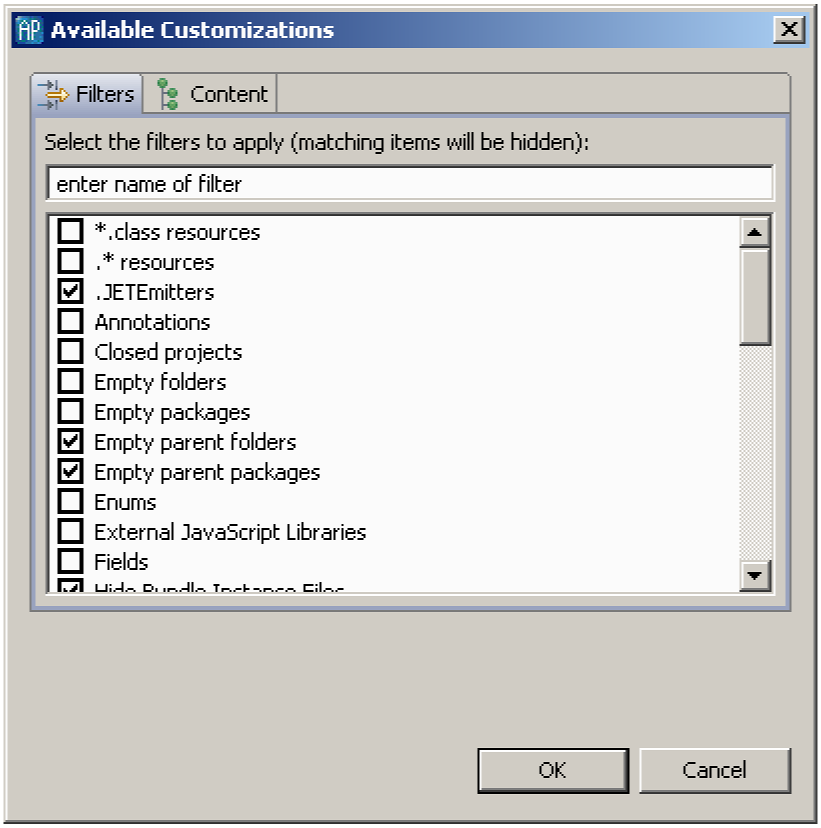
The steps below show how to configure payload logging for a convention-based mapping such as the following:



1. In the Project Explorer toolbar, click View Menu (▽) and select Customize View.



This displays the Available Customizations dialog:



Select the filters to apply (matching items will be hidden):

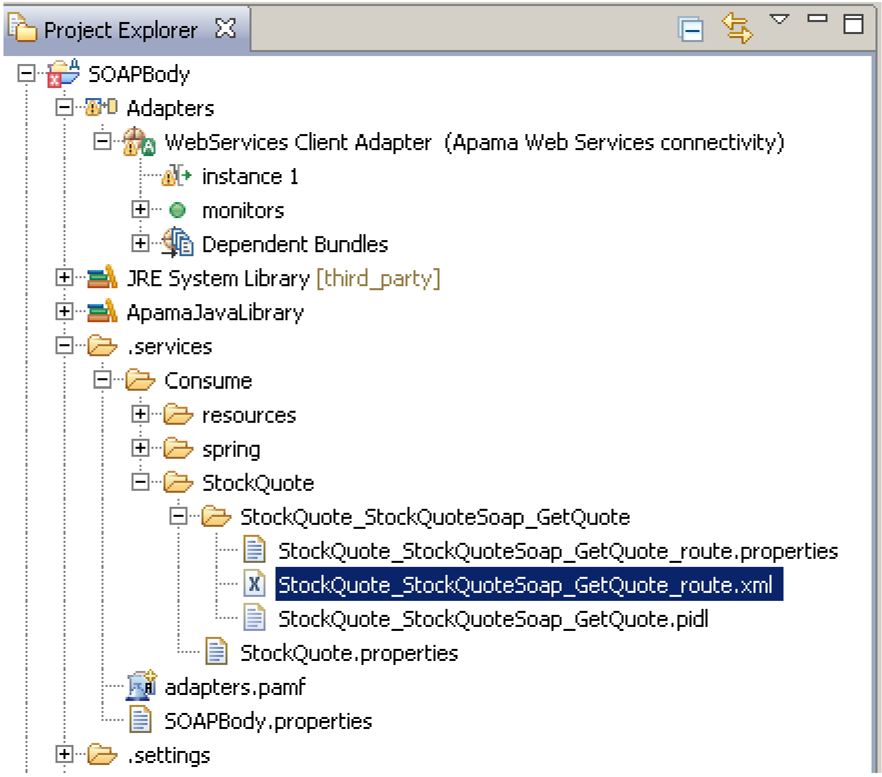
enter name of filter

☐ *.class resources
☐ .* resources
☑ .JETEmitters
☐ Annotations
☐ Closed projects
☐ Empty folders
☐ Empty packages
☑ Empty parent folders
☑ Empty parent packages
☐ Enums
☐ External JavaScript Libraries
☐ Fields
☑ Hide Bundle Instance Files

2. In the Filters tab, ensure that the .*resources checkbox is not selected. Click OK.

   The .services folder now appears in the project's hierarchy in the Project Explorer.

3. For the operation for which you want to enable payload logging, in the project's hierarchy in Project Explorer, open the XML file for the operation's `route` statement. For the mapping shown in the image at the beginning of these instructions, you would select the `StockQuote_StockQuoteSoap_GetQuote_route.xml` file:


Project Explorer
SOAPBody
Adapters
WebServices Client Adapter (Apama Web Services connectivity)
instance 1
monitors
Dependent Bundles
JRE System Library [third_party]
ApamaJavaLibrary
.services
Consume
resources
spring
StockQuote
StockQuote_StockQuoteSoap_GetQuote
StockQuote_StockQuoteSoap_GetQuote_route.properties
StockQuote_StockQuoteSoap_GetQuote_route.xml
StockQuote_StockQuoteSoap_GetQuote.pidl
StockQuote.properties
adapters.pamf
SOAPBody.properties
.settings

4. In the operation's `_route.xml` file, add the following lines just after the `<cxf.properties>` section:

   ```
   <cxf:features>
      <bean class="org.apache.cxf.feature.LoggingFeature"/>
   </cxf:features>
   ```

   For example:

   ```
   <cxf:cxfEndpoint
   . . .>
      <cxf:properties>
         <entry key="mtom-enabled" value="${CXF_mtom_enabled_StockQuote_StockQuoteSoap_GetQuote_route}"/>
         <entry key="dataFormat" value="${CXF_dataFormat_StockQuote_StockQuoteSoap_GetQuote_route}"/>
      </cxf:properties>
      <cxf:features>
         <bean class="org.apache.cxf.feature.LoggingFeature"/>
      </cxf:features>
   </cxf:cxfEndpoint>
   ```

5. In the `APAMA_HOME/adapters/config/adapters-log4j.properties` file, change the default `INFO` setting for `log4j.logger.org.apache.cxf` to the logging level you want.

After you invoke the operation you specified payload logging for, the `iaf.log` file for the Web Services Client adapter displays something like the following:

```
--------------------------------
ID: 1
Address: http://www.webservicex.net/stockquote.asmx
Encoding: UTF-8
Content-Type: text/xml
Headers: {SOAPAction=["http://www.webserviceX.NET/GetQuote"], Accept=[*/*]}
Payload: <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body><GetQuote xmlns="http://www.webserviceX.NET/">
   <symbol>APMA</symbol></GetQuote></soap:Body></soap:Envelope>
--------------------------------
```

The Apama Web Services Client Adapter

Upstream multithreading

The Web Services client adapter supports using multiple threads for sending input messages. In some situations an application may experience throughput performance gains when using multiple threads. Situations that may benefit from the use of multiple threads include:

- Template-based input mapping where there are a large number of substitution variables in the template.
- The number of input mapping rules is high.

Input mapping is scalable in only a few selected types of mapping so, by default the adapter does not use an upstream thread pool and upstream processing takes place in the default IAF thread. If you want your application to use upstream multithreading, you need to specify a value for the `minUpstreamThreads` property.

The Apama Web Services Client Adapter

Specifying upstream multithreading transport properties

By default the Web Services Client adapter does not use an upstream thread pool (or upstream queue) and upstream processing takes place in the default thread. If you want to use upstream multithreading, the `minUpstreamThreads` property needs to be set. You can also set the `maxUpstreamThreads` and `upstreamQueueSize` properties.

- `minUpstreamThreads` - There is no default for this property, which means that the adapter does not use an upstream thread pool. When this property is specified, an upstream thread pool and queue is used. Note that these threads are only used to perform the input mapping part of the WebService workflow.
- `maxUpstreamThreads` - This property is only applicable if the `minUpstreamThreads` property is set. If not specified, the default value for this property is twice the value of the `minUpstreamThreads` property.
- `upstreamQueueSize` - This property is only applicable if the `minUpstreamThreads` property is set. If not specified, the default value for this property is 20000.

You add these property settings to the adapter instance's XML code as follows:

1. In the Project Explorer, expand the project’s `Adapters` node and then expand the `WebServices Client Adapter` node.
2. Double-click the entry for the adapter instance you want to configure. This opens the adapter instance configuration in the Web Service adapter editor.
3. In the Web Service adapter editor, select the XML Source tab.
4. In the adapter's `<transport>` element add a `<property>` sub-element, specifying the `name` attribute of the property you want and its `value` attribute.

Upstream multithreading

Web Services Client adapter artifacts

Apama Studio automatically generates the various artifacts that enable the application to connect to the Web Service and invoke its operations. The following types of files are generated:

- Web Services files - These include files that define, for example, the interface, beans, and routes used by the Web Services itself.
- Apama files - These are files that define Apama service monitors and mapping rules. These files listen for Apama events that are then mapped to Web Services request messages. The mapping rules also determine how Web Services response messages are associated with fields in Apama events.

The Apama Web Services Client Adapter

12 Correlator-Integrated JMS Messaging

Apama support for Java Message Service (JMS) messaging is integrated into the Apama correlator. This provides an efficient method for Apama applications to use JMS messages to communicate with external systems.

Correlator-integrated JMS messaging overview

The Java Message Service (JMS) is a Java API that provides a common programming model for asynchronously sending events and data across enterprise messaging systems. JMS supports two models, *publish-and-subscribe* for one-to-many message delivery and *point-to-point* for one-to-one message delivery. Apama's correlator-integrated JMS messaging implements both these models.

When configured to use correlator-integrated JMS messaging, Apama applications map downstream (incoming) JMS messages to Apama events and map upstream (outgoing) Apama events to JMS messages.

Apama's correlator-integrated JMS messaging implements three levels of reliability, leveraging the more basic reliability mechanisms provided by JMS itself, and making them simpler to use. When the reliability level is set to `EXACTLY_ONCE` or `AT_LEAST_ONCE`, delivery is guaranteed because messages are logged by the broker in persistent storage until they are received and acknowledged. When the reliability level is set to `BEST_EFFORT`, message delivery is not guaranteed. For applications that do not require guaranteed message delivery, the `BEST_EFFORT` mode provides greater performance.

You can specify the JMS configuration in Apama Studio, either in the JMS Correlator-Integrated adapter editor or by editing sections of the XML and `.properties` configuration files directly. Note, however that the mapping configuration should always be edited using the Apama Studio adapter editor in this release.

Correlator-Integrated JMS Messaging

Correlator-Integrated JMS example applications

Apama Studio provides the following example applications that illustrate the use of Correlator-integrated JMS messaging. The examples are located in the *`apama_dir`*`\samples\correlator_jms` directory.

- `simple-send-receive` - This application demonstrates simple sending and receiving. It sends a sample event to a JMS queue or topic as a JMS TextMessage using the automatically configured default sender and receives the message using a statically-configured receiver.
- `dynamic-event-api` - This application demonstrates how to use the event API to dynamically add and remove JMS senders and receivers. In addition, it shows how to monitor senders and receivers for errors and availability.
- `flow-control` - This application demonstrates how to use the event API to avoid sending events faster than JMS can cope with and a separate demonstration of how to avoid receiving messages from JMS faster than the EPL application can cope with.

Correlator-integrated JMS messaging overview

Key concepts

The key JMS concepts when implementing an Apama application with correlator-integrated JMS messaging are *connections*, *receivers*, and *senders*.

JMS connections

To use JMS you must configure one or more named connections to the JMS broker. If you need to connect to multiple separate JMS broker instances (which may be using the same JMS provider/ vendor or different ones) you need a connection for each; it's also possible to add multiple connections for the same broker (for example, for rare cases where it improves performance scalability). In Apama Studio you can select from a variety of JMS providers that come with default connection configurations.

JMS receivers

A receiver is a single-threaded context for receiving messages from a single JMS queue or topic (with a single JMS `Session` and `MessageConsumer` object). A connection to a JMS broker can be configured with any number of receivers. Many, but not all, JMS providers support creating multiple receivers for a single queue (or in some cases, topic) either to scale throughput performance, or when using JMS 'message selectors' to partition the messages on a destination.

JMS senders

A sender is a single-threaded context for sending messages (with a single JMS `Session` and `MessageProducer` object). A connection to a JMS broker can be configured with any number of senders. You can add any number of senders, but by default if no senders are explicitly configured, a single sender called "default" will be created implicitly. Each sender can send messages to any JMS destination (a queue or topic); the destination is specified on a per-message basis in the mapping rule set (either hardcoded by specifying a constant value per message type in the mapping rules or mapped from a destination field in the apama event). Messages sent by a single sender with the same JMS headers ("priority" for example) will usually be delivered in order by the provider (although this may not be the case if there is a failure), but the ordering of sends across senders is undefined. Multiple senders can be created for a single connection to scale throughput performance, or for sending messages with different `senderReliability` modes. Each sender is represented by its own correlator output channel.

Correlator-integrated JMS messaging overview

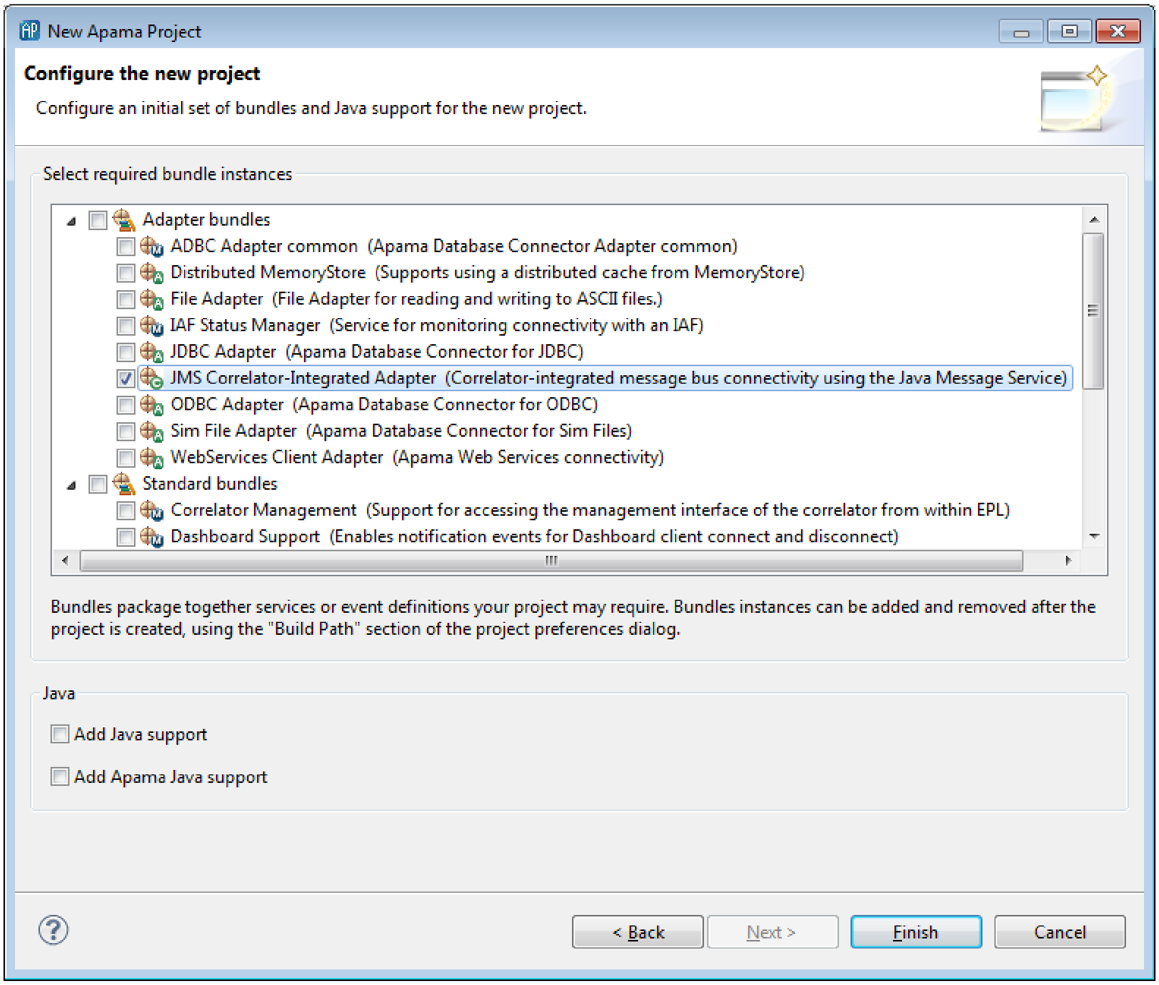
Getting started - creating an application with simple JMS messaging

This section describes the steps for creating an Apama application that uses correlator-integrated JMS messaging where guaranteed delivery is not required. Apama Studio provides an example application that illustrates a simple use of JMS messaging in the *`apama_dir`*`\samples\correlator_jms\simple-send-receive` directory.

To make correlator-integrated JMS messaging available to an Apama project:

1. Select File > New > Apama Project from the Apama Studio menu. This launches the **New Apama Project** wizard.

2. In the **New Apama Project** wizard, give the project a name, and click Next. The second page of the wizard is displayed, listing the available Apama resource bundles.



New Apama Project

Configure the new project

Configure an initial set of bundles and Java support for the new project.

Select required bundle instances

- ☐ Adapter bundles
  - ☐ ADBC Adapter common (Apama Database Connector Adapter common)
  - ☐ Distributed MemoryStore (Supports using a distributed cache from MemoryStore)
  - ☐ File Adapter (File Adapter for reading and writing to ASCII files.)
  - ☐ IAF Status Manager (Service for monitoring connectivity with an IAF)
  - ☐ JDBC Adapter (Apama Database Connector for JDBC)
  - ☑ JMS Correlator-Integrated Adapter (Correlator-integrated message bus connectivity using the Java Message Service)
  - ☐ ODBC Adapter (Apama Database Connector for ODBC)
  - ☐ Sim File Adapter (Apama Database Connector for Sim Files)
  - ☐ WebServices Client Adapter (Apama Web Services connectivity)
- ☐ Standard bundles
  - ☐ Correlator Management (Support for accessing the management interface of the correlator from within EPL)
  - ☐ Dashboard Support (Enables notification events for Dashboard client connect and disconnect)

Bundles package together services or event definitions your project may require. Bundles instances can be added and removed after the project is created, using the "Build Path" section of the project preferences dialog.

Java

☐ Add Java support

☐ Add Apama Java support

3. Apama's correlator-integrated JMS messaging makes use of the Apama JMS Correlator-Integrated adapter. In the **New Apama Project** wizard, from the Select required bundle instances field, select the JMS Correlator-Integrated Adapter bundle.

4. Click Finish.

Apama Studio adds the JMS Correlator-Integrated adapter to the project's `Adapters` node. In addition, Apama Studio generates all the necessary resources to support correlator-integrated JMS messaging. Note, you can only add a single instance of the JMS Correlator-Integrated adapter to an Apama project.

After you add the JMS Correlator-Integrated adapter, you need to configure connections to a JMS broker and configure senders and receivers.

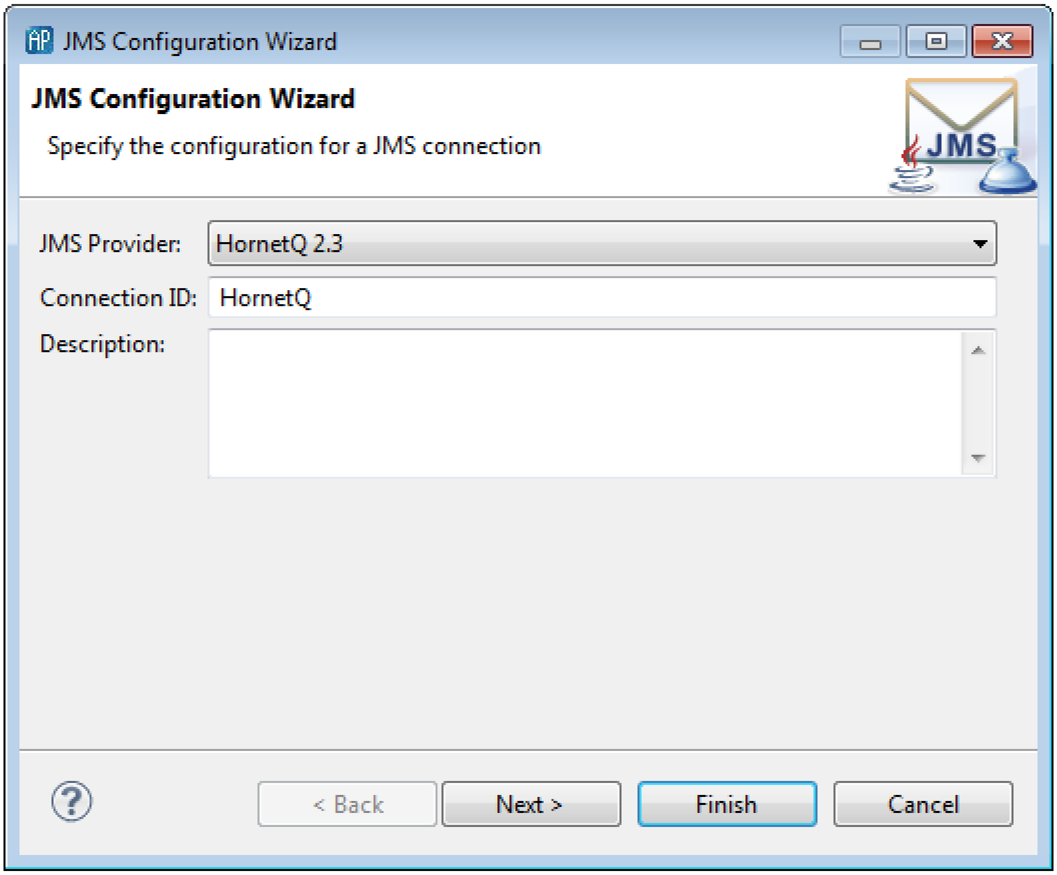
Correlator-Integrated JMS Messaging

Configuring connections

When you first add the correlator-integrated JMS messaging bundle to an Apama Studio project, the list of connections is initially empty. You can add one or more connection to JMS providers.

To establish a connection to a JMS broker:

1. In the Project Explorer expand the project's `Adapters` node and then expand the `Correlator-Integrated JMS` node.
2. Double-click the adapter instance. This opens the instance's configuration in the JMS Correlator-Integrated adapter editor.
3. In the adapter editor's Settings tab, click the Add Connection button (+). This displays the JMS Configuration wizard.



JMS Configuration Wizard

Specify the configuration for a JMS connection

JMS Provider: HornetQ 2.3

Connection ID: HornetQ

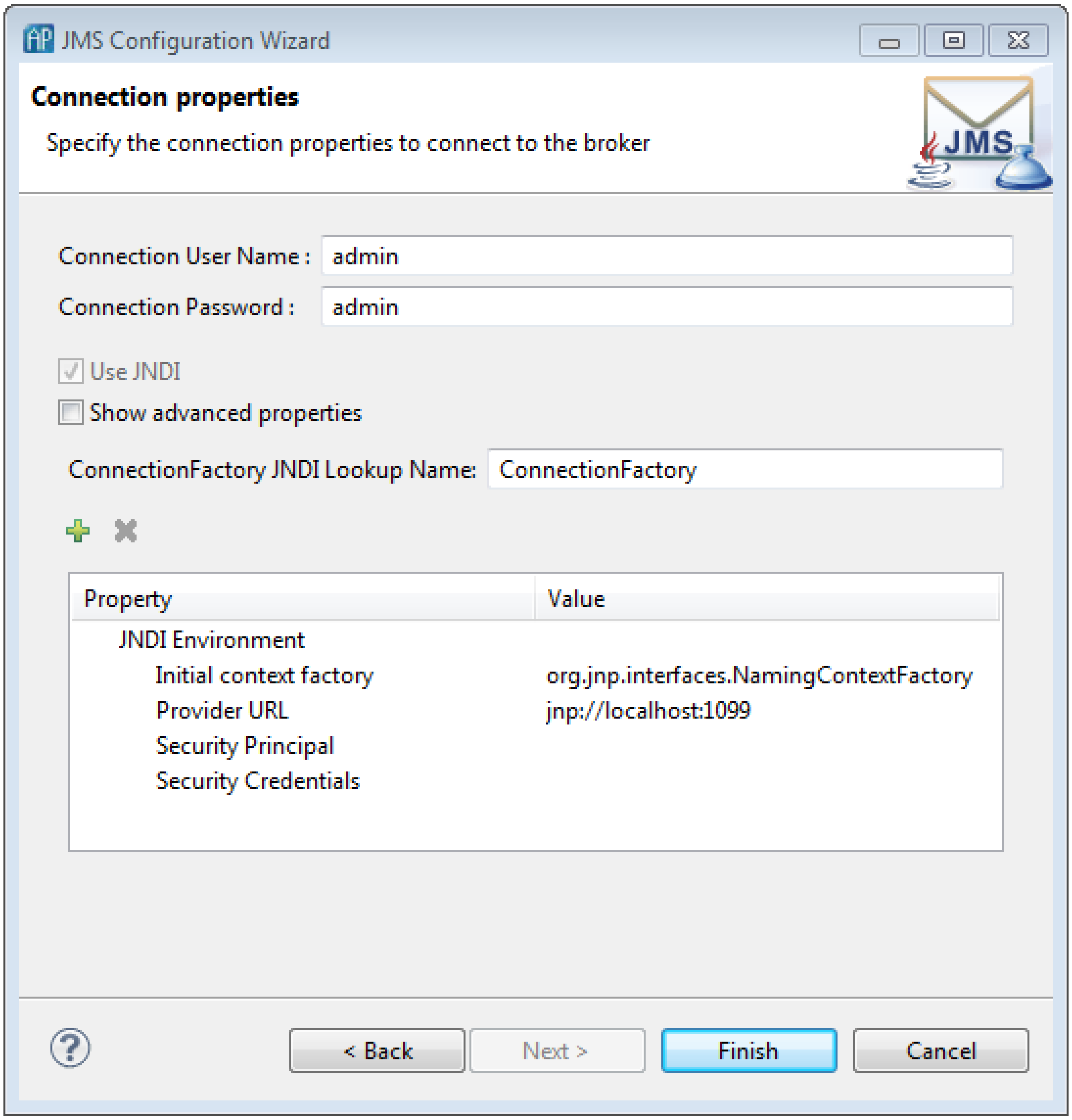
Description:

4. On the first page of the JMS Configuration wizard specify the following.
   a. JMS Provider, select from the drop-down list.
   b. Connection ID, this needs to be unique, and will be used throughout the configuration files and Apama application to identify this broker connection (but the Apama connection identifier is not exposed to the JMS provider in any way). The Connection ID is used when sending JMS messages from the Apama application. The value for the Connection ID should not contain any spaces.
   c. Description, this is optional and currently unused.
5. Click Next.

   This displays the second page of the **JMS Configuration** wizard.
6. The second page of the **JMS Configuration** wizard displays the default Classpath details for the JMS Provider you selected in the previous step. If necessary, add or modify the values as appropriate for your environment.

7. Click Next.

   This displays the third page of the **JMS Configuration** wizard.



JMS Configuration Wizard

Connection properties

Specify the connection properties to connect to the broker

Connection User Name : admin

Connection Password : admin

☑ Use JNDI

□ Show advanced properties

ConnectionFactory JNDI Lookup Name: ConnectionFactory

Property	Value
JNDI Environment	
Initial context factory	org.jnp.interfaces.NamingContextFactory
Provider URL	jnp://localhost:1099
Security Principal	
Security Credentials	

8. The third page of the **JMS Configuration** wizard displays the default Connection properties for the JMS Provider you selected. If necessary, add or modify the values as appropriate.

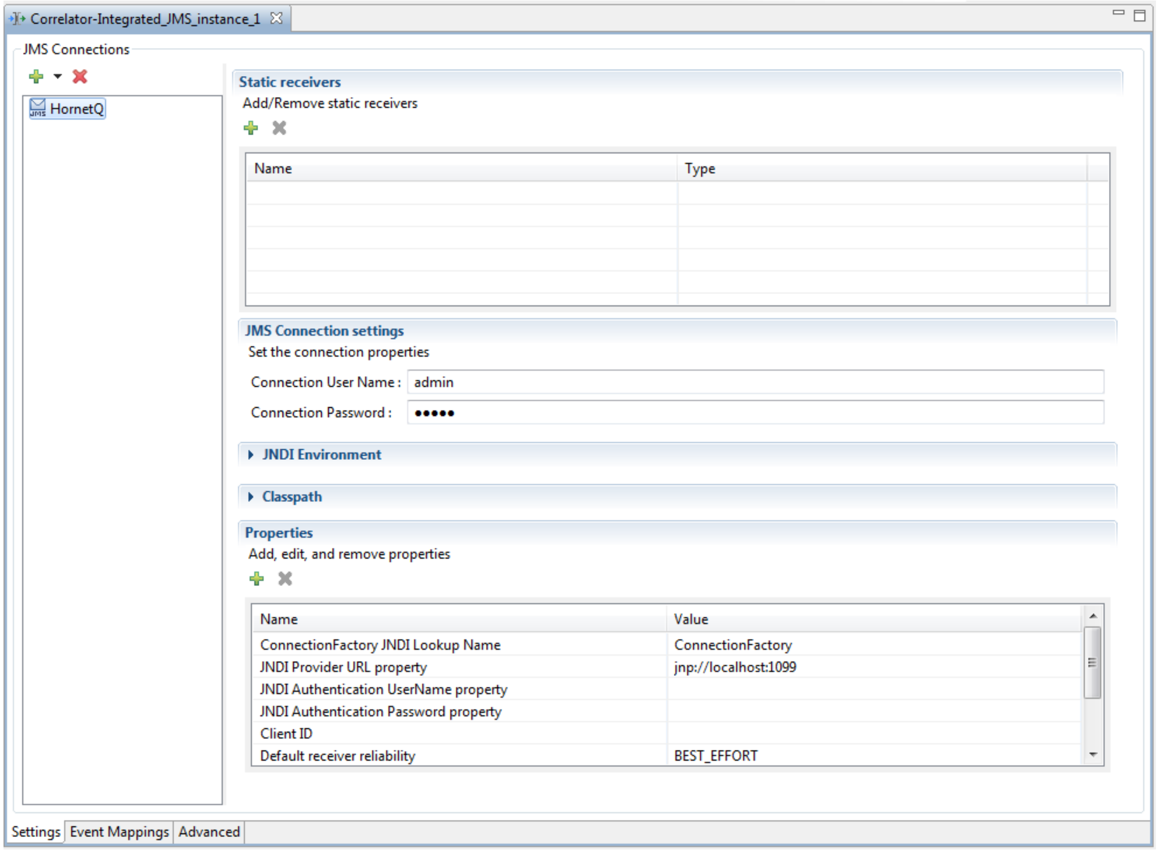
   a. By default, the JMS Connection uses JNDI (which is the recommended option); remove the check in the Use JNDI check box if your application will not use JNDI and your JMS provider supports non-JNDI instantiation of the `ConnectionFactory` class (note that many providers do not).

   b. By default, the **JMS Configuration** wizard lists a subset of standard connection properties. If Use JNDI is enabled, the Connection details field shows `JNDI Environment` properties. If Use JNDI is not enabled, the Connection details field shows `ConnectionFactory` properties. To show the complete list of properties add a check to the Show advanced properties check box.

   c. You can add and remove properties and you can modify the properties' values. To modify a value, click in the Value column and enter the required information.

9. Click Finish.

Apama Studio updates the adapter editor to display the new connection in the JMS Connections section.



Name	Type

Connection User Name : admin

Connection Password : •••••

Name	Value
ConnectionFactory JNDI Lookup Name	ConnectionFactory
JNDI Provider URL property	jnp://localhost:1099
JNDI Authentication UserName property	
JNDI Authentication Password property	
Client ID	
Default receiver reliability	BEST_EFFORT

After you establish a connection to a JMS broker, you need to add JMS receivers and specify mapping configurations for receivers and senders.

Getting started - creating an application with simple JMS messaging

Adding JMS receivers

JMS receivers are added to JMS Connections. To add a JMS receiver to a project:

1. In the Project Explorer, double-click the project's JMS Correlator-Integrated Adapter instance. This opens the instance configuration in the Apama Studio adapter editor.
2. Select the desired JMS Connection.
3. In the Static Receivers section click the Add destination button (✚).

   This adds a receiver with a default name to the Name column and a default type (queue) to the Type column.
4. If desired, you can edit the value in the Name column. You can edit the value in the Type column by clicking the value and selecting a new type from the drop-down list at the right.

After you have configured the JMS receivers for each queue or topic of interest, you need to configure how the received JMS messages will be mapped to Apama events.

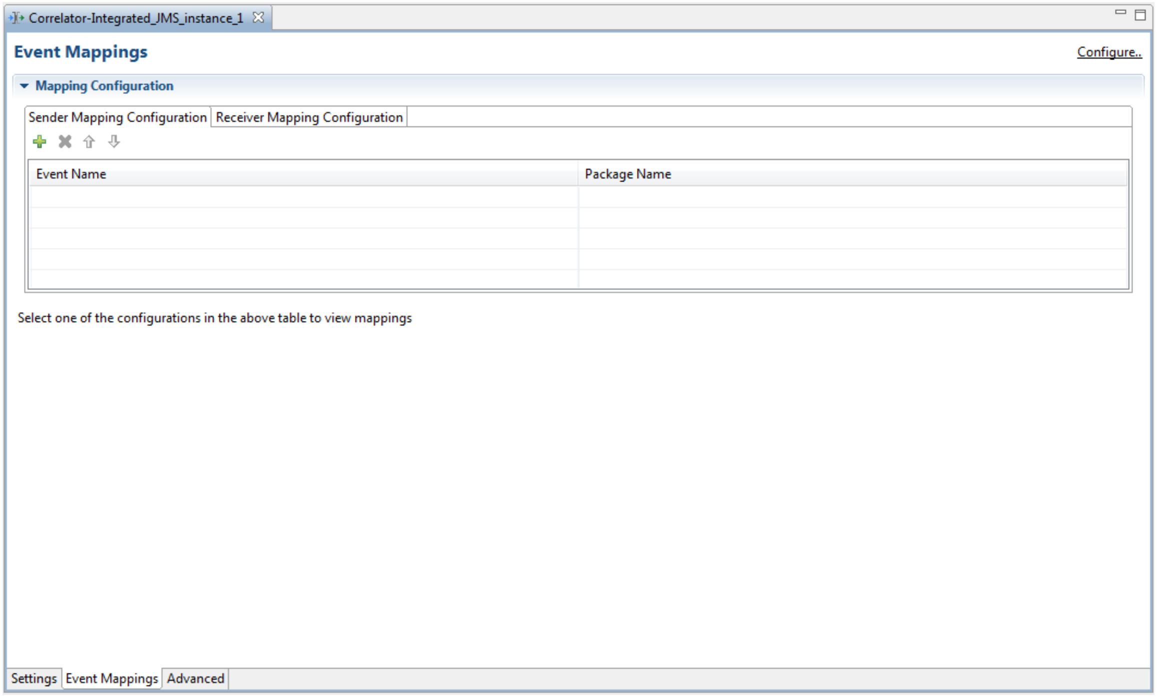
Getting started - creating an application with simple JMS messaging

Configuring Receiver Event Mappings

Each event mapping for a received JMS message is configured by specifying the target Apama event type, a conditional expression to determine which source JMS messages should be mapped to this event type, and a set of mapping rules that populate the fields of the target Apama event based on the contents of the source JMS message.

To configure an event mapping:

1. Ensure that the Apama event types you wish to use for mapping have been defined in an EPL file in your project.
2. In the Project Explorer, double-click the project's JMS Correlator-Integrated Adapter instance. This opens the instance configuration in the Apama Studio JMS Correlator-Integrated adapter editor.
3. In the adapter editor, select the Event Mappings tab.



Select one of the configurations in the above table to view mappings

4. On the adapter editor's Event Mappings tab in the Mapping Configuration section, select the Receiver Mapping Configuration tab.
5. Click the Add Event button (✚).

   This displays the **Event Type Selection** dialog.
6. In the **Event Type Selection** dialog's Event Type Selection field, enter the name of the event. As you type, event types that match what you enter are shown in the Matching Items list.
7. In the Matching Items list, select the name of the event type you want to associate with the JMS message. The name of the EPL file that defines the selected event is displayed in the status area at the bottom of the dialog.
8. Click OK.

This updates the display in the adapter editor's Event Mappings tab to show a hierarchical view of the JMS message on the left (the mapping source) and a hierarchical view of the Apama event on the right (the mapping target). In addition, the Expression column displays a default JUEL conditional expression that determines which JMS messages will use the specified mapping rules. If you need to use a different conditional expression, you can edit the default. For more information see "Using conditional expressions" on page 277.

9. Map the JMS message to the Apama event by clicking on the entity in the Message tree and dragging a line to the entity in the Event tree. For example, the simplest mapping for a standard JMS `TextMessage` would be a single mapping rule from the JMS message Body to a single string field in the Apama event. More complex mapping involves mapping the value of one or more JMS headers or properties or parsing XML content out of the text message. For more information see "Mapping Apama events and JMS messages" on page 283.

If a receiver mapping configuration lists multiple events, the Correlator-JMS mapper evaluates the expressions from top to bottom, stopping on the first mapping whose conditional expression evaluates to true. You can use the up and down arrows to change the order in which the evaluations are performed.

Getting started - creating an application with simple JMS messaging

Using conditional expressions

When you configure event mappings for received JMS messages, you specify Apama event types to which JMS messages will be mapped along with the mapping rules. The correlator-integrated JMS mapper uses JUEL expressions to indicate which mapping rules to use. JUEL (Java Unified Expression Language) expressions are a standard way to access data. When you specify an event type for a receiver, Apama Studio creates a default conditional expression that evaluates a JMS property named `MESSAGE_TYPE`, testing to see if its value is the name of the specified Apama event type. You can modify the default expression if you need to test for a different condition, depending on the format of the JMS messages that Apama will be receiving.

Depending on your application's needs, you can create a conditional expression for the following cases:

- Match a JMS Header
- Match a JMS Property
- If the XML document root element exists
- Match an XPath into the JMS message body
- When you want to use a custom expression

1. On the Receiver Mapping Configuration tab, click the expression in the Expression column.
2. Click the Browse button next to the expression. This displays the **Conditional Expression** dialog, where you can edit the default expression.
3. In the Condition field, select the type of conditional expression you want from the drop-down list. Depending on your selection, the remaining available fields will vary.
4. Fill in the remaining fields as required. For some fields you select from drop-down lists, for others you enter values directly. If you select the Custom type of conditional expression, you can edit the expression directly, but note that if a string literal in the expression contains a single or double quotation mark, it needs to be escaped with the backslash character (`\'` or `\"`).

5. Click OK. The new expression is displayed in the Express column of the Receiver Mapping Configuration tab.

Conditional operators in custom expressions

The following operators are available:

- `==` equal to
- `!=` not equal
- `lt` less than
- `gt` greater than
- `le` less than or equal
- `ge` greater than or equal
- `and`
- `or`
- `empty` null or empty
- `not`

Custom conditional expression examples

In most cases the decision about which Apama event type to map to for a given JMS message is based on a JMS message property value or sometimes a header, such as JMSType. In other cases, when there is no alternative, the decision is made by parsing XML content in the document body and evaluating an XPath expression over it. Here are some examples of typical conditional expressions.

- JUEL boolean expression based on a JMS string property value:

  ```
  ${jms.property['MY_MESSAGE_TYPE'] == 'MyMessage1'}
  ```

- JUEL boolean expression based on a JMS header value:

  ```
  ${jms.header['JMSType'] == 'MyMessage1'}
  ```

- JUEL boolean expression based on the existence of the XML root element 'message1' in the body of a TextMessage:

  ```
  ${xpath(jms.body.textmessage, 'boolean(/message1)')}
  ```

- JUEL boolean expression based on testing the value of an XML attribute in the body of a TextMessage:

  ```
  ${xpath(jms.body.textmessage, '/message/info/@messageType') == 'MyMessage'}
  ```

- JUEL boolean expression for matching based on message type (`TypeMessage`, `MapMessage`, `BytesMessage`, `ObjectMessage`, or `Message`):

  ```
  ${jms.body.type == 'TextMessage'}
  ```

The following boolean JUEL expressions show advanced cases demonstrating what is possible using JUEL and illustrating how the syntax works with example XML documents

- JUEL expression that matches all messages:

  ```
  ${true}
  ```

- 'greater than' numeric operator:

  ```
  ${jms.property['MY_LONG_PROPERTY'] gt 120}
  ```

- Using backslash to escape quotes inside a JUEL expression:

  ```
  ${jms.body.textmessage == 'Contains \'quoted\' string'}
  ```

- Operators 'not', 'and', 'or', and 'empty':

  ```
  ${not (jms.property['MY_MESSAGE_TYPE'] == 'MyMessage1' or
                      jms.property['MY_MESSAGE_TYPE'] == 'MyMessage2') and
                      not empty jms.property['MY_MESSAGE_TYPE']}
  ```

- Testing the value of an entry in the body of a MapMessage:

  ```
  ${jms.body.mapmessage['myMessageTypeKey'] == 'MapMessage1'}
  ```

- An advanced XPath query (and use of JUEL double-quoted string literal and XPath single-quoted string literal in the same expression)

  ```
  ${xpath(jms.body.textmessage, " (count(/message3/e) > 2) and
           /message3/e[2] = 'there' and
           (/message3/e[1] = /message3/e[3]) ")}
  ```

  For an XML document such as

  ```
  <message3><e>Hello</e><e>there</e><e>Hello</e></message3>
  ```

- XPath namespace support:

  ```
  ${xpath(jms.body.textmessage, " /message4/*[local-name()='element1' and
           namespace-uri()='http://www.myco.com/testns']/text() ") ==
           'Hello world'}
  ```

  For an XML document such as

  ```
  <message4 xmlns:myprefix="http://www.myco.com/testns">
         <element1>No namespace</element1>
         <myprefix:element1>Hello world</myprefix:element1></message4>
  ```

- Recursively parsing XML content nested in the CDATA section of another XML document:

  ```
  ${xpath( xpath(jms.body.textmessage, '/messageA/text()'),
           '/messageB/text()') == 'MyNestedMessageType'}
  ```

  For an XML document such as

  ```
  <messageA><![CDATA[
     <messageB>MyNestedMessageType</messageB> ]]>
  </messageA>
  ```

For a table of expressions for getting and setting values in JMS messages and recommended mappings to Apama event types, see "JUEL Mapping Expressions Reference" on page 298.

Configuring Receiver Event Mappings

Configuring Sender Event Mappings

Each event mapping for a JMS message to be sent is configured by specifying the source Apama event type, and a set of mapping rules that populate the target JMS message from the fields of the source Apama event.

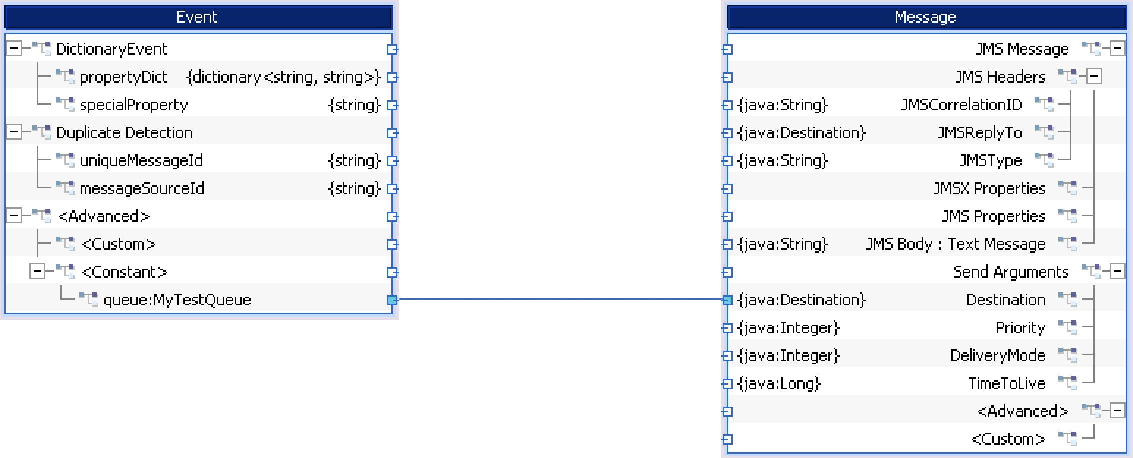
To configure an event mapping:

1. Ensure that the Apama event types you wish to use for mapping have been defined in an EPL file in your project.
2. If necessary, in the Project Explorer, double-click the project’s JMS Correlator-Integrated Adapter instance. This opens the instance configuration in the Apama Studio adapter editor.

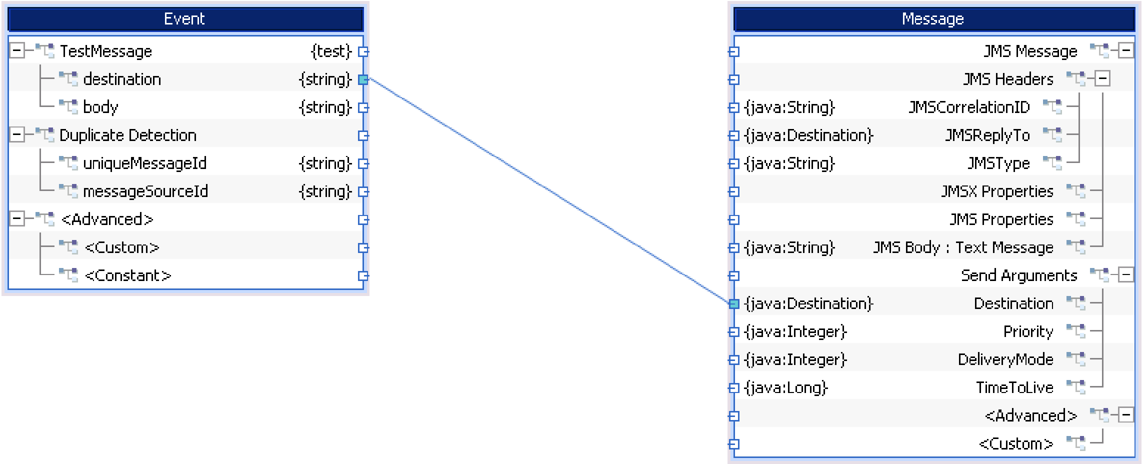
3. Select the JMS Connection.
4. In the JMS Correlator-Integrated Adapter editor, select the Event Mappings tab.
5. On the adapter editor's Event Mappings tab, select the Sender Mapping Configuration tab.
6. On the Sender Mapping Configuration tab click the Add Event button (✚). This displays the **Select Event** dialog.
7. In the **Event Type Selection** dialog‘s Event Type Selection field, enter the name of the event. As you type, event types that match what you enter are shown in the Matching Items list.
8. In the Matching Items list, select the name of the event type you want to associate with the JMS message. The name of the EPL file that defines the selected event is displayed in the status area at the bottom of the dialog.
9. Click OK.

   This updates the display in the adapter editor's Event Mappings tab to show a hierarchical view of the Apama event on the left (the mapping source) and a hierarchical view of the JMS message on the right (the mapping target).
10. Create a mapping rule as follows:
    a. If necessary, click on the event to be mapped in the Event name column.
    b. Click on the entity in the Event tree and drag a line to the entity in the Message tree.

    For example, a simple mapping would be from a single `string` field in an Apama event to the JMS message Body. More complex mappings might involve mapping an event field to a specific JMS property.For more information see "Mapping Apama events and JMS messages" on page 283.
11. Specify the message's JMS destination in either of two ways:
    - Specify a constant value in the event type's mapping:



      For more information on specifying a constant value, see "Using expressions in mapping rules" on page 284.
    - Specify a destination in an event field and map that field to the message:



Note *destination* is always specified as topic:*name*, queue:*name*, or jndi:*name*

Getting started - creating an application with simple JMS messaging

Using EPL to send and receive JMS messages

The EPL code necessary for using correlator-integrated JMS message is minimal.

- Initialization - Your application needs to notify the correlator that the application has been injected and is ready to process events from the JMS broker.

  1. Apama recommends that after *all* an application's EPL has been injected, the application should send an application-defined 'start' event using a .evt file. Using an event is clearer and more reliable than enabling JMS message receiving using monitor `onload()` actions because it is easier to guarantee that all EPL has definitely been injected and is in a good state before the event is sent and JMS message receiving commences.

  2. Any monitors that need to use the Correlator-integrated JMS event API will have a variable (typically a monitor-global field) holding a JMS event object.

  3. The monitor that handles the application-defined start event (from step 1), should use this JMS event object to notify Correlator-integrated JMS that the application is initialized and ready to receive messages, for example:

     ```
     on com.mycompany.myapp.Start() {
                     jms.onApplicationInitialized();
                     // any other post-injection startup logic goes here too
     }
     ```

- Receiving events - Once you have configured a JMS receiver, to receive JMS messages, you simply need to add EPL listeners for the events specified in the mapping configuration.

- Sending events - To send JMS messages emit the Apama event associated with the JMS message in the Sender Mapping Configuration using the following syntax.

  ```
  emit event_name to "jms:senderId";
  ```

  Note, *senderId* is typically "*connectionId*-default-sender" unless explicitly configured with a different name. For example to emit an event to the default sender on a connection called "MyConnection", use the following:

  ```
  emit MyEvent to "jms:MyConnection-default-sender";
  ```

For more information on specifying the message's JMS destination, see "Configuring Sender Event Mappings" on page 279.

Getting started - creating an application with simple JMS messaging

Getting started - creating an application with reliable JMS messaging

This section describes the steps for creating an Apama application that uses reliable correlator-integrated JMS messaging in an environment where guaranteed delivery is required. In order to provide reliable JMS messaging, you set specific JMS Connection properties. In addition, reliable JMS messaging makes use of Apama's correlator persistence feature, which specifies that the correlator periodically writes its state to stable storage.

The steps described in this section build on the example created in "Getting started - creating an application with simple JMS messaging" on page 271

To enable reliable correlator-integrated JMS messaging for an Apama project:

1. If necessary, create an Apama project that uses correlator-integrated JMS messaging as described in "Getting started - creating an application with simple JMS messaging" on page 271.
2. If necessary, in the Project Explorer expand the project's `Adapters` node, expand the `JMS Correlator-Integrated Adapter` node, and double-click the adapter instance. This opens the instance's configuration in the JMS Correlator-Integrated Adapter editor.
3. In the adapter editor, display the Settings tab and in the JMS Connection section, select the JMS connection to use.
4. Click the Properties section to expand it.
5. In the Properties section, select `EXACTLY_ONCE` or `AT_LEAST_ONCE` for the Default receiver reliability and Default sender reliability properties. Both `EXACTLY_ONCE` and `AT_LEAST_ONCE` reliability modes prevent message loss, but while `AT_LEAST_ONCE` is simpler and offers greater performance, `EXACTLY_ONCE` adds detection and elimination of duplicate messages (if configured correctly) which may be required for some applications.
6. If receiving with `EXACTLY_ONCE` reliability, it is necessary to configure additional mapping rules to specify an application-level unique identifier for each received message that will function as the key for detecting functionally duplicate messages. To add these mapping rules, display the Event Mappings tab and in the source event tree, map the `uniqueMessageId` and (optionally, but recommended) `messageSourceId` entities to appropriate values in the JMS message. For example, they could be mapped to JMS message properties called `UNIQUE_MESSAGE_ID` and `MESSAGE_SOURCE_ID` (or to nodes within an XML document in the message body). When sending JMS messages, the mapping rules provide a way to expose the `uniqueMessageId` and `messageSourceId` that Apama automatically generates for emitted messages to whatever downstream JMS client will be receiving them, so that it can perform duplicate detection.
7. In your application's EPL code, add the `persistent` keyword before the monitor declarations for monitors listening for Apama events associated with JMS messages.
8. In the project's Run Configuration, enable correlator persistence as follows.
   a. In the **Run Configuration** dialog, select the Components tab.
   b. Select Default correlator and click Edit. The **Correlator Configuration** dialog is displayed.
   c. In the **Correlator Configuration** dialog, select the Persistence Options tab, add a check mark (tick mark) to the Enable correlator persistence setting, and click OK.

Running a correlator in this way causes the it to periodically write its state to stable storage.

For more information on correlator persistence, see "Using Correlator Persistence" in *Developing Apama Applications in EPL*.

Correlator-Integrated JMS Messaging

Mapping Apama events and JMS messages

After you specify which Apama events you want to associate with JMS messages, you need to create mapping rules that associate Apama event fields with parts of the JMS messages. The Apama Studio adapter editor provides a visual mapping tool to create the mapping rules. There are several approaches for how to map Apama events to the JMS messages.

- "Simple mapping for JMS messages" on page 283 - Use this approach when a simple Apama event field can be associated with a corresponding value in the JMS message.
- "Using expressions in mapping rules" on page 284 - Use this when sending or receiving JMS messages and you need to write a customized JUEL expression for a mapping rule.
- "Template-based XML generation" on page 284 - Use this when sending JMS messages containing XML; you assign a template to generate an XML document, containing placeholder values substituted in from fields in the source event.
- "Specifying an XPath transformation for JMS messages" on page 285 - Use this when receiving JMS messages containing XML to specify values from the XML document that are to be used to populate the fields in the target Apama event.
- "Specifying an XSLT transformation for JMS messages" on page 285 - Use this when receiving JMS messages containing XML, to change or simplify the structure of the XML document.
- "Convention-based XML mapping" on page 286 - Use this to parse or generate XML documents by using event definitions that follow specific conventions to implicitly encode the structure of the XML document. This approach allows mapping of sequences to elements of the same type. It avoids the need for XPath, but does impose some limitations on the XML naming and structure.

Correlator-Integrated JMS Messaging

Simple mapping for JMS messages

When creating a simple 1:1 mapping rule for an Apama event field to part of a JMS message that contains a similar type, you can drag a line between the elements as follows:

1. In the JMS Correlator-Integrated adapter editor, display the Event Mapping tab.
2. For each mapping rule, click on the entity you want to map and drag a line to the entity you want to map it to.

Apama Studio represents each rule with a blue line between entities. If the types of the source and target do not match, type coersion will be performed automatically at runtime.

Mapping Apama events and JMS messages

Using expressions in mapping rules

In many cases, a mapping rule requires customization. For example, if you map an event field to a JMS Property, you need to specify which JMS property to use. In other cases you may want to use a constant value in a mapping rule or to create a JUEL expression, for example to execute an XPath query on nested XML documents.

1. Drag a mapping line from the entry in the source tree to the target. If one side of the mapping rule requires a more specific expression, the **Connection Participants** dialog is displayed.
2. In the **Connection Participants** dialog's Type field, select an entry from the drop-down list.
3. In the next field enter the name of the JMS Body type, the JMS Property name, a constant value, or a custom JUEL expression. As you enter this information, the expression that wil be used in the mapping rule is displayed in the Expression Value field.
4. Click OK.

For a table of expressions for getting and setting values in JMS messages and recommended mappings to Apama event types, see "JUEL Mapping Expressions Reference" on page 298.

Mapping Apama events and JMS messages

Template-based XML generation

With the template-based approach to mapping, you can map fields in an Apama event to elements and attributes in complex XML structures. The template consists of a sample XML document with placeholders that will be replaced with values from the Apama event fields. When you assign a template, these variables are displayed in the JMS message tree; you then map event fields to the variables.

To assign a template for mapping:

1. In the adapter editor's Event Mappings tab, right-click the JMS Body entry and select Assign Template. The **Assign Template** dialog is displayed.
2. In the XML Template file field, enter the name of the template file you want to use or use the Browse and Down Arrow buttons to locate the file.

   When you specify a template file, the contents of the file are added to the text field in the dialog.

   It is usually best to create the template file from a sample XML document before opening this dialog, but it is also possible to perform this task from the dialog itself, for small XML documents. To create the XML template, you define placeholders to represent field values that you want the adapter to obtain from the input event. To define a placeholder, insert a dollar sign (`$`) following by the placeholder name. After you click OK, the placeholder appears as a new child of the target's JMS body node.
3. In the source event, click the Apama event field and drag a line to the desired element or attribute in the target JMS message.

Mapping Apama events and JMS messages

Specifying an XPath transformation for JMS messages

If a mapping rule from an Apama event to JMS message requires an XPath transformation, specify the transformation details as follows:

1. If necessary, in the adapter editor's Event Mappings tab, in the Mapping Element Details section, draw the line indicating the mapping from source to target.
2. In the Mapping Element Details section, click on the line that specifies the mapping rule you are interested in.
3. In the Mapping Element Details section, in the Transformation Type field select XPath.
4. In the XPath Expression field, specify a valid XPath expression. You can either enter the XPath expression directly or you can use the **XPath** builder tool to construct an expression.

   To use the **XPath Builder**:

   a. Click the Browse button [...] to the right of the XPath Expression field. The Select input for XPath helper dialog is displayed.

   b. In the Select input for XPath helper dialog, click Browse [...] and select the name of the file that contains a definition of the XML structure (the drop-down arrow allows you to select the scope of the selection process). Click OK. The **XPath Helper** opens, showing the XML structure of the selected file in the left-hand pane.

   c. In the **XPath Helper** build the desired XPath expression by double-clicking on nodes of interest in the left hand pane. The resultant XPath expression is displayed in the XPath tab in the upper right-hand pane. If the XML document makes use of namespaces, change the namespace option from `Prefix` to `Namespace` or `Local name`.

   d. In the **XPath Helper**, click OK. The **XPath Builder** closes and the XPath Expression field displays the XPath expression you built.

Mapping Apama events and JMS messages

Specifying an XSLT transformation for JMS messages

If a mapping rule from an Apama event to a JMS message requires an XSLT transformation, specify the transformation details as follows:

1. If necessary, in the adapter editor's Event Mappings tab, in the Mapping Element Details section, draw the line indicating the mapping from source to target.
2. In the Mapping Element Details section, click on the line that specifies the mapping rule you are interested in.
3. In the Transformation Type field select XSLT Transformation from the drop-down list. This displays the Stylesheet URL field.
4. In the Stylesheet URL field, click Browse [...] to locate the file of the stylesheet to use.

Mapping Apama events and JMS messages

Convention-based XML mapping

Convention-based mapping allows XML documents to be created or parsed, based on a document structure encoded in the definition of the source or target Apama event type.

The first stage when using convention-based mapping is to examine the structure of the XML document, and create an event definition to represent its root element, with fields for each attribute, text node, sub-element or sequence (of attributes, text nodes or sub-elements). The actual names of the event types are not important, but the event field names and types must follow the following conventions:

- XML attributes can be represented by any EPL simple type such as `string`, `integer`, etc. The name used should be preceded by an underscore,for example `boolean _flag;`.
- XML text nodes are represented by either:
  - A field inside an Apama event representing the parent of the element containing the text, named after the element that encloses the text such as `string myelement;`. This avoids the need to create an event type to represent the element in cases where the element only contains a text node, and no attributes or children. The field type may be any primitive EPL type (for example `string`, `integer`, etc.).
  - A field inside an Apama event representing the element that directly contains the text, named `xmlTextNode`. This is necessary in cases where an Apama event type is needed to represent the element so that attributes and/or child elements can also be mapped. The field type may be any primitive EPL type (for example `string`, `integer`, etc.).
- XML elements containing attributes or sub-elements of interest are represented by a field of an event type which follows these same conventions. The event type can have any name, but the field must be named after the element, for example, `MyElementEventType myelement`.
- XML attributes, text nodes or elements which may occur more than once in the document are represented by a sequence field of the appropriate primitive or event type, named after the element, for example, `sequence<string> myelement` or `sequence<MyElement> myelement`.

Some special cases to be aware of when naming fields to match element/attribute names are:

- XML nodes which are inside an XML namespace are always referenced by their local name only (the namespace or namespace prefix is ignored).
- XML node names that are Apama EPL keywords (such as `<return>`) must be escaped in the event definition using a hash character, for example, `string #return;`. When generating an XML document, each field in the event will be processed in order and used to build up the output document. When parsing an XML document, each field in the event will be populated with whatever XML content matches the field name and type (based on the conventions above); any XML content that is not referenced in the event definition will be silently ignored.
- XML node names containing any character that is not a valid EPL identifier character (anything other than than a-z, A-Z, 0-9 and _) must be represented using a $hexcode escape sequence. Of the characters that are not valid EPL identifier characters, only the hyphen and dot are supported. Note that the hexcode based escape sequences are case sensitive. For representing the hyphen or dot use the following:
  - Hyphen '-' is represented as `$002d`.
  - Dot '.' is represented as `$002e`.

Limitations of convention-based XML mapping

In this release it is not possible to generate documents that contain elements in different XML namespaces (although when parsing this is not a problem).

The following limitations apply to the Apama event definitions that can be used to generate XML:

- Dictionary event field types are not supported
- If an event field is of type `sequence`, the sequence can contain simple types or events. The sequence cannot contain sequences of sequences or sequences of dictionaries

Mapping Apama events and JMS messages

Convention-based JMS message mapping example

The following example shows how to parse a JMS message whose body contains an XML document and map it to an Apama event called `MyEvent`.

To parse a JMS whose body contains the following XML document:

```
<?xml version='1.0' encoding='UTF-8'?>
<myroot xmlns:p='http://www.myco.com/dummy-namespace'>
  <myelement1>An element value</myelement1>
  <myelement2 myattribute='123' myboolattribute='true'>456</myelement2>
    <ignoredElement>XML content that is not included in the event definition
             is ignored</ignoredElement>
    <e1>Hello</e1>
    <e1>there</e1>
    <e-2 e2att='value1'><subElement>e2-sub-value1</subElement></e-2>
 <e-2 e2att='value2'><subElement>e2-sub-value2</subElement></e-2>
 <e1>world</e1>
 <namespacedElement xmlns='urn:xmlns:foobar'>My namespaced
       text</namespacedElement>
 <p:namespacedElement>My namespaced text 2</p:namespacedElement>
 <namespacedElement>My non-namespaced text 3</namespacedElement>
 <return>Element whose name is an EPL keyword</return>
</myroot>
```

Define the Apama event `MyEvent` as follows:

```
 event MyElement2
{
 string _myattribute;
    boolean _myboolattribute;
    string xmlTextNode;
}
event E2
{
 string _e2att;
 string subElement;
}
event MyRoot
{
 string myelement1;
 MyElement2 myelement2;
 sequence<string> e1;
 sequence<string> namespacedElement;
 string #return;
 sequence<E2> e$002d2;
}
event MyEvent
{
 string destination;
 MyRoot myroot;
}
```

Note that the field names and types matter but the event type names do not.

The document above would be parsed to the following Apama event string:

```
MyEvent("queue:MyQueue",
 MyRoot("An element value",
  MyElement2("123",true,"456"),
  ["Hello","there","world"],
  ["My namespaced text","My namespaced text 2","My non-namespaced text 3"],
  "Element whose name is an EPL keyword",
  [E2("value1","e2-sub-value1"),E2("value2","e2-sub-value2")]
))
```

The exact same event definitions could be used in the other direction for creating an XML document, although the node order will be slightly different from that of the document shown above (based on the field order) and everything would be in the same XML namespace.

Convention-based XML mapping

Using convention-based XML mapping when receiving/parsing messages

To map a received JMS message to an Apama event using the convention-based approach:

1. Create an Apama event type with fields that correspond in type and order to the structure of the XML document, and ensure that the target event type you are mapping to has a field of this type, whose field name is the root element of the expected XML document.
2. Drag a mapping line from the JMS message node containing the XML document (for example, the `JMS Body`) to the field named after the root element in the target Apama event. Assuming the JMS message contains an XML document (a string beginning with an open angle bracket character "<"), the document will be parsed and the results will be used to populate the fields of the target event.

Convention-based XML mapping

Using convention-based XML mapping when sending/generating messages

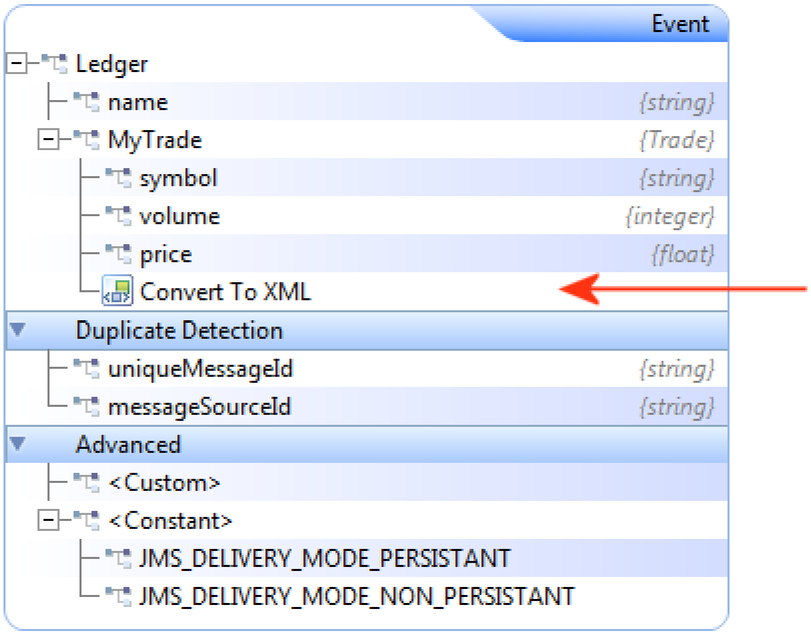
To map an Apama event to a JMS message using the convention-based approach:

1. Create an Apama event type with fields that correspond in type and order to the structure of the XML document. In order to use convention-based XML mapping, the event type representing the XML document must be nested inside a parent event type, so ensure that such a parent event type has been created. Typically the parent event type might have two fields, a string field representing the JMS destination, and an event field representing the root of the XML document.
2. In the adapter editor's Event Mappings tab, click the Add Event button (+)to add a mapping for the desired parent event type (that is, the event type which contains the event field representing the XML root element).
3. In the adapter editor's Event Mappings tab, right-click the Apama event representing the root node of the XML document and select Add Computed Node. The **Add Computed Node** dialog is displayed.
4. In the **Add Computed Node** dialog's Select Method field, select Convert to XML from the drop-down list. The dialog is updated to show more information.

   You can specify a namespace and namespace prefix for the generated XML document if desired, or else leave them blank. By default the Include empty fields option is enabled. This specifies that empty XML nodes will be generated when empty EPL string fields are encountered within an

Apama event. This option does not affect empty strings within a sequence of EPL strings. If you clear the check box to disable the option, empty XML nodes will not be generated.

5. Click OK.
6. In the mapping tree, Apama Studio adds an entry of type `Convert To XML` to the selected event node.



Event

Ledger	
name	{string}
MyTrade	{Trade}
symbol	{string}
volume	{integer}
price	{float}
Convert To XML	
Duplicate Detection	
uniqueMessageId	{string}
messageSourceId	{string}
Advanced	
<Custom>	
<Constant>	
JMS_DELIVERY_MODE_PERSISTANT	
JMS_DELIVERY_MODE_NON_PERSISTANT	

7. Drag a mapping line from the `eventToXml` entry to the desired node in the XML message, for example, to the `JMS Body`.

Convention-based XML mapping

Combining convention-based XML mapping with template-based XML generation

Note that for maximum flexibility it is also possible to combine the convention-based approach with template-based XML generation. An XML template can be used to generate the top-level XML document, while one or more placeholders can be added which are mapped to XML sub-document strings generated by convention-based XML mapping.

To combine these approaches, simply right-click a source event (representing an XML root element) or sequence (representing a list of XML elements) and click Add Computed Node; then select Convert to XML from the drop-down list and click OK. This will result in an `Convert To Xml` node representing a generated XML string, then drag a mapping line from that node to the target `$placeholder` node specifying where the XML snippet should be inserted into the top-level document.

Convention-based XML mapping

Dynamic senders and receivers

In addition to specifying static senders and receivers in the adapter's configuration file as introduced in "Getting started - creating an application with simple JMS messaging" on page 271, you can dynamically add and manage senders and receivers using actions on Apama's `JMSConnection` event.

(Note, for more information on static senders and receivers, see "Adding static senders and receivers" on page 303.)

The unique identifiers specified when adding dynamic senders or receivers must not clash with the identifiers used for any static senders and receivers in the configuration file. You cannot dynamically remove a sender or receive that was defined statically in the configuration file; only dynamically added senders and receivers can be removed.

It is currently valid to emit events to the channel associated with a newly created dynamic sender as soon as the add action has returned. In this case Correlator-JMS ensures that the events get sent to the JMS broker eventually. However, best practice is to add a listener for `JMSSenderStatus` and wait for the `OK` status before beginning to emit to a dynamic sender. It valid to emit events to an existing sender's channel at any point until its removal is requested by calling the `remove()` action. It is not valid to emit any events to that channel after `remove()` has been called, and any events emitted after this point are in doubt and could be ignored without any error being logged. Applications that make use of multiple contexts may need to co-ordinate across contexts to ensure that no `emit` or other operations are performed on senders that have been removed in another context.

For more information on dynamically adding senders and receivers, see the `JMSConnection` event documentation in the ApamaDoc documentation.

The example Apama application located in the `*apama_dir*\samples\correlator_jms\dynamic-event-api` directory demonstrates how to use the event API to dynamically add and remove JMS senders and receivers. In addition, it shows how to monitor senders and receivers for errors and availability.

Correlator-Integrated JMS Messaging

Durable topics

JMS durable topic subscriptions are supported for both static and dynamic receivers, enabling an Apama application to persistently register interest in a topic's messages with the JMS broker, such that if the correlator is down, messages sent to the topic will be held ready for delivery when the correlator recovers.

Statically configured durable topic subscriptions cannot be removed. When a dynamic receiver using a durable topic subscription is removed, the JMS subscription to the topic will be removed at the same time, before the `REMOVED` receiver status notification event is sent. A consequence of this is that the removal of a receiver will not be completed until the JMS connection is up, in order that the subscription can be removed from the JMS broker. Note that durable topic subscriptions cannot be created using `BEST_EFFORT` receivers.

The preferred method of subscribing to a durable topic is to use the `addReceiverWithDurableTopicSubscription` (or `addReceiverWithConfiguration`) action on the `com.apama.correlator.jms.JMSConnection` event. For more information on these actions, see the `JMSConnection` event documentation in the ApamaDoc documentation.

Correlator-Integrated JMS Messaging

Receiver flow control

It is possible to give an EPL application control over the rate at which events are taken from the JMS queue or topic by each JMS receiver. To enable this option, set the `receiverFlowControl` property to true in the JmsReceiverSettings bean. The configuration for this bean is found in the jms-global-spring.xml file. To display the file in Apama Studio, select the Advanced tab of the Correlator-JMS configuration editor.

Once receiverFlowControl has been enabled, use the com.apama.correlator.jms.JMSReceiverFlowControlMarker event to enable receiving events from each receiver, by specifying a non-zero window size. For example, to ensure that each receiver will never add more than 5000 events to the input queue of each public context, add the following EPL code

```
using com.apama.correlator.jms.JMSReceiverFlowControlMarker;
...
JMSReceiverFlowControlMarker flowControlMarker;
on all JMSReceiverFlowControlMarker(): flowControlMarker
{
    flowControlMarker.updateFlowControlWindow(5000);
}
```

A flow control marker is an opaque event object that is always enqueued (to the correlator's public contexts) when a new receiver is first added and during recovery of a persistent correlator. The message is also enqueued regularly as new messages are received and mapped, which typically happens at the end of each received batch, for example, at least once every 1000 successfully-mapped events if the default setting for maxBatchSize is used. The marker event indicates a specific point in the sequence of events sent from each receiver, and the application must always respond by calling the updateFlowControlWindow action on this marker event. This sets the size of the window of new events the receiver is allowed to take from the JMS queue or topic, relative to the point indicated by the marker.

More advanced applications that need to block JMS receivers until asynchronous application-specific operations arising from the processing of received messages (such as database writes, messaging sending, etc.) have completed can factor the number of pending operations into the flow control window. To reliably do this, it is necessary to stash the marker events for each receiver in a dictionary and add logic to call updateFlowControlWindow when the number of pending operations falls, so that any receivers that were blocked due to those operations can resume receiving. It is the application's responsibility to ensure that receivers do not remain permanently blocked, by calling updateFlowControlWindow sufficiently often. For an example of how receiver flow control can be used together with asynchronous per-event operations, see the Correlator-JMS 'flow-control' sample application located in the *apama_dir*\samples\correlator_jms directory.

Applications must make sure that they listen for all JMSReceiverFlowControlMarker events, and that their listener for the flow control markers is set up before JMS.onApplicationInitialized is called. Any stale or invalid JMSReceiverFlowControlMarker event, for example, from before a persistent correlator was restarted, cannot be used to update the flow control window, and any calls on such stale events will simply be ignored.

Full ApamaDoc documentation for the JMSReceiverFlowControlMarker event is available together with the rest of the Correlator-JMS event API. See the ApamaDoc documentation.

The current window size for all receivers is indicated by the rWindow item in the "JMS Status" lines that are periodically logged by the Correlator, and this may be a useful debugging aid if receivers appear to be blocked indefinitely.

Correlator-Integrated JMS Messaging

Monitoring Correlator-integrated JMS status

Apama applications often need to monitor the status of JMS connections, senders, and receivers when the application needs to wait for a receiver or sender to be available (status "OK") before using it, and, conversely, to detect and report error conditions.

The main way to monitor status is to simply set up EPL listeners for the JMSConnection, JMSReceiverStatus, and JMSSenderStatus events which are sent to all public correlator contexts automatically, both on startup and whenever the status of these items changes. Note that there is no need to 'subscribe' to receive these events — provided JMS.onApplicationInitialized() was called, these events will be sent automatically, so all that is required is to set up listeners.

Some applications, especially those built using scenarios rather than EPL, prefer to monitor status using the standardized event API defined by StatusSupport.mon. The CorrelatorJMSStatusManager monitor, which is part of the Correlator-Integrated JMS bundle, acts as a bridge between the JMS-specific status events and this API, to allow Apama applications to monitor the status of JMS connections, senders and receivers using the standard Status Support interface. To use this interface:

1. Send a com.apama.statusreport.SubscribeStatus event, which is defined as:

   ```
   event SubscribeStatus {
     string serviceID;
     string object;
     string subServiceID;
     string connection;
     }
   ```

   The fields for the SubscribeStatus event are:

   - serviceID - This should be set to CORRELATOR_JMS.
   - object - Can be CONNECTION, RECEIVER, or SENDER, or "" (empty string). If "" is specified, the application will subscribe to status events for all connections, receivers, and senders.
   - subServiceID - The name of a specific receiver or sender if RECEIVER or SENDER is specified in the object field. If the object field specifies RECEIVER or SENDER, the subServiceID field must have a valid, non-empty value. If the object field specifies CONNECTION this field must be "".
   - connection - The name of a specific connection. If the object field specifies a value, the connection field must have a valid, non-empty value.

2. Create listeners for com.apama.statusreport.Status events (and optionally for StatusError events which are sent if the status subscription failed due to an invalid identifier being specified).

3. To unsubscribe, send an UnsubscribeStatus event with field values that match the corresponding SubscribeStatus event.

For more information on monitoring Correlator-integrated JMS connections, receivers, and senders, see the descriptions of the JMSConnectionStatus, JMSReceiverStatus, and JMSSenderStatus events in the ApamaDoc documentation.

Correlator-Integrated JMS Messaging

Logging JMS enabled correlator status

The correlator writes status information to its log file every five seconds or at an interval that you set with the --logQueueSizePeriod option. In addition to the standard correlator status information described in "Logging correlator status" on page 111, correlators with JMS support enabled log JMS-specific status information. This information is logged in the following form:

```
INFO  [20032] - Correlator Status: sm=2 nctx=1 ls=4 rq=0 eq=0 iq=0 oq=0 rx=8
                tx=6 rt=0 nc=1 vm=251384 runq=0
INFO  [20032:Status] - JMS Status: s=1 tx=6 sRate=1,200 sOutst=9,005 r=2
                rx=4 rRate=1,180 rWindow=-1 rRedel=3 rMaxDeliverySecs=2.1
                rDupsDet=2 rDupIds=2,005,023 connErr=2 jvmMB=49
```

The JMS-specific fields are as follows:

Table 4. JMS-enabled correlator status log fields

Field	Full name	Description
`s`	Number of senders	The current number of JMS senders (both static and dynamic) on all JMS connections.
`tx`	Sent events	The total number of events emitted to a sender that have been fully processed (either sent to JMS or exhausted the maximum failure retry limit). Includes events sent to dynamic senders that have since been removed, but does not include events sent before the correlator was restarted.
`sRate`	Send throughput rate	The total number of events sent per second across all senders, calculated over the interval since the last status line (typically 5 seconds).
`sOutst`	Outstanding sent events	The total number of events that have been emitted by EPL but are still queued waiting to be sent to JMS.
`r`	Number of receivers	The current number of JMS receivers (both static and dynamic) on all JMS connections.
`rx`	Received messages	The total number of messages received from JMS, including messages received but not yet mapped to Apama events and added to the input queue of each public context. Includes events received by dynamic receivers that have now been removed, but does not include events received before the correlator was restarted, nor does it include any `JMSReceiverFlowControlMarker` events enqueued when the `receiverFlowControl` is enabled.
`rRate`	Received throughput rate	The total number of JMS messages received per second across all receivers, calculated over the interval since the last status line (typically 5 seconds).
`rWindow`	Receiver flow control window size	If `receiverFlowControl` is disabled for all receivers, this has the special value "`-1`". If any receivers have flow control enabled, `rWindow` gives a measure of the number of events that can be received before all flow controlled receivers will block, calculated as the sum of all the non-

Field	Full name	Description
		negative receiver window sizes. Note that even if this value is greater than 0 there could still be one or more receivers which have exhausted their own windows and are blocked, so consider enabled `logDetailedStatus` if *per-receiver* flow control diagnostics are required.
`rRedel`	Redelivered messages	The total number of JMS messages received with the `JMSRedelivered` flag set to true, indicating they are in-doubt and may have already been delivered in the past. For many JMS providers this flag is not always set reliably/consistently, but it does at least provide an indication of whether redeliveries may be taking place.
`rMaxDeliverySecs`	Maximum delivery time	The highest time taken by the JMS broker to deliver a message, based on the difference between the time when each message is received and the value of the `JMSTimestamp` message header field which indicates the time when it was sent. This is likely to be a low number during normal operation, but will rise during failure modes such as loss of network connectivity or machine crashes as the JMS broker attempts to redeliver messages. This value is useful for understanding the redelivery behavior of the JMS provider in use and for choosing a sensible time expiry window if `EXACTLY_ONCE` duplicate detection is being used (see `dupDetectionExpiryTimeSecs` property). A high `rMaxDeliverySecs` value during testing may indicate that messages remaining on a JMS queue or durable topic from a previous test run may be interfering with the current test run. Note that any difference in the system time on the sending and receiving hosts will add an error to this value, which can result in negative values.
`rDupsDet`	Duplicate messages detected	The total number of duplicate messages detected by `EXACTLY_ONCE` receivers and suppressed because their `uniqueMessageId` was already present in the duplicate detector. Does *not* include dynamic receivers that have now been removed.
`rDupIds`	Duplicate ids in memory	The total number of `uniqueMessageId`s being kept in memory for duplicate detection purposes by `EXACTLY_ONCE` reliable receivers. This is the total of the size of all per-message-source fixed-

Field	Full name	Description
		size expiry queues plus the unbounded time-based expiry queue. If this becomes too large it is possible the correlator could run out of memory.
`connErr`	Connection errors	The total number of times a valid JMS connection has gone down. Note that this tracks errors in existing connections and does not include repeated failures to establish a connection.
`jvmUsedMB`	JVM used memory	The amount of memory used by the JVM in Megabytes (heap plus non-heap), which can be compared with the maximum memory size provided for the JVM to check how much spare memory there is and ensure that the correlator is not close to running out of memory. This is particularly useful for checking peak memory consumption, such as testing when any `EXACTLY_ONCE` duplicate detectors are fully populated with the maximum likely number of `uniqueMessageId`s and any Apama Java (JMON) applications are running in the same correlator are also near the maximum memory they are likely to use. Note that the memory usage figure reported by the JVM includes both live objects and *objects waiting to be garbage collected*, so inevitably this will go up and down a certain amount as garbage collections occur.
`...`	onApplication Initializedindicator	The suffix `<waiting for onApplicationInitialized>` will be added to the status lines if the EPL application has not yet called `jms.onApplicationInitialized()` as a reminder that status or JMS message events cannot be passed into the correlator until this action is invoked.

Detailed JMS status lines

If the `logDetailedStatus` property in an Apama application that uses Correlator-integrated JMS messaging is set to true in the `JmsSenderSettings` or `JmsReceiverSettings` configuration object, then additional lines will also be logged for each sender and receiver and their parent connections, for example.

```
INFO  [19276] - Correlator Status: sm=2 nctx=1 ls=4 rq=0 eq=0 iq=0 oq=0
                rx=8 tx=6 rt=0 nc=1 vm=252372 runq=0
INFO  [19276:Status] - JMS Status: s=1 tx=6 sRate=0 sOutst=0 r=2 rx=4
                         rRate=0 rWindow=1500 rRedel=0
                         rMaxDeliverySecs=0.0 rDupsDet=1 rDupIds=3
                         connErr=0 jvmMB=67
INFO  [19276:Status] -     JMSConnection myConnection: s=1 r=2 connErr=0
                             sessionsCreated=3
INFO  [19276:Status] -       JMSSender   myConnection-default-sender: tx=6
                               sRate=0 sOutst=0 msgErrors=2
```

```
INFO  [19276:Status] -       JMSReceiver myConnection-receiver-SampleQ2:
                               rx=4 rRate=0 rWindow=1500 rRedel=0
                               rMaxDeliverySecs=0.0 msgErrors=1 rDupsDet=1
                               perSourceDupIds=3 timeExpiryDupIds=0
INFO  [19276:Status] -       JMSReceiver myConnection-receiver-SampleT2:
                               rx=0 rRate=0 rWindow=-1 rRedel=0
                               rMaxDeliverySecs=0.0 msgErrors=0 rDupsDet=0
                               perSourceDupIds=0 timeExpiryDupIds=0
```

The JMS connector-specific status lines contain:

Table 5. JMS connector status log fields

Field	Full name	Description
`s`	Number of senders	The current number of JMS senders (both static and dynamic) on all JMS connections.
`r`	Number of receivers	The current number of JMS receivers (both static and dynamic) on all JMS connections
`connErr`	Connection errors	The total number of times this JMS connection has gone down, Note that this tracks errors in existing connections and does not include repeated failures to establish a connection.
`sessionsCreated`	Send/receive sessions created	The total number of JMS Sessions that have been created during the lifetime of this JMS connection. In normal operation a single session is created for each sender or receiver, but if a connection failure or serious sender/receiver error occurs, a new session will be created, causing this counter to be incremented. Note, this counter is not decremented when the previous session is closed.

The JMS sender-specific status lines contain:

Table 6. JMS sender status log fields

Field	Full name	Description
`tx`	Sent events	The number of events that were emitted to this sender and have been fully processed (either sent to JMS, or exhausted the maximum failure retry limit).
`sRate`	Send throughput rate	The number of events sent per second to this sender, calculated over the interval since the last status line (typically 5 seconds).
`sOutst`	Outstanding sent events	The number of events that have been emitted by EPL but are still queued waiting to be passed to JMS by this sender.

Field	Full name	Description
sMsgErrors	Per-message error count	The number of Apama events that could not be sent to JMS due to some error, typically a mapping failure or destination not found error. See the log file for WARN and ERROR messages that will provide more details.

The JMS receiver-specific status lines contain:

Table 7. JMS receiver status log fields

Field	Full name	Description
rx	Received messages	The number of messages received from JMS, including messages received but not yet mapped to Apama events and added to the input queue of each public context.
rRate	Receive throughput rate	The number of JMS messages received per second by this receiver, calculated over the interval since the last status line (typically 5 seconds).
rWindow	Flow control window size	If receiverFlowControl is disabled this has the special value "-1". If it is enabled, this gives the current flow control window size, that is the number of successfully mapped events that can still be received before the receiver will block. A zero value indicates that the flow control window has been exhausted and the application should call JMSReceiverFlowControlMarker.updateFlowControlWindow() to unblock the receiver. A negative value indicates that the window has been updated to a negative value, which has the same effect as a window of 0.
rRedel	Redelivered messages	The number of JMS messages received with the JMSRedelivered flag set to true.
rMaxDeliverySecs	Maximum delivery time	The highest time taken by the JMS broker to deliver a message to this receiver, based on the difference between the time when each message is received and the value of the JMSTimestamp message header field which indicates the time when it was sent.
rMsgErrors	Per-message error count	The number of received JMS messages that could not be passed to the Apama application due to some error, typically a mapping failure. See the log file for WARN and ERROR messages that will provide more details.
rDupsDet	Duplicate messages detected	The number of duplicate messages detected by this EXACTLY_ONCE receiver and suppressed because their uniqueMessageId was already present in this receiver's

Field	Full name	Description
		duplicate detector. Only displayed for EXACTLY_ONCE receivers.
perSourceDupIds	Per-source duplicate ids in memory	The total number of uniqueMessageIds being kept in memory for duplicate detection purposes by all per-message-source fixed-size expiry queues. Only displayed for EXACTLY_ONCE receivers.
timExpiryDupIds	Time-based duplicate ids in memory	The total number of uniqueMessageIds being kept in memory for duplicate detection purposes by the unbounded time-based expiry uniqueMessageId queue.

Correlator-Integrated JMS Messaging

JUEL Mapping Expressions Reference

The expressions that can be used to get or set elements of a JMS message are listed below, along with the set of Apama field types that are recommended for use when mapping when sending or receiving JMS messages:

JMS message element	JMS EL expression	Compatible Apama field type(s) when sending	Compatible Apama field type(s) when receiving
Dictionary of all message headers	jms.headers	dictionary<string, string>	dictionary<string, string>
JMSDestination	jms.header['JMSDestination']	string (with jndi:/ topic:/queue: prefix)	string (with topic:/ queue: prefix)
JMSReplyTo	jms.header['JMSReplyTo']	string (with jndi:/ topic:/queue: prefix)	string (with topic:/ queue: prefix)
JMSCorrelationID	jms.header['JMSCorrelationID']	string	string
JMSType	jms.header['JMSType']	string	string
JMSPriority	jms.header['JMSPriority']	integer, string	integer, string
JMSDeliveryMode	jms.header['JMSDeliveryMode']	integer, string (must be a number (though display string can be used (only) when	integer, string

		mapping a constant value in tooling); 1=NON_PERSISTENT, 2=PERSISTENT)	
JMSTimeToLive	`jms.header[ 'JMSExpiration']` (* will be changed to JMSTimeToLive in future version)	`integer, string` (in milliseconds from the time JMS sends the message)	N/A when receiving
JMSExpiration	`jms.header[ 'JMSExpiration']`	N/A when sending	`integer, string` (in milliseconds since the epoch)
JMSMessageID	`jms.header[ 'JMSMessageID']`	N/A when sending	`boolean, string`
JMSTimestamp	`jms.header[ 'JMSTimestamp']`	N/A when sending	`integer, string` (in milliseconds since the epoch)
JMSRedelivered	`jms.header[ 'JMSRedelivered']`	N/A when sending	`string`
Dictionary of all message properties	`jms.properties`	`dictionary<string, string>`	`dictionary<string, string>`
String Message Property	`jms.property[ 'propName']`	`string`	`string`
Boolean Message Property	`jms.property[ 'propName']`	`boolean`	`boolean, string`
Long Message Property	`jms.property[ 'propName']`	`integer`	`integer, string`
Double Message Property	`jms.property[ 'propName']`	`float`	`float, string`
Byte Message Property	`jms.property[ 'propName']`	Not supported	`string`
Short Message Property	`jms.property[ 'propName']`	Not supported	`string`
Integer Message Property	`jms.property[ 'propName']`	Not supported	`string`
Float Message Property	`jms.property[ 'propName']`	Not supported	`string`

JMSX Property	`jms.xproperty['propName']`	same as other properties	same as other properties
Dictionary of all JMSX properties	`jms.xproperties`	`dictionary<string, string>`	`dictionary<string, string>`
TextMessage Body	`jms.body.textmessage`	`string, event`[1]	`string, event`[1]
MapMessage Body	`jms.body.mapmessage`	`dictionary<string, string>`	`dictionary<string, string>`
MapMessage Body Entry	`jms.body.mapmessage['mapKey']`	`string`	`string`
ObjectMessage Body with a serializable `java.util.Map<Object,Object>`	`jms.body.objectmessage`	`dictionary<string, string>`	`dictionary<string, string>`
ObjectMessage Body with a serializable `java.util.List<Object>`	`jms.body.objectmessage`	`sequence<string>`	`sequence<string>`
ObjectMessage Body with any serializable Object	`jms.body.objectmessage`	N/A	`string`
BytesMessage Body	`jms.body.bytesmessage`	`string, sequence<string>, dictionary<string, string>, event`	`string, sequence<string>, dictionary<string, string>`
TextMessage, MapMessage, BytesMessage, ObjectMessage, Message	`jms.body.type`	`string`	`string`

[1] If a string from the JMS message is mapped to an event, the string should be either of:

- An Apama event string (as generated by the Apama Event Parser), whose event type matches the type of the field it is being mapped to in the source/target Apama event.
- An XML document starting with a < character, whose structure matches what is implied by the event type definition it is being mapped to (see "Convention-based XML mapping" on page 286 for more information)

Note, the JMS headers JMSMessageID, JMSRedelivered and JMSDeliveryMode are supported for completeness but will not normally be required by Apama applications, since built-in duplicate detection based on application-level unique identifiers replaces the first two, and rather than overriding the per-message delivery mode it is usually best to use the default PERSISTENT/ NON_PERSISTENT setting implied by the sender's senderReliability value.

Correlator-Integrated JMS Messaging

JMS configuration reference

This section includes topics relating to the configuration for applications using Apama's correlator-integrated JMS messaging. It covers configuration files, configuration objects, and the configuration properties that can be set when developing your applications.

Correlator-Integrated JMS Messaging

Configuration files

The correlator-integrated JMS configuration consists of a set of XML files and .properties files.

A correlator with JMS support enabled will have the following two files:

- jms-global-spring.xml
- jms-mapping-spring.xml

In addition, for each JMS connection added to the configuration, there will be an additional XML and .properties file:

- *connectionId*-spring.xml
- *connectionId*-spring.properties

When the correlator is started with the --jmsConfig *configDir* argument, it will load all XML files matching *-spring.xml in the specified configuration directory, and also all *.properties files in the same directory. (Note, the correlator will not start unless the specified directory contains at least one configuration file.)

Global configuration that is shared across all a correlator's connections is stored in jms-global-spring.xml, the rules for mapping between JMS messages and Apama events are stored in jms-mapping-spring.xml, and the *connectionId*-spring.xml files contain the configuration for each JMS broker connection added to the configuration. Each XML file can contain ${...} property placeholders, whose values come from the *.properties files. This provides a way for the configuration to be defined once in the XML files, then customized for development, UAT, and different deployment scenarios by creating separate copies of the .properties files.

When using Apama Studio, all these files are generated automatically. A new *connectionId*-spring.xml and *connectionId*-spring.properties file is created when the **JMS Configuration Wizard** is used to add a JMS connection, and the most commonly used settings can be changed at any time using the correlator-integrated JMS instance editor window (which rewrites the .properties file whenever the configuration is changed). Apama Studio makes it easy to set and edit basic configuration options with the adapter editor. In addition, the jms-global-spring.xml and *connectionId*-spring.xml files can be edited manually in Apama Studio to customize more advanced configuration aspects such as advanced sender/receiver settings, logging of messages, etc. To edit the XML, open the correlator-integrated JMS editor and click on the Advanced tab; the various configuration files can be accessed through the hyperlinks on this tab. Once the editor for an XML file has been opened, you can switch between the Design and Source views using the tabs at the bottom of the editor window.

Note that unlike the other XML files, Apama does not support manual editing of the jms-mapping-spring.xml file in this release, and the format of that file may change at any time without notice. We recommend using Apama Studio for all mapping configuration tasks.

JMS configuration reference

XML configuration file format

The Correlator-integrated JMS configuration files use the Spring XML file format, which provides an open-source framework for flexibly wiring together the difference parts of an application, each of which is represented by a *bean*. Each bean is configured with an associated set of *properties*, and has a unique identifier which can be specified using the id= attribute.

For example:

```
<bean id="globalReceiverSettings"
      class="com.apama.correlator.jms.config.JmsReceiverSettings">
    <property name="logJmsMessages" value="true"/>
    <property name="logProductMessages" value="false"/>
</bean>
```

Or:

```
<jms:receiver id="myReceiver1">
    <property name="destination" value="queue:SampleQ1"/>
</jms:receiver>
```

It is not necessary to have a detailed knowledge of Spring to configure correlator-integrated JMS, but some customers may wish to explore the Spring 3.0.5 documentation to obtain a deeper understanding of what is going on and to leverage some of the more advanced functionality that Spring provides.

The key beans making up the Apama configuration are jms:connection, jms:receiver and jms:sender, plus additional beans that are usually stored in the jms-global-spring.xml file and shared across all configured connections, such as the reliable receive database and the advanced sender/receiver settings beans.

Bean ids

All receiver, sender, connection, and other configuration beans have an "id=" attribute that specifies a unique identifier. These identifiers are used in log messages, when monitoring status from EPL applications, and, when necessary, for references between different Spring beans in the XML configuration files. It is important that all identifiers are completely unique, for example the same id cannot be used for senders and receivers in different connections, or for both a sender and a receiver, even if they located in different XML files.

Setting property values

Most bean properties have primitive values (such as string, number, boolean) which are set like this:

```
<property name="propName" value="my value"/>
```

However, there are also a few properties that reference other beans, such as the reliableReceiveDatabase property on jms:connection and the receiverSettings property on jms:receiver. These property values can be set by specifying the id of a top-level bean like this (where it is assumed that globalReceiverSettings is the id of a JmsReceiverSettings bean):

```
<property name="receiverSettings" ref="globalReceiverSettings"/>
```

Any top-level bean may be referenced in this way, that is, any bean that is a child of the <beans> element and not nested inside another bean. Referencing a bean that is defined in a different configuration file is supported, and the `jms-global-spring.xml` file is intended as a convenient place to store top-level beans that should be shared across many different JMS connections.

Instead of referencing a shared bean, it is also possible to configure a bean property by creating an 'inner' configuration bean nested inside the property value like this:

```
<property name="receiverSettings">
    <bean class="com.apama.correlator.jms.config.JmsReceiverSettings">
        <property name="logJmsMessages" value="true"/>
    </bean>
</property>
```

(Note, advanced users may want to exploit Spring's property inheritance by using the `parent=` attribute on an inner bean to inherit most properties from a standard top-level bean while overriding some specific subset of properties or by type-based 'auto-wiring' - any non-primitive property of `jms:connection`/`receiver`/`sender` for which no value is explicitly set will implicitly reference a top-level bean of the required type. This is how `jms:connection` beans get a reference to the `reliableReceiveDatabase` and `defaultSender`/`ReceiverSettings` beans. Most configuration can just ignore this detail and use the automatically wired property values, but if desired the defaults for individual connections/senders/receivers can be customized independently of each other by specifying the property values explicitly.)

Adding static senders and receivers

For simple cases where detailed configuration of receivers is not required, it is possible to configure static receivers using a simple semicolon-delimited list of JMS destinations, for example:

```
<property name="staticReceiverList"
          value="topic:MyTopic;jndi:/sample/some-jndiqueuename" />
```

The `staticReceiverList` bean property is represented by a placeholder in the *`connectionId`*`-spring.properties` file, and can be edited using Apama Studio.

For more advanced receiver configuration, it is necessary to edit the *`connectionId`*`-spring.xml` file manually, and provide a list of `jms:receiver` beans as the value of the `staticReceivers` property:

```
<property name="staticReceivers">
    <list>
        <jms:receiver id="myReceiver1">
            <property name="destination"
                      value="queue:SampleQ1"/>
        </jms:receiver>
        <jms:receiver id="myReceiver2">
            <property name="destination"
                      value="jndi:/sample/my-jndi-topic-name"/>
            <property name="durableTopicSubscriptionName"
                      value="MyTopicSubscription"/>
        </jms:receiver>
    </list>
</property>>
```

Senders may be configured in the same way, for example:

```
<property name="staticSenders">
    <!-- each static sender results in a Correlator emit channel
         called "jms:senderId" -->
    <list>
        <jms:sender id="MyConnection-default-sender">
        </jms:sender>
        <jms:sender id="myReliableSender">
            <property name="senderReliability" value="EXACTLY_ONCE"/>
        </jms:sender>
        <jms:sender id="myUnreliableSender">
            <property name="senderReliability" value="BEST_EFFORT"/>
        </jms:sender>
    </list>
</property>
```

If a sender list is not explicitly configured, a single sender with id *`connectionId`*`-default-sender` will be created.

JMS configuration reference

XML configuration bean reference

This topic lists the various configuration objects (beans) and the supported properties for each bean.

jms:connection

This bean defines the information needed to establish a JMS Connection to a single JMS broker instance. Its required properties are: `connectionFactory` or `connectionFactory.jndiName`, and (if JNDI is used to locate the connection factory), `jndiContext`.

Example:

```
<jms:connection id="MyConnection">
    <property name="staticReceiverList"
              value="${staticReceiverList.MyConnection}" />
    <property name="defaultReceiverReliability"
              value="${defaultReceiverReliability.MyConnection}"/>
    <property name="defaultSenderReliability"
              value="${defaultSenderReliability.MyConnection}"/>
    <property name="connectionFactory.jndiName"
              value="${connectionFactory.jndiName.MyConnection}" />
    <property name="jndiContext.environment">
        <value>
            ${jndiContext.environment.MyConnection}
        </value>
    </property>
    <property name="connectionAuthentication.username"
              value="${connectionAuthentication.username.MyConnection}" />
    <property name="connectionAuthentication.password"
              value="${connectionAuthentication.password.MyConnection}" />
</jms:connection>
```

Supported properties:

- `connectionFactory.jndiName` - the JNDI lookup name for the `ConnectionFactory` object that should be used for this `jms:connection`.
- `connectionFactory` - a JMS provider bean that implements the JMS `ConnectionFactory` interface, if the `ConnectionFactory` is to be instantiated directory by the Spring framework (rather than using JNDI to lookup the `ConnectionFactory`). The bean value that is provided will usually required properties and/or constructor arguments to be specified in order to fully initialize it.
- `connectionAuthentication.username` - the name of the user/principal to be used for the JMS connection (note that this is often different from the username/password needed to login to the JNDI server, which is part of the JNDI environment configuration). Default value is "".
- `connectionAuthentication.password` - the password/credentials to be used for the JMS connection.
- `jndiContext.environment` - the set of properties that specify the environment for initializing access to the JNDI store. Typically includes some standard JNDI keys such as `java.naming.factory.initial`, `java.naming.provider.url`, `java.naming.security.principal` and

java.naming.security.credentials, and maybe also some provider-specific keys. The usual way to specify a properties map value is key=value entries delimited by newlines and surrounded by the <value> element, e.g. <property name="jndiContext.environment"><value>...</value><</property>.

- clientId - the JMS client ID which uniquely identifies each connected JMS client to the broker. Default value is "" although some JMS providers may require this to be set, especially when using durable topics.
- defaultReceiverReliability - the Apama reliability mode to use for all this connection's receivers unless overridden on a per-receiver basis; valid values are BEST_EFFORT, AT_LEAST_ONCE, EXACTLY_ONCE. Default value is BEST_EFFORT.
- defaultSenderReliability - the Apama reliability mode to use for all this connection's senders unless overridden on a per-sender basis; valid values are BEST_EFFORT, AT_LEAST_ONCE, EXACTLY_ONCE. Default value is BEST_EFFORT.
- staticReceiverList - a list of destinations to receive from, delimited by semi-colons. Each destination must begin with "queue:", "topic:" or "jndi:". This property provides a simple way to add static receivers when the more advanced configuration options provided by the staticReceivers property are not needed. staticReceiverList receivers are always added in addition to any receivers specified by staticReceivers. The staticReceiverList property cannot contain duplicate destination entries (see the staticReceivers property if this is required). Default value is "".
- staticReceivers - a list of jms:receiver beans specifying JMS receivers to create for this connection. The jms:receiver elements are wrapped in a <list> element, for example, <property name="staticReceivers"><list>...</list></property>. Default value is an empty list.
- staticSenders - a list of sender beans specifying JMS senders to create for this connection. The jms:sender elements are wrapped in a <list> element, for example, <property name="staticSenders"><list>...</list></property>. Default value is a single sender called "default".
- defaultReceiverSettings (advanced users only) - a reference to a JmsReceiverSettings bean, which provides access to advanced settings that are usually shared across all configured receivers for this connection. Default value is a reference to the JmsReceiverSettings bean instance defined in the jms-global-spring.xml file (this uses Spring's byType auto-wiring; if multiple top-level JmsReceiverSettings beans exist in the configuration then the reference must be specified explicitly in each jms:connection).
- defaultSenderSettings (advanced users only) - a reference to a JmsSenderSettings bean, which provides access to advanced settings that are usually shared across all configured senders for this connection. Default value is a reference to the JmsSenderSettings bean instance defined in the jms-global-spring.xml file (this uses Spring's byType auto-wiring; if multiple top-level JmsSenderSettings beans exist in the configuration then the reference must be specified explicitly in each jms:connection).
- reliableReceiveDatabase (advanced users only) - a reference to a ReliableReceiveDatabase bean, which is required for implementing the AT_LEAST_ONCE and EXACTLY_ONCE reliability modes for any receivers added to this jms:connection. Default value is a reference to the single DefaultReliableReceiveDatabase bean instance defined in the jms-global-spring.xml file (this uses Spring's byType auto-wiring; if multiple top-level DefaultReliableReceiveDatabase beans exist in the configuration then the reference must be specified explicitly in each jms:connection). The only reason for changing this property would be to use separate databases or different jms:connection which could in some advanced cases provide a performance

advanced, depending on the application architecture and the configuration of the jms:connection and disk hardware.

- connectionRetryIntervalMillis - Specifies how long to wait between attempts to establish the JMS connection. Default value is 1000 ms.

jms:receiver

This bean defines a single-threaded context for receiving events from a single JMS destination. Its only required property is "destination".

Example:

```
<jms:receiver id="myReceiver">
    <property name="destination" value="topic:SampleT1"/>
</jms:receiver>
```

Supported properties:

- destination - the JMS queue or topic to receive from. Must begin with the prefix "queue:", "topic:" or "jndi:". A JMS queue or topic name can be specified with the "queue:" or "topic:" prefixes, or if the queue or topic should be looked up using a JNDI name then the "jndi:" prefix should be used instead.
- receiverReliability - the Apama reliability mode to use when JMS messages are received; valid values are BEST_EFFORT, AT_LEAST_ONCE, EXACTLY_ONCE. Default value is provided by the parent jms:connection's defaultReceiverReliability setting.
- durableTopicSubscriptionName - if specified, a durable topic subscriber will be created (instead of a queue/topic consumer), and registered with the specified subscription name. Default value is "", which means do not create a durable topic subscription. Note that some providers will require the connection's clientId property to be specified when using durable topics.
- messageSelector - a JMS message selector string that will be used by the JMS provider to filter the messages pulled from the queue or topic by this receiver, based on the header and/or property values of the messages. Default value is "" which means that no selector is in operation and all messages will be received. Message selectors can be used to partition the messages received by multiple receivers on the same queue or durable topic. The JMS API documentation describes the syntax of message selectors in detail; a simple example selector is "JMSType = 'car' AND color = 'blue' AND weight > 2500".
- noLocal - an advanced JMS consumer parameter that prevents a connection's receivers from seeing messages that were sent on the same (local) JMS connection. Default value is "false".
- dupDetectionDomainId - an advanced Apama setting for overriding the way receivers are grouped together for duplicate detection purposes when using EXACTLY_ONCE receive mode. Set this to the same string value for a set of receivers to request detection of duplicate uniqueMessageIds across all the messages from those receivers. Default value is "<connectionId>:<destination>" (that is, look for duplicates across all receivers for the same queue/topic only within the same jms:connection).
- receiverSettings - a reference to a JmsReceiverSettings bean, which provides access to advanced settings that are usually shared across all configured receivers. Default value is provided by the parent connection's defaultReceiverSettings property (which is usually a reference to the JmsReceiverSettings bean instance defined in the jms-global-spring.xml file).

jms:sender

This bean defines a single-threaded context for sending events to a JMS destination, and results in the creation of a correlator output channel called `jms:senderId`. It has no required properties.

Example:

```
<jms:sender id="mySender">
    <property name="senderReliability" value="EXACTLY_ONCE"/>
    <property name="senderSettings" ref="globalSenderSettings"/>
</jms:sender>
```

Supported properties:

- `senderReliability` - the Apama reliability mode to use when events are emitted to this sender; valid values are `BEST_EFFORT`, `AT_LEAST_ONCE`, `EXACTLY_ONCE`. Default value is provided by the parent `jms:connection`'s `defaultSenderReliability` setting.
- `senderSettings` - a reference to a `JmsSenderSettings` bean, which provides access to advanced settings that are usually shared across all configured senders. Default value is provided by the parent connection's `defaultSenderSettings` property (which is usually a reference to the `JmsSenderSettings` bean instance defined in the `jms-global-spring.xml` file).

ReliableReceiveDatabase

This bean defines a database used by Apama to implement reliable receiving. It has no required properties. Typically all connections in a Correlator will share the same receive database; if the correlator is not started with the `-P` (persistence enabled) flag, this bean will be ignored.

Example:

```
<bean id="myReliableReceiveDatabase"
      class="com.apama.correlator.jms.config.DefaultReliableReceiveDatabase">
    <property name="storePath" value="jms/my-receive.db"/>
    <!-- either absolute path, or path relative to correlator store location -->
</bean>
```

Supported property:

- `storePath` - the path where the message store database should be created. Default value is `jms-receive-persistence.db`. Use an absolute path, or a path relative to the store location specified for use by the correlator state persistence store on the correlator command line.

JmsSenderSettings

This bean defines advanced settings for message senders. It has no required properties. Typically all receivers in all connections will share the same `JmsSenderSettings` bean, but it is also possible to use different settings for individual senders.

Example:

```
<bean id="globalSenderSettings"
      class="com.apama.correlator.jms.config.JmsSenderSettings">
    <property name="logJmsMessages" value="false"/>
    <property name="logJmsMessageBodies" value="false"/>
    <property name="logProductMessages" value="false"/>
</bean>
```

Supported properties

- `logJmsMessages` - if true, log information about all JMS messages that are received (but not the entire body) at `INFO` level. Default value is "false".

- `logJmsMessageBodies` - if true, log information about all JMS messages that are received, including the entire message body at `INFO` level. Default value is "false".
- `logProductMessages` - if true, log information about all Apama events that are received at `INFO` level. Default value is "false".
- `logDetailedStatus` - Enables logging of a dedicated `INFO` status line for each sender and a summary line for each parent connection. The default value is "false" (detailed logging is disabled), which results in a single summary line covering all senders and connections.
- `logPerformanceBreakdown` - Enables periodic logging of a detailed breakdown of how much time is being taken by the different stages of mapping, sending, and disk operations for each sender. By default, the messages are logged every minute at the `INFO` level. The interval can be changed if desired. The default is false, and Apama recommends disabling this setting in production environments to prevent the gathering of the performance information from reducing performance.
- `logPerformanceBreakdownIntervalSecs` - Specifies the interval in seconds over which performance throughput and timings information will be gathered and logged. Default is 60.
- `sessionRetryIntervalMillis` - Specifies how long to wait between attempts to create a valid JMS session and producer for this sender either after a serious error while using the previous session or after a previous failed attempt to create the session. However, if the underlying JMS connection has failed the `connectionRetryIntervalMillis` is used instead. Default value is 1000 ms

JmsReceiverSettings

This bean defines advanced settings for message senders. it has no required properties. Typically all receivers in all connections will share the same JmsSenderSettings bean, but it is also possible to use different settings for individual senders.

Example:

```
<bean id="globalReceiverSettings"
      class="com.apama.correlator.jms.config.JmsReceiverSettings">
    <property name="dupDetectionPerSourceExpiryWindowSize" value="2000"/>
    <property name="dupDetectionExpiryTimeSecs" value="120"/>
    <property name="logJmsMessages" value="false"/>
    <property name="logJmsMessageBodies" value="false"/>
    <property name="logProductMessages" value="false"/>
</bean>
```

Supported properties:

- `logJmsMessages` - if true, log information about all JMS messages that are sent (but not the entire body) at `INFO` level. Default value is "false".
- `logJmsMessageBodies` - if true, log information about all JMS messages that are sent, including the entire message body at `INFO` level. Default value is "false".
- `logProductMessages` - if true, log information about all Apama events that are sent at `INFO` level. Default value is "false".
- `logDetailedStatus` - Enables logging of a dedicated `INFO` status line for each receiver and a summary line for each parent connection. The default value is "false" (detailed logging is disabled), which results in a single summary line covering all receivers and connections.
- `logPerformanceBreakdown` - Enables periodic logging of a detailed breakdown of how much time is being taken by the different stages of mapping, receiving, and disk operations for each

receiver. By default, the messages are logged every minute at the INFO level. The interval can be changed if desired. The default is false, and Apama recommends disabling this setting in production environments to prevent the gathering of the performance information from reducing performance.

- logPerformanceBreakdownIntervalSecs - Specifies the interval in seconds over which performance throughput and timings information will be gathered and logged. Default is 60.
- dupDetectionPerSourceExpiryWindowSize - used for EXACTLY_ONCE receiving, and specifies the number of messages that will be kept in each duplicate detection domain per messageSourceId (if messageSourceId is set on each message by the upstream system - messages without a messageSourceId will all be grouped together into one window for the entire dupDetectionDomainId). Default value is "2000". It can be set to 0 to disable the fixed-size per-sender expiry window.
- dupDetectionExpiryTimeSecs - used for EXACTLY_ONCE receiving, and specifies the time for which uniqueMessageIds will be remembered before they expire. Default value is "120". It can be set to 0 to disable the time-based expiry window.
- maxExtraMappingThreads - Specifies the number of additional (non-receiver) threads to use for mapping received JMS messages to Apama events. The default value is 0. Using a value of 1 means all mapping is performed on a separate thread to the thread receiving messages from the bus; a value greater than 1 provides additional mapping parallelism. This setting cannot be used if maxBatchSize has been set to 1. Using multiple separate threads for mapping may improve performance in situations where mapping of an individual message is a heavyweight operation (for example, for complex XML messages) and where adding separate receivers is not desired (because they involve the overhead of additional JMS sessions and reduced ordering guarantees). Note that strictly speaking JMS providers do not have to support multi-threaded construction of JMS messages (since all JMS objects associated with a receiver's Session are meant to be dedicated to a single thread), so although in practice it is likely to be safe, it is important to verify that this setting does not trigger any unexpected errors in the JMS provider being used.

  The order in which mapped events are added to the correlator input queue (of each public context) is not changed by the use of extra mapping threads, as messages from all mapping threads on a given receiver are put back into the original receive order at the end of processing each receive batch.
- sessionRetryIntervalMillis - Specifies how long to wait between attempts to create a valid JMS session and consumer for this receiver either after a serious error while using the previous session, or after a previous failed attempt to create the session. However, if the underlying JMS connection has failed the connectionRetryIntervalMillis is used instead). Default value is 1000 ms.
- receiverFlowControl - Specifies whether application-controlled flow is enabled for each receiver. When set to true application-controlled flow control is enabled for each receiver, by listening for the com.apama.correlator.jms.JMSReceiverFlowControlMarker event and responding by calling the updateFlowControlWindow() action as appropriate. Default value is false.

JMS configuration reference

Advanced configuration bean properties

The following properties are *advanced* tuning parameters, for use only when really necessary to improve performance or work around a JMS provider bug. Since these are advanced properties, it is possible that the default values may change in any future release or that new tuning parameters may be added that could alter the semantics of the existing ones, so please be sure to carefully check the *What's New in Apama* document when upgrading, if you use any of these properties.

JMSSenderSettings

- `maxBatchSize` - The maximum (and target) number of events to be batched together for sending inside a JMS local (non-XA) transaction (which improves performance on many JMS providers). The `maxBatchSize` indicates the target number of events that will normally be sent in a single batch unless the `maxBatchIntervalMillis` timeout expires first. The `maxBatchSize` must be greater than `0` and the special value of `1` is used to indicate that a non-transacted JMS session should be used instead. Note that the same batching algorithm and parameters are used for both reliable and non-reliable senders. The default value in this release is 500.
- `maxBatchIntervalMillis` - The maximum time a sender will wait for more events to be emitted (and for reliable senders, also included in a correlator persist cycle) before timing out and sending the events ready to be sent in the batch, even if the batch size is less than `maxBatchSize`. The default value in this release is 500 ms.

JMSReceiverSettings

- `receiveTimeoutMillis` - The timeout that will be passed to the JMS provider's `MessageConsumer.receive(timeout)` method call to indicate the maximum time it should block for when receiving the next message before returning control to the Apama JMS runtime. The default value in this release is 300 ms. Some providers may require this timeout to be increased to ensure that messages can be successfully received in high-latency network conditions, although well-behaved providers should always work correctly with the default value. Reducing this timeout may improve receive latency (due to reduced time waiting for the batch to complete) on some providers; although note that many JMS providers do not strictly obey the timeout specified here so the real time spent blocking while no messages are available may be significantly higher.
- `maxBatchSize` - The maximum (and target) number of JMS messages to be received before the batch is committed to the receive-side database (if receiver is reliable) and then added to the input queues of public contexts and acknowledged to the JMS broker (whether reliable or not). The `maxBatchSize` indicates the target number of messages that will normally be received in a single batch unless the `maxBatchIntervalMillis` timeout expires first. The `maxBatchSize` must be greater than `0` and for `BEST_EFFORT` (non-reliable) receivers the special value of `1` is used to indicate that an `AUTO_ACKNOWLEDGE` session will be used instead of the default `CLIENT_ACKNOWLEDGE` session (though for reliable receivers `CLIENT_ACKNOWLEDGE` is always used even if `maxBatchSize` is `1`). The default value in this release is 1000.
- `maxBatchIntervalMillis` - the maximum time a receiver will attempt to wait for more messages to be received (and mapped) before timing out and processing the messages already received as a single batch, even if the size of that batch is less than `maxBatchSize`. The default value in this release is 500 ms. Note that in practice, when no messages are available, many JMS providers seem to block for longer than the specified `receiveTimeoutMillis` before returning, which may lead to the true maximum batch interval being significantly longer than the value specified here.

XML configuration bean reference

Designing and implementing applications for JMS

This section describes guidelines for designing and implementing applications that make use of correlator-integrated JMS messaging.

Correlator-Integrated JMS Messaging

Using correlator persistence with Correlator-integrated JMS

Correlator-integrated JMS can be used with or without the correlator's state persistence feature. In a persistent correlator, all reliability modes can be used (both reliable and unreliable messaging), but in a non-persistent correlator only `BEST_EFFORT` (unreliable) messaging is supported, and attempts to add senders or receivers using any other reliability will result in an error.

In a persistent correlator, information about all senders and receivers is always stored in the persistent data store - unreliable ones as well as reliable, statically defined as well as dynamic. This means that persistent Apama applications never need to worry about re-creating previously-added JMS senders and receivers after recovery, as this will happen automatically - even for `BEST_EFFORT` (unreliable) senders and receivers; if also means that for reliable sender and receivers no messages or duplicate detection information will be lost after a crash or restart.

Note that because sender and receiver information is stored in the database, it is not permitted to start a persistent correlator, shut it down, then make changes such as removing existing static senders and receivers in the configuration file before restarting. If the ability to remove senders and receivers is required, they must be added dynamically using EPL rather than from the configuration file. However, you can add new senders and receivers to the configuration files between restarts, provided the identifiers do not clash with any previously defined static or dynamic sender or receiver.

It is not possible to change the configuration of dynamic senders or receivers once created under any circumstances. For static senders and receivers this is also mostly prohibited, with the exception that the destination of a static receiver defined explicitly in the configuration file can be changed between restarts of the Correlator (provided the receiverId and dupDetectionDomainId remain the same). However to retain maximum flexibility, Apama recommends that customers follow the industry standard practice of using JNDI names for queues and topics so that it is always possible to configure any necessary redirections to allow the same logical (JNDI) name to be used in different deployment environments, such as production and deployment (for dynamic as well as static receiver).

As there is no restriction on changing the connection factory or JNDI server details between restarts of a persistent correlator, by using the same JNDI names (or if necessary, queue and topic names) in all environments, but different isolated JMS and JNDI servers for production and testing, it is possible to avoid unintended interactions between the production and test environments while also keeping the two configurations very similar and allowing production data stores to be examined in the test environment if necessary.

Designing and implementing applications for JMS

How reliable JMS sending integrates with correlator persistence

This topic describes the details of how JMS sending integrates with correlator persistence and is intended for advanced users.

When sending JMS messages in a persistent correlator, all events emitted to a JMS sender are queued inside the correlator until the next persist cycle begins. The events cannot be passed to JMS until the EPL application state changes that caused them to be sent have been persisted, otherwise the downstream receiver might see an inconsistent set of events in the case of a failure and recovery.

Note that to avoid potentially inconsistent output in the event of a failure, this is true not only for reliable AT_LEAST_ONCE and EXACTLY_ONCE messages but also for BEST_EFFORT messages in a persistent correlator. The only differences between BEST_EFFORT and the reliable sending modes (other than whether the messages use the JMS PERSISTENT delivery mode flag), is that for the reliable modes the messages are guaranteed to remain on disk until they have been successfully sent to the JMS broker, whereas for BEST_EFFORT this is not the case.

Unique identifiers are generated and assigned to each message when they are emitted, and persisted with the events to allow downstream receivers to perform EXACTLY_ONCE duplicate detection if desired (note, this assumes the uniqueMessageId is mapped into the JMS message in some fashion).

Once the next persist cycle has completed and both the events and the application state that caused them has been committed to disk, the events can be sent to JMS. After messages have been successfully sent to the JMS broker they are lazily removed from the correlator's in-memory and on-disk data structures. The latency of sent messages is therefore dependent on the time taken for the correlator to perform a persist cycle (including the persist interval, the time taken to snapshot the correlator's state and commit it to disk, and any retries if the state cannot be snapshotted immediately), plus any time spent waiting to fill the batch of events to be sent (although this is usually relatively small). Note that if a message send fails and it is not due to the JMS connection being lost then after a small number of retries it will be dropped with an ERROR message in the log. If a send fails because the connection is down, the correlator simply waits for it to come up again in all cases.

Using correlator persistence with Correlator-integrated JMS

How reliable JMS receiving integrates with correlator persistence

This topic describes the details of how JMS receiving integrates with correlator persistence and is intended for advanced users.

When receiving in AT_LEAST_ONCE or EXACTLY_ONCE mode, messages are taken from the JMS queue or topic in batches (using JMS CLIENT_ACKNOWLEDGE mode). The resulting Apama events are persisted in the reliable receive JMS datastore (which is separate from the correlator's recovery datastore) and then acknowledged back to JMS before the next batch of messages is received. After a batch of events finishes being asynchronously committed to the data store, it is added to the input queue of each context. When the correlator next completes a persist cycle, all events that had at least been added to the input queue by the beginning of the persist cycle have been (or will be) reliably passed to the application. This means that in AT_LEAST_ONCE mode they can be removed from the receive data store immediately.

If EXACTLY_ONCE is being used and the event was mapped with a non-empty uniqueMessageId from the JMS message, the uniqueMessageId and other meta-data are stored both in memory and in the on-disk reliable data store, and are kept there until the associated uniqueMessageId is expired from the duplicate detector. Note however, that as an optimization, because the persisted event strings are no longer needed once the event has been included in the correlator state database, any particularly long event strings may be nulled out in the database. The latency of received messages is therefore dependent on the time spent waiting for other messages to be received to fill the batch, and the time taken to commit the batch to the receive data store.

When a persistent correlator is restarted and recovers its state from the recovery data store, no new JMS messages will be received from the broker until recovery is complete, specifically until the correlator calls the onConcludeRecovery() action on all EPL monitors that have defined one. It is possible that EPL monitors will see a small number of JMS messages that were received and added to the input queue before the correlator was restarted, so to be safe any required listeners in non-persistent monitors should be set up in `onBeginRecovery()`.

Since a batch of messages is acknowledged to the JMS broker as soon as they have been written to the Apama reliable receive JMS datastore, there is no relationship between JMS message acknowledgement to the broker and when the correlator begins or completes a correlator state datastore persistence cycle. The maximum number of messages that may be received from the JMS broker but not yet acknowledged is limited by the configured `maxBatchSize` (typically this is 1000 messages).

Using correlator persistence with Correlator-integrated JMS

Using the correlator replay log with Correlator-integrated JMS

The correlator replay log can be used in applications that use most Correlator-integrated JMS features including sending, receiving and listening for status events. The replay log will include a record of all events that were received from JMS so there is no need for JMS to be explicitly enabled with the `--jmsConfig` option when performing replay. Instead, the resulting replay log can be extracted and used in the normal way, without the `--jmsConfig` option. Attempting to perform replay with JMS enabled is not supported and is likely to fail, especially with reliable receivers in a persistent correlator.

Note that the "dynamic" capabilities of Correlator-integrated JMS do not currently work in a replay correlator (because a correlator plug-in is used behind the scenes), so if you need to retain the possibility of using a replay log you must not use dynamic senders and receivers or call the `JMSSender.getOutstandingEvents()` method.

Designing and implementing applications for JMS

Reliability considerations

When using the `EXACTLY_ONCE` or `AT_LEAST_ONCE` mode, Apama's correlator-integrated JMS messaging provides a "reliable" way to send messages into and out of the correlator such that in the event of a failure, any received messages whose effects were not persisted to stable storage will be redelivered and processed again, and that the events received from the correlator by external systems are consistent with the persisted and recovered state.

- Apama's JMS messaging guarantees no message loss, assuming there is stable storage and that the JMS broker behaves reliably. Also, there must be no fatal message mapping errors.
- Apama's JMS messaging guarantees no message duplication within a specifically configured window size. The window size, for example, might be set to the last 2000 events or events received in the last two minutes. Note that even with the help of Apama's `EXACTLY_ONCE` functionality, JMS message duplicate detection is not a simple or automatic process and requires careful design. Customers are strongly encouraged to architect their applications to be tolerant of duplicate messages and use the simpler `AT_LEAST_ONCE` reliably mode instead of EXACTLY_ONCE when possible.
- Apama's JMS messaging provides a *best effort* correct message ordering but this is not guaranteed. The exact message ordering behavior is broker-specific. In accordance with the JMS specification, the Apama JMS messaging does not make ordering guarantees in the event of a broker or client failure. Occasionally, some JMS brokers reorder messages unexpectedly. If your application requires correct message order, it may be possible to set the `JMSXGroupSeq` and `JMSXGroupID` message properties to request the chosen JMS provider implementation to

provide ordering for a group of related messages. It is not possible to provide ordering across all messages without forcing use of a single consumer, which would reduce throughput scalability.

Care must be taken when designing, configuring and testing the application to ensure it can cope with significant fluctuations in message rates, as well as serious failures such as network or component failures that lasts for several minutes, hours or days. Consider using JMS message expiry to avoid flooding queues with unnecessary or stale messages on recovery after a long period of down time.

Designing and implementing applications for JMS

Duplicate detection

Apama provides an `EXACTLY_ONCE` receiver reliability setting that allows a finite number of duplicate messages to be detected and dropped before they get to the correlator. This setting can be used to reduce the chance of duplicates; however with JMS, duplicate detection is a complex process. Therefore, customers are strongly encouraged to architect their applications to be tolerant of duplicate messages and use the simpler `AT_LEAST_ONCE` reliability mode instead of `EXACTLY_ONCE` when possible.

Configuring duplicate detection is an inexact science given that it depends considerably on the behavior of the sender(s) for a queue, and requires careful architecture and sizing to ensure robust operation in normal use and expected error cases. Moreover it is not possible to guarantee duplicate messages will never be seen without an infinite buffer of duplicates. Give particular attention to architectures where multiple sender processes are writing to the same queue, especially if it is possible that one sender may send a duplicate message it has taken off another failed sender that has not recorded the fact that it is already processed and sent out a given message.

Duplicate detection is a trade-off between probability of an old duplicate not being recognized as such, and the amount of memory and disk required, which will also have an impact on latency and throughput.

Selecting the right value for the `dupDetectionExpiryTimeSecs` is a very important aspect of ensuring that the duplicate detection process will operate reliably — detecting duplicates where necessary without running out of memory when something goes wrong. The expiry time used for the duplicate detector should take into account how the JMS provider will deal with several consecutive process or connection failures on the receive side, especially if the JMS provider temporarily holds back messages for failed connections in an attempt to work around temporary network problems. Be sure to consult the documentation for the JMS provider being used to understand how it handles connection failures. It is a good idea to conduct tests to see what happens when the connection between the JMS broker and Correlator-JMS client goes down. When testing, consider using the "`rMaxDeliverySecs=`" value from the "`JMS Status:`" line in the correlator log to help understand the minimum expiry time needed to catch redelivered duplicates. Note, however, this is only useful if the JMS provider reliably sets the `JMSRedelivered` flag when performing a redelivery. A good rule of thumb is to use an expiry time of two to three times the broker's redelivery timeout.

Note that although space within the reliable receive (duplicate detection) data store is reclaimed and reused when older duplicates expire, the file size will not be reduced. There is currently no mechanism for reducing the amount of disk space used by the database, so the on-disk size may grow, bounded by the peak duplicate detector size, but will not shrink.

Messages that are subject to duplicate detection contain:

- uniqueMessageId - an application-level identifier which functions as the key for determining whether a message is functionally equivalent (or identical) to a message already processed, and should therefore be ignored.
- messageSourceId - an optional application-specific string which acts as a key to uniquely identify upstream message senders. This could be a standard GUID (globally unique identifier) string. If provided, the messageSourceId is used to control the expiry of uniqueMessageIds from the duplicate detection cache, allowing dupDetectionPerSourceExpiryWindowSize messages to be kept per messageSourceId. This massively improves the reliability of the duplicate detection while keeping the window size relatively small, since if one sender fails then recovers several hours later, there is no danger of another (non-failed) sender filling up the duplicate detection cache in the meantime and expiring the ids of the first sender causing its duplicates to go undetected.

The key configuration options for duplicate detection are:

- dupDetectionPerSourceExpiryWindowSize - The number of messages that will be kept in each duplicate detection domain per messageSourceId (if messageSourceId is set on each message by the upstream system - messages without a messageSourceId will all be grouped together into one window for the entire dupDetectionDomainId). This property is specified on the global JmsReceiverSettings bean. It is usually configured based on the characteristics of the upstream JMS sender, and the maximum number of in-doubt messages that it might resend in the case of a failure. The default value in this release is 2000. It can be set to 0 to disable the fixed-size per-sender expiry window.
- dupDetectionExpiryTimeSecs - The time for which uniqueMessageIds will be remembered before they expire. This property is specified on the global JmsReceiverSettings bean. The default value in this release is 2 minutes. It can be set to 0 to disable the time-based expiry window (which makes it easier to have a fixed bound on the database size, though this is not an option if the JMS provider itself causes duplicates by redelivering messages after a timeout due to network problems).
- dupDetectionDomainId - An application-specific string which acts as a key to group together receivers that form a duplication detection domain, for example, a set of receivers that must be able to drop duplicate messages with the same uniqueMessageId (which may be from one, or multiple upstream senders). This property is specified on the jms:receiver bean. By default, the duplicate detection domain is always the same as the JMS destination name and connectionId, so cross-receiver duplicate detection would happen only if multiple receivers in the same connection are concurrently listening to the same queue; duplicates would not be detected if sent to a different queue name, or if sent to the same queue name on a different connection, or if JNDI is used to configured the receiver but the underlying JMS name referenced by the JNDI name changes. Also note that if the message streams processed by each receiver were being partitioned using message selector, unnecessary duplicate detection would be performed in this case. The duplicate detection domain name can be specified on a per-receiver basis to increase, reduce or change the set of receivers across which duplicate detection will be performed. Common values are:
  - dupDetectionDomainId=connectionId+":"+jmsDestinationName - the default for queues.
  - dupDetectionDomainId=jmsDestinationName - if using receivers to access the same queue from multiple separate connections.
  - dupDetectionDomainId=jndiDestinationName - if using JNDI to configure receiver names, and needing the ability to change the queue or topic that the JNDI name points to.

- `dupDetectionDomainId=connectionId+":"+receiverId` - the default for topics; also used if each receiver should check for duplicates independently of other receivers. This is useful if receivers are already using message selectors to partition the message stream, which implies that cross-receiver duplicates are not possible.
- `dupDetectionDomainId=<application-defined-name>` - if using multiple receivers per selector-partitioned message stream. The name is likely to be related to the message selector expression.

Duplicate detection only works if the upstream JMS sender has specified a `uniqueMessageId` for each message (the `uniqueMessageId` is typically as a message property, but could alternatively be embedded within the message body if the mapper is configured to extract it). Any messages that do not have this identifier will not be subject to duplicate detection. The `uniqueMessageId` string is expected to be unique across all messages within the configured `dupDetectionDomainId` (for example, `queue`), including messages with different `messageSourceId`s. By default, sent JMS messages would have a `uniqueMessageId` of `seqNo:messageSourceId`, where *`seqNo`* is a contiguous sequence number that is unique for the sender, for example:

```
uniqueMessageId=1:mymachinename1.domain:1234:567890:S01
uniqueMessageId=2:mymachinename1.domain:1234:567890:S01
uniqueMessageId=3:mymachinename1.domain:1234:567890:S01
uniqueMessageId=1:mymachinename2.domain:4321:987654:S01
uniqueMessageId=2:mymachinename2.domain:4321:987654:S01
uniqueMessageId=1:mymachinename2.domain:4321:987654:S02
uniqueMessageId=2:mymachinename2.domain:4321:987654:S02
...
```

To reliably perform duplicate detection if there are multiple senders writing to the same queue (without the Apama receiver having to configure a very large and therefore costly time window to prevent premature expiry of ids from a sender that has failed and produces no messages for a while then recovers, possibly sending duplicates as it does so), the upstream senders should be configured to send with a globally-unique `messageSourceId` identifying the message source/sender, which should also be configured in the mapping layer of the receiver.

Apama's duplicate detection involves a set of fixed-size per-sourceId queues, and when the queue is full the oldest items are expired to a shared queue ordered by timestamp (time received by the correlator JMS runtime) whose items are expired based on a time window. So the receiver settings controlling duplicate detection window sizes are:

- `dupDetectionPerSourceExpiryWindowSize`
- `dupDetectionExpiryTimeSecs`

`uniqueMessageId`s are expired from the per-source queue (and moved to the time-based queue) when it is full of newer ids, or when a newer message with the same `uniqueMessageId` already in the queue for that source is received.

`uniqueMessageId`s are expired from the time-based queue (and removed from the database permanently) when they are older than the newest item in the time-based queue by more than `dupDetectionExpiryTimeSecs`.

Reliability considerations

Performance considerations

When designing an application that uses correlator-integrated JMS messaging it may be relevant to consider the following topics that relate to performance issues.

There are no guarantees about maximum latency. Persistent JMS messages inevitably incur significant latency compared to unreliable messaging, and Apama's JMS support is focused around throughput rather than latency. Messages can be held up unexpectedly by many factors such as: the JMS provider; by connection failures; by waiting a long time for the receive-side commit transaction; by the broker `acknowledge()` call taking a long time; or by waiting a long time for the correlator to do an in-memory copy of its state.

Multiple receivers on the same queue may improve performance. But consider that "For PTP, JMS does not specify the semantics of concurrent `QueueReceivers` for the same `Queue`; however JMS does not prohibit a provider from supporting this. Therefore, message delivery to multiple `QueueReceivers` will depend on the JMS provider's implementation. Applications that depend on delivery to multiple `QueueReceivers` are not portable".

- If performance is an issue, be sure to check the correlator log for `WARN` and `ERROR` messages, which may indicate your application or configuration has a connection problem that may be responsible for the performance problem.
- Ensure that the correlator is not running with `DEBUG` logging enabled or is logging all messages. Either of these will obviously cause a big performance hit. Apama recommends running the correlator at `INFO` log level; this avoids excessive logging, but still retains sufficient information that may be indispensable for tracking problems.
- In practice, most performance problems are caused by mapping, especially when XML is used. Whenever possible, Apama recommends avoiding the use of XML in JMS messages due to the considerable overhead that is always added by using such a complex message format. For example, use `MapMessage` or a `TextMessage` containing an Apama event string.
- If you are receiving several different event types, ensure that the conditional expressions used to select which mapping to execute are as simple as possible. In particular, there will be a significant performance improvement if JMS message properties are used to distinguish between different message types instead of XML content inside the message body itself because JMS message properties were designed in part for this purpose.
- Use the Correlator Status lines in the log file to check whether the bottleneck is the JMS runtime or in the EPL application itself. A full input queue ("`iq=`") is a strong indicator that the application may not be consuming messages fast enough from JMS.
- Consider enabling the `logPerformanceBreakdown` setting in `JmsSenderSettings` and `JmsReceiverSettings` to provide detailed low-level information about which aspects of sending and receiving are the most costly. This may indicate whether the main bottleneck, and hence the main optimization target, is in the message mapping or in the actual sending or receiving of messages. If mapping is not the main problem, it may be possible to achieve an improvement by customizing some of the advanced sender and receiver properties such as `maxBatchSize` and `maxBatchIntervalMillis`.
- Consider using `maxExtraMappingThreads` to perform the mapping of received JMS messages on one or more separate threads. This is especially useful when dealing with large or complex XML messages.
- Take careful measurements. The key to successful performance optimization is taking and accurately recording good measurements, along with the precise configuration changes that were made between each measurement. It is also a good idea to take multiple measurements over a period of at least several minutes (at least), and take account of the amount of variation or error in the measurements (by recording minimum, mean, and maximum or calculating the standard deviation). In this way it is possible to notice configuration changes that have made a

real and significant impact on the performance, and distinguish them from random variation in the results. Note that many JMS providers are observed to behave badly and exhibit poor performance when overloaded (for example, when sending so fast that queues inside the broker fill up and things begin to block). For this reason, the best way to test maximum steady-state performance is usually to create a way for the process that sends messages to be notified by the receiving process about how far behind it is. For example, if the sender and receiver are both correlators, `engine_connect` can be used to create a fast channel from the receiver back to the sender, and the test system set to emit Apama events to the sender every 0.5 seconds so it knows how many events have been received so far. This allows better performance testing with a bound on the maximum number of outstanding messages (sent but not yet received) to prevent the broker being overwhelmed.

- Be careful when measuring performance using a virtual machine rather than dedicated hardware. VMs often have quite different performance characteristics to physical hardware. Take particular care when using VMs running on a shared host, which may be impacted by spikes in the disk/memory/CPU/network of other unrelated VMs running on the same host that belong to different users.

Designing and implementing applications for JMS

JMS performance logging

The `JmsSenderSettings` and `JmsReceiverSettings` configuration objects both contain a property called `logPerformanceBreakdown` which can be set to true to enable measurement of the time taken to perform the various operations required for sending and receiving, with messages logged periodically at `INFO` level with a summary of measurements taken since the last log message. The default logging interval is once per minute.

Although this property should not be enabled in a production system where performance is a priority because the gathering of the performance data adds unnecessary overhead, it can be indispensable during development and testing for demonstrating what each sender and receiver thread is spending its time doing. To produce more useful statistics, note that the first batch of messages sent or received after connection may be ignored (which will affect all statistics logged, including the number of messages received and throughput). All times are measured using the standard Java `System.nanoTime()` method, which should provide the most accurate time measurements the operating system can achieve, though not usually to nano second accuracy. For more information on the `logPerformanceBreakdown` property, see "XML configuration bean reference" on page 304.

Performance considerations

Sender performance breakdown

Each sender performance log message has a low-level breakdown of the percentage of thread time spent on various aspects of processing each message and message batch, as well as a summary line stating the approximate throughput rate over the previous measurement interval, and an indication of the minimum, mean (average) and maximum number of events in each batch that was sent.

The items that may appear in the detailed breakdown are:

- `MAPPING` - time spent mapping each Apama event to the corresponding JMS message. This includes the time spent looking up any JMS queue, topic, or jndi destination names, unless cached.
- `SENDING` - time spent in the JMS provider's `MessageProducer.send()` method call for each message.

- JMS_COMMIT - time spent in the JMS provider's Session.commit() method call for each batch of sent messages (only if a JMS TRANSACTED_SESSION is being used to speed up send throughput).
- WAITING - the total time spent waiting for the first Apama event to be passed from EPL to the JMS runtime for sending, per batch. This is affected by what the EPL code is doing, and for reliable sender modes, also by the (dynamically tuned) period of successful correlator persist cycles.
- BATCHING - the total time spent waiting for enough Apama events to fill each send batch, after the first event has been passed to the JMS runtime.
- TOTAL - aggregates the total time taken to process each batch of sent messages.

JMS performance logging

Receiver performance breakdown

Each receiver performance log message has a low-level breakdown of the percentage of thread time spent on various aspects of processing each message and message batch, as well as a summary line stating the (approximate) throughput rate over the previous measurement interval, and an indication of the minimum, mean (average) and maximum number of events in each batch that was received.

The items that may appear in the detailed breakdown are:

- RECEIVING - time spent in the JMS provider's MessageConsumer.receive() method call for each message received.
- MAPPING - time spent mapping each JMS message to the corresponding Apama event. If maxExtraMappingThreads is set to a non-zero value then this is the time spent waiting for remaining message mapping jobs to complete on their background thread(s) at the end of each batch.
- DB_WAIT - (only for reliable receive modes) time spent waiting for background reliable receive database operations (writes, deletes, etc) to complete, per batch.
- DB_COMMIT - (only for reliable receive modes)time spent committing (synching) received messages to disk at the end of each batch.
- ENQUEUING - (only for BEST_EFFORT receive mode) time spent adding received messages to each public context't input queue.
- JMS_ACK - time spent in the JMS provider's Message.acknowledge() method call at the end of processing each batch of messages.
- R_TIMEOUTS - the total time spent waiting for JMS provider to complete MessageConsumer.receive() method calls that timed out without returning a message from the queue or topic, per batch. Indicates either that Apama is receiving messages faster than they are added to the queue or topic or that the JMS provider is not executing the receive (timeout) call very efficiently or failing to return control at the end of the requested timeout period.
- FLOW_CONTROL - the total time spent (before each batch) blocking until the EPL application increases the flow control window size by calling JMSReceiverFlowControlMarker.updateFlowControlWindow(...). In normal usage, this should be negligible unless some part of the system has failed or the application is not updating the flow control window correctly.
- TOTAL - aggregates the total time taken to process each batch of received messages.

JMS performance logging

Configuring Java options and system properties

Sometimes it is necessary to specify Java system properties to configure a JMS provider's client library, or to change JVM options such as the maximum memory heap size. Because these settings inevitably affect all JMS providers that the correlator is connecting to, in addition to any Apama Java (JMON) applications in the correlator, Java options must be specified on the correlator command line rather than in a JMS connection's configuration file.

Each Java option to be passed to the correlator should be prefixed with `-J` on the command line, for example, `-J-Dpropname=propvalue -J-Xmx512m`. To set Java options when starting the Correlator from Apama Studio, edit the Apama launch configuration for your project as follows:

1. In the Project Explorer right-click the project name and select Run As > Run Configurations. The **Run Configurations** dialog is displayed.
2. In **Run Configurations** dialog, in the Project field, make sure the your project is selected.
3. On **Run Configurations** dialog's Components tab, select the correlator to use and click Edit. The **Correlator Configuration** dialog is displayed.
4. In the **Correlator Configuration** dialog, in the Extra command line arguments field, add the system property, for example, `-J-Dpropname=propvalue -J-Xmx512m` and click OK.

Designing and implementing applications for JMS

Diagnosing problems when using JMS

This topic contains several approaches for diagnosing JMS issues you may encounter.

- Consider contacting the vendor of the JMS provider that is being used. JMS brokers are complex pieces of software with many configuration options. JMS providers often maintain on-line databases of known bugs and issues. Software AG is not in a position to provide detailed support or performance tuning for JMS brokers provided by a third party, but the provider may be able to suggest useful changes to configuration options that can affect performance and reliability trade-offs and provide further assistance tracking down crashes, hangs, performance, disconnection and flow control problems.
- Check the correlator log file for `WARN` and `ERROR` messages that may indicate the underlying problem. Also check for any log lines "`Longest delay between a JMS message being sent and the broker delivering it to this receiver is now`", especially after an unexpected disconnection or when testing a correlator/broker machine or network failure. This will give an indication of how your broker redelivers in-doubt messages, and may affect the size of the duplicate detection time-based expiry window.
- Check the JMS broker's log files and console for error messages or warnings.
- Consider temporarily using `logJMSMessages` and `logProductMessages` to display all messages being sent and received. This is particularly useful for problems related to mapping; on the other hand it is not useful for diagnosing performance-related issues.
- Use the "`JMS Status:`" lines to understand what is going on in more detail. Consider setting `logDetailedStatus=true` to get more in-depth per-sender and per-receiver status lines.

- Check for any log lines "`Longest delay between a JMS message being sent and the broker delivering it to this receiver is now` " ... which may indicate that the broker is behaving strangely or that queued messages from a previous test run are unexpectedly being received, perhaps causing mapping failures or performance problems.
- If further assistance from Software AG is required to track down a problem, it is essential to provide a copy of the full correlator log file and the JMS configuration being used to ensure that all the required information is available.
  - To capture the correlator log output, edit the launch configuration as follows:
    1. Right-click the project and select Run As > Run Configurations from the pop-up menu.
    2. Ensure the configuration for this project is selected.
    3. Select the Components tab,
    4. Edit the Default Correlator setting by adding extra command line arguments: `--logfile logs/correlator.log`.
    5. Optionally add `--truncate` to clear the log file at start up to eliminate confusion with output from previous runs.

    Note, simply copying lines from the Console view is usually *not* adequate for support purposes (for example, status lines are missing and in some cases header information is missing as well).
  - To collect the essential JMS configuration files.
    1. Right-click the project and select Properties from the pop-up menu.
    2. In the Resource section, note the directory information listed in the Location field. (Copy the information if desired.)
    3. In the file system, navigate to that directory. (Paste the directory information into the Run command of the Windows Start menu.)
    4. Zip up the contents of the `bundle_instance_files\Correlator-Integrated_JMS` sub-directory.

Correlator-Integrated JMS Messaging

JMS failures modes and how to cope with them

Apama provides many features that simplify the integration process, but JMS brokers are complex pieces of software performing a complex task and successfully architecting a truly reliable application built on JMS requires careful thought and testing, as well as a full understanding of the behavior and configuration of your chosen JMS provider.

The following list highlights some of the things that can go wrong, and how they are handled by Apama along with suggestions of how they might be handled by a solution architect.

- Failure of connection between the correlator and the JMS-broker (due to machine failure or network problems) – Apama handles this by writing an `ERROR` to the correlator log and sending `JMSConnectionStatus`, `JMSSenderStatus`, `JMSReceiverStatus` events detailing the error to all affected connections, senders, and receivers. An application can use these events to display the problem on a dashboard or send an email or text message to notify an administrator. Once the connection has gone down Apama will repeatedly try to re-establish it, at a rate determined by the `connectionRetryIntervalMillis` property of `jms:connection` (once per second by default). As soon as the connection has been re-established, all associated senders and receivers will create a session using the new connection and begin to send and receive again. Note that occasionally some third party JMS libraries have been observed to hang after a network problem, preventing successful reconnection, especially when there is a mismatch between the .jar versions used on the client and server; it is worth testing to ensure this does not affect your deployment. During the period when the connection is down Apama will be unable to send emitted events to JMS, so all such events will be queued in memory - see the "Sending messages too fast" failure mode for more details.

- Sending messages too fast (because the connection is down; because the broker's queue is exceeded due to a downstream JMS client receiving blocking; or simply because the attempted send rate is too high) – A bounded number of unsent messages will be held in a Java buffer until sent to JMS, but if the number of outstanding events exceeds that buffer they will be queued in C++ code. It is possible the correlator could fail with a C++ out of memory error in rare cases where too many events are emitted to a reliable sender between persistence cycles. However in most cases the behavior will be that the JMS runtime acts as an Apama 'slow consumer' and in time causes correlator contexts to block when calling `emit` until the messages can be processed. In time this may also cause the input queue to fill up, to prevent an out of memory error occurring. All of this behavior can be avoided if necessary by using the `JMSSender.getOutstandingEvents()` action to keep track of the number of outstanding events and take some policy-based action when this number gets too high. Typical responses might be to page some out to a database, notify an administrator, or begin to drop messages. Also note that many JMS providers have built in support for 'paging' or 'flow to disk' that, when enabled, allows messages to be buffered on disk client-side if the broker cannot yet accept them. In some cases this may be more desirable than causing the correlator to block.

- Receiving messages too fast – In a well-designed system an Apama application will usually be able to keep up with the rate of messages arriving from JMS. However it is important to consider the possibility of a large number of messages being received quickly on startup or after a period of downtime (for example, due to hardware failure), or from a backlog of input messages building up when downstream systems such as databases or JMS destinations that the application needs to use to complete processing of input messages become unresponsive.

  If messages are received too fast for the Apama application's listeners to synchronously process them, the input queue will fill up, after which the JMS receivers will be blocked from sending more messages until the backlog is cleared. However, if the listeners for the input messages complete quickly but kick off asynchronous operations for each input message (for example, event listeners for database requests, or adding the messages to EPL data structures) then it is possible that the correlator could instead run out of memory if messages continue to be received faster than they can be fully processed. Correlator-JMS provides a feature called *receiver flow control* to deal with these situations, which allows an EPL application to set a window size representing the number of events that each JMS receiver can take from the broker, thereby putting a finite bound on the number of outstanding events and operations. See "Receiver flow control" on page 290 for more information about receiver flow control. Another approach to avoid a very large warm-up period when dealing with old messages during startup is to make use of the JMS message time-to-live header when sending messages. This ensures that older messages can be deleted from the queue by the JMS broker once they are no longer useful. Some JMS providers may also have configuration options to enable throttling of message rates.

- JMS destination not found for a receiver (when the JMS connection is still up) – This could be a transient problem such as a situation where a JMS server is up but a JNDI server is down, where or a JNDI name has not yet been configured. The failure could also be a permanent one such as a destination name that is invalid. Apama handles this case by writing an ERROR log message, sending a JMSReceiverStatus event with status of "DESTINATION_NOT_FOUND" (or in non-conformant JMS implementations, possibly "ERROR"), then backing off for the configured sessionRetryIntervalMillis (1 second by default), before retrying. If it is expected that destination names may often be invalid, it might be best to use dynamic rather than static receivers. This allows the Apama application to take a policy-based decision on whether to give up trying to look up the destination and remove the receiver after a timeout period.

- JMS error sending message (when the JMS connection is still up) – This could be a transient problem such as a situation where the JMS server has a problem but the connection's exception listener not yet triggered. The failure could be permanent one such as a case where a JMS message is invalid for some reason. Apama writes an ERROR log message when this happens. If the error is specific to this message such as MessageFormatException or InvalidDestinationException then the message is simply dropped. In other error cases, Apama will back off for the configured sessionRetryIntervalMillis (1 second by default) then close and recreate the session and MessageProducer before retrying once. After two failed attempts Apama stops trying to send the message to avoid the sender getting stuck. If a number of messages are being sent in a transacted batch for performance reasons, when a failure occurs Apama retries each message in the batch one by one in their own separate transactions to ensure that problems with one message do not affect other messages.
- JMS destination not found when sending a message (when the JMS connection is still up) – This could be a transient problem such as a JMS server being up but with a JNDI server down, or a JNDI name not configured yet. It could be a permanent failure such as a destination name that is invalid. Apama handles this case in a fashion similar to the way it handles the "JMS error sending message" case mentioned above, except that it does not attempt to retry sending if it determines that a destination not found error was the cause, since it is unlikely to work a second time after an initial failure, and other messages being sent to different destinations would get held up if it did.
- Exception while a mapping message (during sending or receiving; typically caused by invalid mapping rules, invalid conditional expressions, or malformed messages, such as an unexpected XML schema) – If the mapping error is so serious that the message cannot be mapped at all (for example, receiving a message that did not map any of the defined conditional mapping expressions), an ERROR is logged and the message is dropped. If the error affects only one of the field mapping rules, then an ERROR is logged and the field will be given a default value such as "", 0, null, etc. Note that a large batch of badly formed messages can result in a large number of messages and stack traces being written to the log, so care should be taken to avoid this by comprehensive testing and careful writing of conditional expressions.
- Error parsing received event type (due to mismatch between mapping rules and injected event types, or failure to inject the required types) – The correlator logs a WARN message when events are received that do not match any injected event type; the log file should be checked during integration testing to ensure this is not happening.
- EXACTLY_ONCE duplicate detector fails to detect duplicates – Correctly detecting all duplicate messages involves ensuring that the upstream JMS client (if different from Correlator-JMS) is correctly putting truly unique identifiers into all the messages it sends, and that the receiving JMS client is configured with a sufficiently large window of duplicate identifiers to catch all likely cases in which duplicates might be sent. When configuring the receiver's duplicate detector, it is particularly important to understand the circumstances under which your JMS provider will redeliver messages — some providers will redeliver messages several minutes after they were originally sent especially in the event of a failure, which means the duplicate detector time window needs to be at least two or three times larger than the redelivery window. If messages are being put onto the bus from multiple senders, it is an extremely good idea to set

a messageSourceId on each message to allow Correlator-integrated JMS to maintain a separate duplicate detection window for each message source. In some applications it may be useful to set a time-to-live on sent messages to place a bound on the maximum delay between sending a message and having it received and successfully recognized as a duplicate, in those situations where it is better to risk dropping potentially non-duplicate older messages than to risk re-processing duplicate older messages.

- EXACTLY_ONCE duplicate detector out of memory – It is important to ensure that there is enough memory on the machine and enough allocated to the Correlator's JVM to hold the all of the duplicate detection information required for both normal usage and exceptional cases; if this memory is exceeded then the correlator process will fail with an out of memory error. Note that this only applies to reliable receivers using EXACTLY_ONCE reliability; due to the additional complexity arising from duplicate detection, customers are advised to use this feature only when really needed — in many cases it is possible to architect an application so that it is tolerant of duplicate messages (idempotent) which completely avoids the need for all design, sizing and testing work that EXACTLY_ONCE mode entails. If duplicate detection is enabled, the total amount of memory required by the duplicate detector for each dupDetectionDomainId is a function of the average message size, the number of distinct messageSourceIds (per dupDetectionDomainId), and the configuration parameters dupDetectionPerSourceExpiryWindowSize and dupDetectionExpiryTimeSecs. It is not practical to accurately estimate the exact memory requirements of the duplicate detector in advance; instead, it is recommended that applications with high reliability requirements are carefully tested to determine how much memory is required with the peak likely memory usage, and to ensure that the correlator's JVM is configured with a sufficiently high maximum memory limit to accommodate this (for example on the command line set -J-Xmx2048m for a 2GB heap). The most important parameter to watch is the dupDetectionExpiryTimeSecs, since the time-based expiry queue does not have a bounded number of items, so if it is set to be too large or a lot of messages are received unexpectedly in a very short space of time it could grow to a very large size. The "JMS Status" lines that the correlator periodically logs provide invaluable information about the number of duplicate detection ids being stored at any time, as well as the amount of memory the JVM is currently using. Enabling the logDetailedStatus receiver settings flag will turn on additional information for each receiver that includes a breakdown of the number of duplicate detection identifiers stored in each part of the duplicate detector.
- Disk errors/corruption – Both correlator persistence and the reliable receive functionality of Correlator-integrated JMS depend on the disk subsystem they are written to. It is important to use some form of storage that is reliable such as a NAS/Network-Attached Storage device or SAN/storage-area network and which is guaranteed to not introduce corruption in the event of a failure such as a power failure. Apama also relies on the file system to implement correct file locking; if this is not the case or if the device is not correctly configured, then it is possible that messages could be lost or the correlator could fail, either in normal operation or in the event of an error.
- JMS provider bugs – A number of widely used enterprise JMS providers have bugs that might result in message loss, reordering, or unexpected re-deliveries (causing duplication). In other cases some bugs manifest as broker or client-side hangs, Java deadlocks, thread and memory leaks, or other unexpected failures. These are especially common when a JMS client like the correlator has been disconnected uncleanly from the JMS broker, perhaps due to the process or network connection being forcibly killed. Correlator-integrated JMS includes workarounds for many known third-party bugs in the JMS providers that Apama supports to make life easier for customers. However, it is not possible to find workarounds for all problems. Therefore Apama encourages customers to familiarize themselves with the release notes and outstanding bugs lists published by their JMS vendor — ideally before selecting a vendor — and to conduct sufficient

testing early in the application development process to allow for a change of JMS vendor if required.

Correlator-Integrated JMS Messaging

13 Managing the Dashboard Data Server and Display Server

Use of a deployed dashboard depends on a running Data Server or Display Server. You start, stop, and manage these Servers with the following executables, which are found in the `bin` directory of your Apama installation:

- `dashboard_server.exe` (Windows)
- `display_server.exe` (Windows)
- `dashboard_management.exe` (Windows)
- `dashboard_server` (UNIX)
- `display_server` (UNIX)
- `dashboard_management` (UNIX)

On Windows, you can also start the Data Server and Display Server from the Start menu.

Prerequisites

In order to start a Data Server or Display Server with all the necessary parameters to support a given deployment, you may need to obtain the following information from the Dashboard Builder:

- For Web-based deployments, the Data Sever or Display Server host and port that the builder supplied to the Deployment Configuration Editor when preparing the dashboard for deployment. See Using the Deployment Configuration Editor in the Preparing Dashboards for Deployment chapter of *Using Apama Studio*.
- The logical name for each Correlator as well as the host name and port for each *deployment* Correlator (if any) that was specified by the dashboard builder in the Apama tab of the Tools > Options... dialog prior to the generation of the deployment package with the Deployment Configuration Editor. See Changing Correlator Definitions for Deployment in the Preparing Dashboards for Deployment chapter of *Using Apama Studio*.

Managing the Dashboard Data Server and Display Server

Starting the Data Server or Display Server

You can start the Data Server and Display Server from the Windows Start menu. Select the following to start the Data Server:

Start > All Programs > Software AG > Apama 5.1 > Runtime > Dashboard Data Server

Select the following to start the Display Server

Start > All Programs > Software AG > Apama 5.1 > Runtime > Dashboard Display Server

The current directory for a Display Server that was started from the Start menu is the `dashboards` directory of your Apama work directory.

The executables for the Data Server and Display Server are found in the `bin` directory of your Apama installation.

You can start the Data Server with the executable `dashboard_server.exe` (on Windows) or `dashboard_server` (on UNIX).

You can start the Display Server with the executable `display_server.exe` (on Windows) or `display_server` (on UNIX). Start the Display Server from the `dashboards` directory of your Apama work directory.

You can run these servers by following these steps:

1. Select Start > All Programs > Software AG > Apama 5.1 > Apama Command Prompt
2. Change directories to`%APAMA_WORK%\dashboards`.
3. Invoke the executable from the command line.

Managing the Dashboard Data Server and Display Server

Description

These tools can be run without arguments, in which case they start a Server or on port `3278` (for Data Servers) or `3279` (for Display Servers) on the local host. (Note that these are the default ports used by the Deployment Configuration Editor—see Using the Deployment Configuration Editor in the Preparing Dashboards for Deployment section of *Using Apama Studio*). You can specify a different port with the `-d` or `--dataPort` option.

The `-c` or `--correlator` option allows you to specify the deployment host and port for a given correlator logical name. See Changing Correlator Definitions for Deployment in the Preparing Dashboards for Deployment chapter of the *Using Apama Studio*.

You can enable logging with the `-f` and `-v` (or `--logfile` and `--loglevel`) options or with the `log4j` properties file.

All this tool's options are described in the next section.

Starting the Data Server or Display Server

Options

Following are the command line options for this executable:

Table 8. Data Server and Display Server options

Options	Description
`-A | --sendAllData`	Send all data over the socket regardless of whether or not it has been updated.
`-a` | `--authUsers` bool	Specifies whether to enable user authentication. *bool* is one of `true` and `false`. By default, authentication is enabled. Set `--authUsers` to `false` for Web deployments for which authentication is performed by the Web layer.
`-c` | `--correlator` logical-name:host:port:raw-channel	Sets the Correlator host and port for a specified logical Correlator name. *raw-channel* is one of `true` and `false`, and specifies whether to use the raw channel for communication. This overrides the host, port, and raw-channel setting specified by the dashboard builder for the given Correlator

Options	Description

	logical name—see Changing Correlator Definitions for Deployment in the Preparing Dashboards for Deployment chapter of the *Using Apama Studio*.This option can occur multiple times in a single command. For example:
	`-c default:localhost:15903:false`
	`-c work1:somehost:19999:false`
	These options set the host and port for the logical names `default` and `work1`.

`-d` | `--dataPort` port	Data Server or Display Server port to which Viewers (for local deployments) or the data servlet (for Web deployments) will connect in order to receive event data. If not specified, the default port (3278 for Data Servers and 3279 for Display Servers) is used.
`-E` | `--purgeOnEdit` bool	Specifies whether to purge all trend data when a Scenario instance is edited. *bool* is one of `true` and `false`. If this option is not specified, all trend data is purged when an instance is edited. In most cases this is the desired mode of operation.
`-f` | `--logfile` file	Full pathname of the file in which to record logging. If this option is not specified, the options in the log4j properties file will be used.
`-G` | `--trendConfigFile` file	Trend configuration file for controlling trend-data caching.
`-h` | `--help`	Emit usage information and then exit.
`-J` | `--jaasFile` file	Full pathname of the JAAS initialization file to be used by the Data Server or Display Server. If not specified, the Server uses the file `JAAS.ini` in the `lib` directory of your Apama installation.
`-L` | `--xmlSource` file	XML data source file. If *file* contains static data, append `:0` to the file name. This signals Apama to read the file only once.
`-m` | `--connectMode` mode	Correlator-connect mode. *mode* is one of `always` and `asNeeded`. If `always` is specified all Correlators are connected to at startup. If `asNeeded` is specified, the Data Server or Display Server connects to Correlators as needed. If this option is not specified, the Server connects to Correlators as needed.
`-N` | `--name` name	Component name for identification in correlator
`--namedServer` logical-name:host:port	Sets the host and port for a specified logical Data Server name. This overrides the host and port specified by the dashboard builder for the given server logical name. This option can occur multiple times in a single command. See "Working with multiple Data Servers" on page 338 for more information.

Options	Description
`--namedServerMode`	Dashboard data server only. Specify this option when you start a data server that is used as a named server by a display-server deployment. See "Working with multiple Data Servers" on page 338 for more information.
`-O` | `--optionsFile` file	Full path of `OPTIONS.ini`
`-P` | `--maxPrecision` n	Maximum number of decimal places to use in numerical values displayed by dashboards. Specify values between 0 and 10, or -1 to disable truncation of decimal places. A typical value for *n* is `2` or `4`, which eliminates long floating point values (for example, 2.2584435234). Truncation is disabled by default.
`-p | --port port`	Port on which this Data Server or Display Server will listen for management operations. This is the port used for communication between the Server and the `dashboard_management.exe` process (on Windows) or `dashboard_management` process (on UNIX).
`-Q | --queueLimit size`	Set the server output queue size to *size*. This changes the default queue size for each client that is connected to the server.
`-R | --purgeOnRemove bool`	Specifies whether to purge all Scenario data when an instance is removed. *bool* is one of `true` and `false`. If this option is not specified, all Scenario data is purged when an instance is removed.
`-r | --cacheUsers bool`	Specifies whether to cache and reuse user authorization information. *bool* is one of `true` and `false`. Specifying `true` can improve performance, because users are authorized only once (per Data Server or Display Server session) for a particular type of access to particular Scenario or Scenario Instance.
`-s | --ssl`	Enable secure sockets for client communication. When secure sockets are enabled, the Data Server will encrypt data transmitted to Dashboard Viewers. Encryption is done using the strongest cipher available to both the Data Server and Viewer. SSL certificates are not supported. The Display Server does not support this option.
`-T | --maxTrend depth`	Maximum depth for trend data, that is, the maximum number of events in trend tables. If this option is not specified, the maximum trend depth is 1000. Note that the higher you set this value, the more memory the Data Server or Display Server requires, and the more time it requires in order to display trend and stock charts.

Options	Description
`-t | --cacheAuthorizations bool`	Cache and reuse scenario instance authorizations. Caching authorizations is enabled by default. When caching is enabled, authorization checks are performed only once per user for each scenario or data view they access. Disabling caching allows the current state of the scenario or data view to be used in the authorization check, but can degrade performance.
`-u | --updateRate rate`	Data update rate in milliseconds. This is the rate at which the Data Server or Display Server pushes new data to deployed dashboards in order to inform them of new events received from the Correlator. *rate* should be no lower than 250. If the Dashboard Viewer is utilizing too much CPU you can lower the update rate by specifying a higher value.If this option is not specified, an update rate of 500 milliseconds is used.
`-V | --version`	Emit program name and version number and then exit.
`-v | --loglevel level`	Logging verbosity. *level* is one of `FATAL`, `ERROR`, `WARN`, `INFO`, `DEBUG`, and `TRACE`. If this option is not specified, the options in the log4j properties file will be used.
`-X | --extensionFile file`	Full pathname of the extensions initialization file to be used by the Data Server or Display Server. If not specified, the Server uses the file `EXTENSIONS.ini` in the `lib` directory of your Apama installation.
`-x | --queryIndex table-name:key-list`	Add an index for the specified SQL-based instance table with the specified compound key. *table-name* is the name of a scenario or DataView. *key-list* is a comma-separated list of varaible names or field names. If the specified scenario or DataView exists in multiple correlators that are connected to the dashboard server, the index is added to each corresponding data table.Example: `--queryIndex Products_Table:prod_id,vend_id` You can only add one index per table, but you can specify this option multiple times in a single command line in order to index multiple tables.
`-Y | --enhancedQuery`	Make SQL-based instance tables available as data tables for visualization attachments. See Attaching Dashboards to Correlator Data in *Building Dashboards*.
`-z | --timezone zone`	Default time zone for interpreting and displaying dates. *zone* is either a Java timezone ID or a custom ID such as `GMT-8:00`. Unrecognized IDs are treated as GMT. See Appendix A of the *Dashboard Viewer* guide for the complete listing of permissible values for *zone*.

Options	Description
`--inclusionFilter value`	Set scenario inclusion filters. Use this option to control scenario/DataView discovery. If not specified, all scenario/DataViews will be discovered and kept in the memory of the dashboard processes, which can be expensive. For example, to include only the `DV_Weather` DataView, specify `--inclusionFilter DV_Weather`. The value can be a comma-separated list of scenario or DataView IDs. If you specify an inclusion filter, any specified exclusion filters are ignored.
`--exclusionFilter value`	Set scenario exclusion filters. Use this option to exclude specific scenarios/DataViews from being kept in the memory of the dashboard processes. If neither exclusion filters nor inclusion filters are specified, all scenario/DataViews will be discovered and kept in the memory of the dashboard processes, which can be expensive. The value can be a comma-separated list of scenario or DataView IDs. If an inclusion filter is specified, any exclusion filters are ignored.

Starting the Data Server or Display Server

Controlling Update Frequency

The correlator sends update events to the Data Server, Display Server, or any clients using the Scenario Service API, whenever the values of fields or output variables in your DataViews or scenarios change. If you have DataViews or scenarios that update frequently, you might need to reduce the frequency of update events sent by the correlator.

You can adjust settings per scenario definition or globally. The global value is used where a given scenario definition has no specific setting. The per-definition values always take precedence over the global values.

The `com.apama.scenario.ConfigureUpdates` event controls the sending of updates for all Apama-supplied monitors that the ScenarioService can be used with (that is, scenarios as generated by Event Modeler, DataViews, and MemoryStore tables with `exposeMemoryView` or `exposePersistentView` set in their schemas).

A `ConfigureUpdates` event consists of the following:

- `scenarioId` (type `string`): May be the empty string to modify the global values, or a definition's `scenarioId`.
- `Configuration` (type `dictionary (string, string)`): Configuration key and values. Key can be one of:
  - `sendThrottled` (type `boolean`): Whether to send throttled updates (on the `scenarioId.Data` channel). The default is `true`.
  - `sendRaw` (type `boolean`): Whether to send every update (on the `scenarioId.Data.Raw` channel). The default is `true`.

- throttlePeriod (type float): Throttle period in seconds. A zero value indicates no throttling. The default is 0.0.
- routeUpdates (type boolean): Whether to route Update events for the benefit of MonitorScript running in the correlator. The default is false.
- sendThrottledUser: (boolean): Whether to send throttled updates on a per-user channel. The default is false.
- sendRawUser (type boolean): Whether to send raw updates on a per-user channel. The default is false.

Those with a User suffix are suitable for using with only custom clients that use ScenarioServiceConfig.setUsernameFilter() on their scenario service configuration.

For example, consider the following:

```
com.apama.scenario.ConfigureUpdates("Scenario_scenario1",
    {"sendRaw":"true"})
com.apama.scenario.ConfigureUpdates("",{"sendRaw":"false",
    "throttlePeriod":"0.1"})
com.apama.scenario.ConfigureUpdates("Scenario_scenario2",
    {"sendRaw":"true"})
com.apama.scenario.ConfigureUpdates(Sscenario_scenario3",
    {"throttlePeriod":"1.0"})
```

The above examples configure Scenario_scenario1 and Scenario_scenario2 to send raw updates; Scenario_scenario3 to use a throttle period of 1 second; and all other scenarios to not send raw updates, and to use a throttle period of 0.1 seconds.

Earlier releases used the com.apama.scenario.SetThrottlingPeriod(x) event. Note that the use of the ConfigureUpdates events allows greater flexibility than the SetThrottlingPeriod event (which only controlled sending of throttled updates for all scenarios).

The use of com.apama.scenario.SetThrottlingPeriod(x) should be replaced with

```
com.apama.scenario.ConfigureUpdates("", {"throttlePeriod":"x"})
```

Note that by default, routeUpdates is false, so any EPL that relies on Update (and other scenario control events) to be routed should route a ConfigureUpdates event for the scenarioIds it is interested in to route Updates. Note that scenarios exported as blocks will do this in the generated block code.

The latest values are always used — thus it is not advisable for a client to send an event requesting (for example) raw updates and then undo this when it disconnects, as that will affect other clients. The recommendation is that the administrator should configure settings at initialization time. routeUpdates is an exception, but be aware that scenarios for which a block is generated will route updates after a scenario using that block has been injected and deleted, which may not be absolutely required.

Runtime performance of scenarios and dataviews can be improved by setting sendRaw and routeUpdates to false and throttlePeriod to a non-zero value. In this case, the cost of an update is reduced (as the Update events are only generated when needed, and if throttling, they are only needed at most once every throttlePeriod).

Managing the Dashboard Data Server and Display Server

Configuring Trend-Data Caching

By default, dashboard servers (Data Servers and Display Servers) collect trend data for all numeric output variables of scenarios and data views running in their associated correlators. This data is cached in preparation for the possibility that it will be displayed as historical data in a trend chart when a dashboard starts up. Without the cache, trend charts would initially be empty, with new data points displaying as time elapses.

Advanced users can override the default caching behavior on a given server, and control caching in order to reduce memory consumption on that server, or in order to cache variables that are not cached by default, such as non-numeric variables.

Important: In many cases, Server performance can be improved by overriding the default caching behavior, and suppressing the caching of those output variables for which trend-chart historical data is not required.

Important: Caching trend data for string variables is very costly in terms of memory consumption.

You control caching with a *trend configuration file*, which allows you to specify the following:

- Individual variables to cache
- Classes of variables to cache
- Default caching rules
- Trend depths (number of data points to maintain) for each scenario and data view

You do not need to provide a trend configuration file. If you provide no trend configuration file, dashboard servers use the default caching behavior described above.

Trend charts can include variables whose trend data is not cached, but they will display no historical (pre-dashboard-startup) data for those variables.

When a Data Server or Display Server starts, it uses the trend configuration file specified with the `-G` option, if supplied. Otherwise it uses the file `trend.xml` in the `dashboards` directory of your Apama work directory, if there is one. (Note, Apama provides an example trend configuration file, *APAMA_HOME*`\etc\dashboard_onDemandTrend.xml`, that you can copy to *APAMA_WORK*`\dashboards\trend.xml` as a basis for a trend configuration file.) Otherwise, it uses the default caching behavior described above.

Here is a sample configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
   <trend>
    <item type="SCENARIO" correlator="*" name="*"
      vars="ALL_NUMERIC_OUTPUT" depth="10000"/>
     <item type="SCENARIO" correlator="*" name="Scenario_scenario1"
      vars="LIST" depth="5000">
        <var name="A"/>
        <var name="B"/>
    </item>
    <item type="DATAVIEW" correlator="production" name="DV_dataview1"
      vars="LIST" depth="5000">
        <var name="A"/>
        <var name="B"/>
    </item>
  </trend>
 </config>
```

This file specifies the following:

- For Scenario_scenario1 in all correlators, cache trend data for variables A and B with a maximum trend depth of 5000.
- For all other scenarios, cache all numeric output variables with a maximum trend depth of 10,000.
- For DV_dataview1 in correlator production, cache variables A and B with a maximum trend depth of 5000.
- For all other data views, cache no trend data.

In general, a trend configuration file is an XML file that includes of one or more item elements with the following attributes:

- type:SCENARIO or DATAVIEW
- correlator: Logical name of correlator. Use * for if the item applies to all correlators
- name: Scenario or data view ID. Use * if the item applies to all scenarios or data views.
- vars: Class of variables to cache trend data for. Specify one of the following:
  - LIST: Cache the individual variables that are listed in var sub-elements.
  - ALL: Cache all input and output variables.
  - ALL_OUTPUT: Cache all output variables.
  - ALL_NUMERIC_OUTPUT: Cache all numeric output variables.
- depth: Maximum depth of trend data to cache.

If the vars attribute of an item element is LIST, the element has zero or more var sub-elements. Each var element has single attribute, name, which specifies the name of a scenario variable or data view field.

The item elements are nested in a trend element, which is nested within a config element.

If a particular data view or scenario on a given correlator matches multiple item elements in a server's trend configuration file, the server chooses the *best-matching*item and caches the variables specified in that item. Following are the ways, in order from best to worst, in which an item can match a data view or scenario on a given correlator:

1. Fully resolved: Exact match for both correlator name and scenario or data-view name
2. Wildcard correlator: Wildcard correlator and exact match for scenario or data-view name
3. Wildcard scenario: Exact match for correlator name and wildcard scenario or data-view
4. Fully wildcarded: Wildcard correlator and wildcard scenario or data view

If there are multiple best matches, the last match is used.

Consider, for example, scenarios named Scenario_scenario1 and Scenario_scenario2, correlators named production and development, and the following item elements:

```
1 <item type="SCENARIO" correlator="production" name="Scenario_scenario1"
    vars="LIST" depth="5000">
2 <item type="SCENARIO" correlator="*" name="Scenario_scenario1" vars="LIST"
    depth="5000">
3 <item type="SCENARIO" correlator="production" name="*" vars="LIST"
    depth="5000">
4 <item type="SCENARIO" correlator="*" name="*" vars="LIST"
    depth="5000">
```

Scenario_scenario1 running on production best matches item*1*.

Scenario_scenario1 on development best matches item*2*.

Scenario_scenario2 on production best matches item*3*.

Scenario_scenario2 on development best matches item*4*.

Below are some additional sample configuration files. The following file caches trend data for all input and output variables:

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
   <trend>
    <item type="SCENARIO" correlator="*" name="*" vars="ALL"
      depth="10000"/>
    <item type="DATAVIEW" correlator="*" name="*" vars="ALL"
      depth="10000"/>
  </trend>
 </config>
```

The following caches trend data for all numeric output variables, the default behavior:

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
   <trend>
    <item type="SCENARIO" correlator="*" name="*"
      vars="ALL_NUMERIC_OUTPUT" depth="10000"/>
    <item type="DATAVIEW" correlator="*" name="*"
      vars="ALL_NUMERIC_OUTPUT" depth="10000"/>
  </trend>
 </config>
```

The following caches no data, which results in trend-data collection only on demand:

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
   <trend>
  </trend>
 </config>
```

The following caches a single variable for a single scenario:

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
   <trend>
    <item type="SCENARIO" correlator="*" name="Scenario_scenario1"
      vars="LIST" depth="5000">
        <var name="PRICE"/>
    </item>
  </trend>
 </config>
```

Managing the Dashboard Data Server and Display Server

Managing Connect and Disconnect Notification

Whenever a dashboard connects to or disconnects from a Data Server or Display Server, the server sends a special notification event to all connected correlators that include the Dashboard Support bundle.

The events are defined as follows:

```
event DashboardClientConnected {
  string userName;
  string sessionId;
  dictionary<string,string> extraParams;
}event DashboardClientDisconnected {
  string userName;
  string sessionId;
  dictionary<string,string> extraParams;
}
```

`userName` specifies the user name with which the dashboard was logged in to the server.

`sessionID` is a unique identifier for the dashboard's session with the server.

`extraParams` may be used in a future release.

Note that the circumstances under which a dashboard disconnects from a server include but are not limited to the following:

- End user exits the Dashboard Viewer or Web browser in which a dashboard is loaded.
- End user exits a Web browser tab in which a dashboard is loaded.
- Network failure causes loss of connectivity to Viewer or Web browser in which a dashboard is loaded.

Note also that disconnect notification might be sent only after a timeout period rather than immediately upon loss of connection.

Follow these steps to manage connect and disconnect notification:

1. Ensure that the Dashboard Support bundle is loaded into all relevant correlators.
2. Use scenarios or monitors to process `DashboardClientConnected` and `DashboardClientDisconnected` events. Base processing on the values of the `userName` and `sessionId` fields.

Managing the Dashboard Data Server and Display Server

Working with multiple Data Servers

Deployed dashboards have a unique associated default Data Server or Display Server. For Web-based deployments, this default is specified in the Startup and Server section of the Deployment Configuration Editor. For Viewer deployments, it is specifed upon Viewer startup. By default, the data-handling involved in attachments and commands is handled by the default server, but advanced users can associate non-default Data Servers with specific attachments and commands. This provides additional scalability by allowing loads to be distributed among multiple servers. This is particularly useful for Display Server deployments. By deploying one or more Data Servers behind a Display Server, the labor of display building can be separated from the labor of data handling. The Display Server can be dedicated to building displays, while the overhead of data handling is offloaded to Data Servers.

Apama supports the following multiserver configurations:

- Builder with multiple Data Servers. See "Builder with multiple Data Servers" on page 339.
- Viewer with multiple Data Servers. See "Viewer with multiple Data Servers" on page 340.
- Display Server (thin client) deployment with multiple Data Servers. See "Display Server deployments with multiple Data Servers" on page 341.
- Applet or WebStart deployment with multiple Data Servers. See "Applet and WebStart deployments with multiple Data Servers" on page 342.

The Attach to Apama and Define ... Command dialogs (except Define System Command) include a Data Server field that can be set to a Data Server's logical name. To associate a logical name with the Data Server at a given host and port, developers use the Data Server tab in the General tab group of the Application Options dialog (select ToolsOptions in Builder).

For Display Server (thin client) deployments, you must use the option `--namedServerMode` whenever you start named Data Servers. See "Display Server deployments with multiple Data Servers" on page 341.

The logical Data Server names specified in the Builder's Application Options dialog are recorded in the file `OPTIONS.ini`, and the deploymenbt wizard incorporates this information into deplyments. You can override these logical name definitions with the `--namedServer` *`name`*`:`*`host`*`:`*`port`* option to the Builder, Viewer, Data Server or Display Server executable. Below is an example. This is a sequence of command line options which should appear on a single line as part of the command to start the executable:

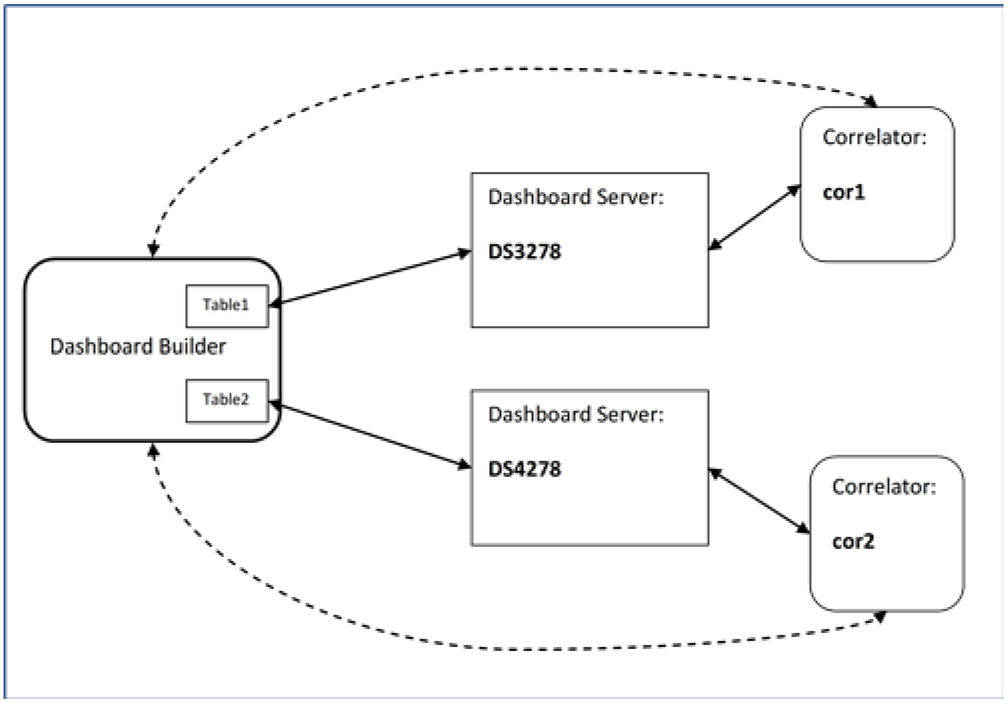
```
--namedServer Server1:ProductionHost_A:3278 --namedServer Server2:ProductionHost_B:4278 --
namedServer Server3:ProductionHost_C:5278
```

Here `Server1`, `Server2` and `Server3` are the server logical names.

Managing the Dashboard Data Server and Display Server

Builder with multiple Data Servers

Builder maintains connections with the Data Servers named in attachments and commands. Note that it connects directly to the correlator (dotted lines in the figure below) in order to populate dialogs with metadata. Correlator event data is handled by the Data Servers.



You can override the logical server names specified in the Application Options dialog with the `--namedServer` *`name`*`:`*`host`*`:`*`port`* option to the Builder executable. Below is an example. This is a sequence of command line options which should appear on a single line as part of the command to start the executable:

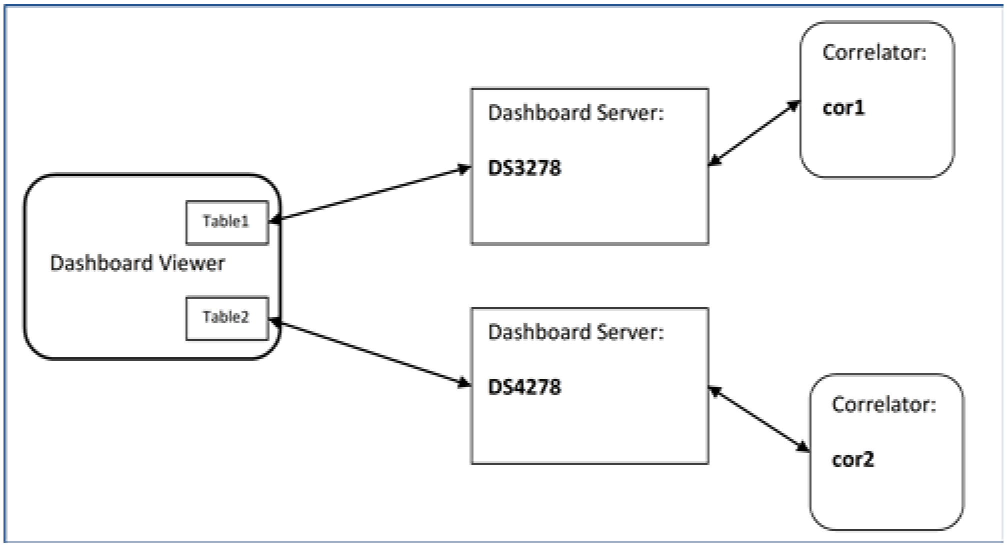
```
--namedServer Server1:ProductionHost_A:3278 --namedServer Server2:ProductionHost_B:4278 --
namedServer Server3:ProductionHost_C:5278
```

Here `Server1`, `Server2` and `Server3` are the server logical names.

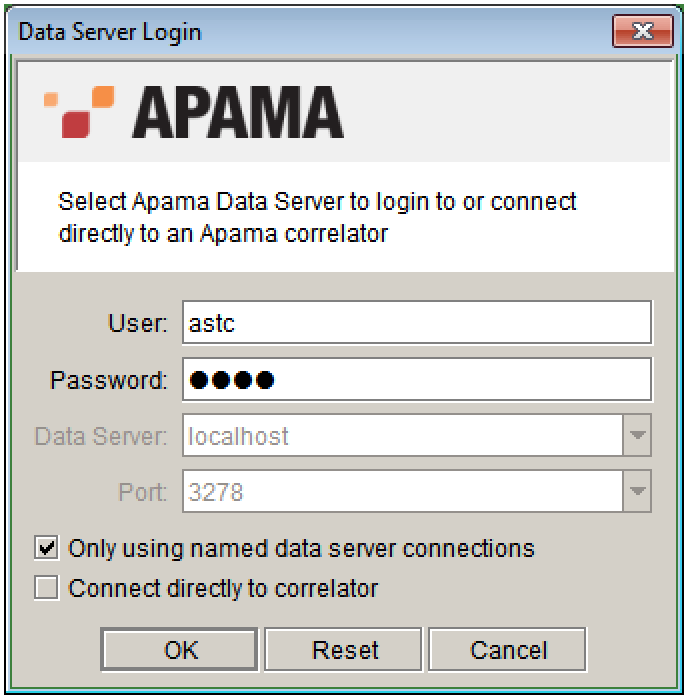
Working with multiple Data Servers

Viewer with multiple Data Servers

Viewer maintains connections with the Data Servers named in attachments and commands of opened dashboards.



In the Data Server Login dialog (which appears upon Viewer startup), end users enter the host and port of the default Data Server (or accept the default field values). If all attachments and commands use named Data Servers, end users can check the Only using named data server connections check box and omit specification of a default server.



Data Server Login

Select Apama Data Server to login to or connect directly to an Apama correlator

User: astc

Password: ●●●●

Data Server: localhost

Port: 3278

☑ Only using named data server connections

☐ Connect directly to correlator

The logical data server names specified in the Builder's Application Options dialog are recorded in the file `OPTIONS.ini`, which is found in the deployed `.war` file along with dashboard `.rtv` files. You can override these logical name definitions with the `--namedServer` *`name`*`:`*`host`*`:`*`port`* option to the Viewer executable. Below is an example. This is a sequence of command line options which should appear on a single line as part of the command to start the executable:

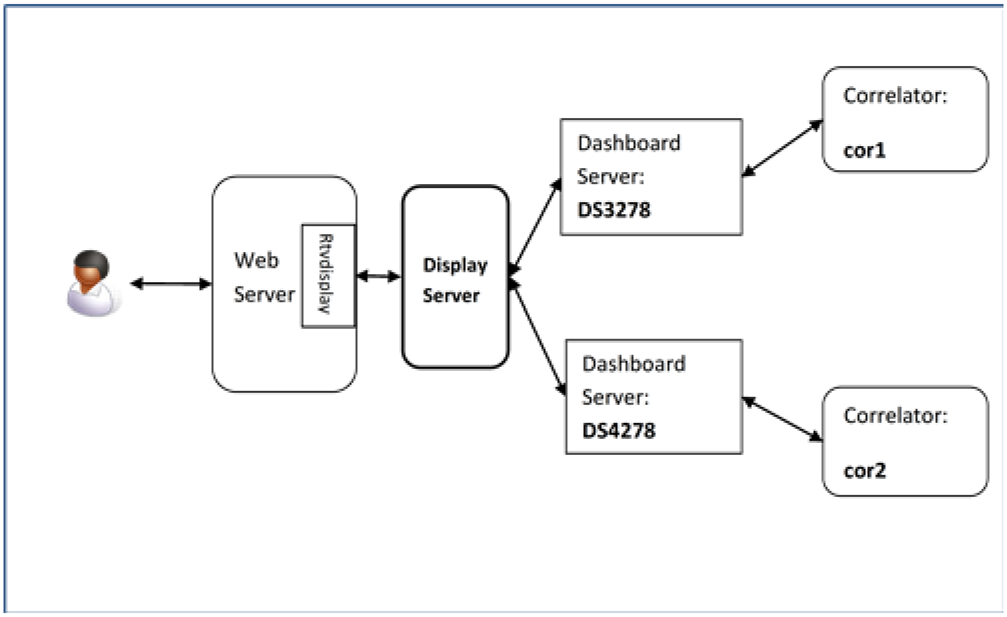
```
--namedServer Server1:ProductionHost_A:3278 --namedServer Server2:ProductionHost_B:4278 --namedServer Server3:ProductionHost_C:5278
```

Here `Server1`, `Server2` and `Server3` are the server logical names.

Working with multiple Data Servers

Display Server deployments with multiple Data Servers

The Display Server maintains connections with the Data Servers named in attachments and commands of its client dashboards.



Note: In a Display Server deployment, each named Data Server must be started with the `--namedServerMode` option.

The logical data server names specified in the Builder's Application Options dialog are recorded in the file `OPTIONS.ini`, which is used by the Deployment Wizard to define deployment logical names. You can override these logical name definitions with the `--namedServer` *`name`*`:`*`host`*`:`*`port`* option to the Display Server executable. Below is an example. This is a sequence of command line options which should appear on a single line as part of the command to start the executable:

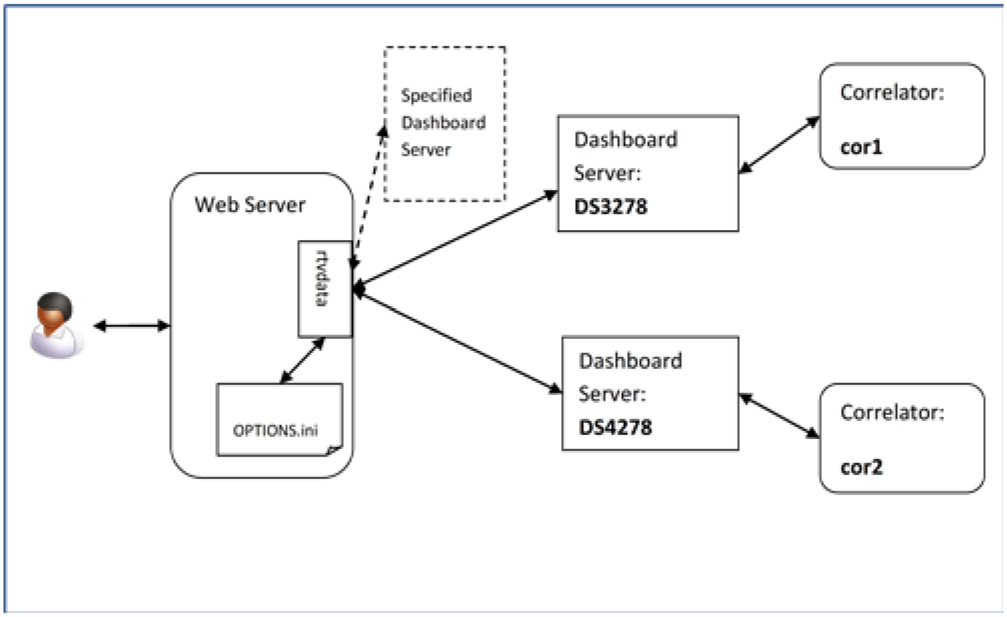
```
--namedServer Server1:ProductionHost_A:3278 --namedServer Server2:ProductionHost_B:4278 --
namedServer Server3:ProductionHost_C:5278
```

Here `Server1`, `Server2` and `Server3` are the server logical names.

Working with multiple Data Servers

Applet and WebStart deployments with multiple Data Servers

Applet and WebStart dashboards maintain connections with the Data Servers named in their attachments and commands.



In this diagram, the dotted line indicates the connection to the default Data Server, which is specified in the Startup and Server section of the Deployment Configuration Editor. The default must be running only if some attachments or commands don't specify a named Data Server.

The logical data server names specified in the Builder's Application Options dialog are recorded in the file `OPTIONS.ini`, which is used by the Deployment Wizard to define deployment logical names.

Working with multiple Data Servers

Managing and stopping the Data Server and Display Server

You stop a Data Server or Display Server and perform certain Data Server or Display Server management operations with the executable `dashboard_management.exe` (on Windows) or `dashboard_management` (on UNIX), which is found in the `bin` directory of your Apama installation. Note that in the environment of the Apama Command Prompt, the `bin` directory is appended to the PATH environment variable.

Managing the Dashboard Data Server and Display Server

Description

You can use this tool to shut down, deep ping, or get the process ID of a Data Server or Display Server on a specified host and port. A successful deep ping verifies that the Server is responding to requests. You can also use this tool to generate a dashboard deployment package, and to sign .jar files as part of deployment-package generation.

When you invoke this tool, you can specify the host and port of the Server you want to manage. For the port, specify the port that was specified with the -p or --port option when the desired Server was started. If the -p or --port option was not specified, you do not need to supply this option. It defaults to the default management port (28888).

All this tool's options are described in the next section.

Managing and stopping the Data Server and Display Server

Options

The tool `dashboard_management.exe` (on Windows) or `dashboard_management` (on UNIX) takes a number of command line options. These are:

Table 9. dashboard_management options

Option	Description
`-a | --alias alias`	Use the alias alias in order to sign the `.jar` files to be included in the deployment package specified by the `-y` or `--deploy` option and the `-c` or `--config` option. Specify the keystore and password with the `-k` or `--keystoreFile` option and the `-o` or `--password` option.
`-c | --config`	Generate a deployment package for the configuration named config-name. Specify the file that defines the configuration with the `-y` or `--deploy` option. Specify the `.rtv` files to use with the `-r` or `--rtvPath` option.
`-D | --displayServer`	Run against Display Server
`-d | --deepping`	Deep-ping the component.
`-e | --encryptString password`	Generate an encrypted version of password. This is useful when you manually add an SQL data source by entering information directly into `OPTIONS.ini`.
`-h | --help`	Display usage information.
`-I | --invalidateAll`	Invalidate all user authentications.
`-i | --invalidateUser username`	Invalidate a user authentication.
`-j | --jar jar file`	Name of a third-party jar file to sign. You can specify multiple `-j | --jar` arguments if you have multiple jar files to sign.
`-k | --keystoreFile path`	Use the keystore file designated by path in order to sign the `.jar` files to be included in the deployment package specified by the `-y` or `--deploy` option and the `-c` or `--config` option. Specify the alias and password with `-a` or `--alias` option and the `-o` or `--password` option. Ensure that the environment variable `JAVA_HOME` is set to a Java Development Kit (JDK).

Option	Description
`-n | --hostname host`	Connect to component on host. If not specified, `localhost` is used.
`-N | --namedServerMode`	
`-p | --password password`	Use the password password in order to sign the `.jar` files to be included in the deployment package specified by the `-y` or `--deploy` option and the `-c` or `--config` option. Specify the keystore and alias with the `-k` or `--keystoreFile` option and the `-a` or `--alias` option.
`-p | --port port`	Connect to component on port. Specify the port that was specified with the -p or --port option when the component was started. If the -p or --port option was not specified, you do not need to supply this option. It defaults to the default management port (28888).
`-r | --rtvPath path`	Generate a deployment package with the `.rtv` files in the directory designated by path. Specify the deployment configuration to use with the `-y` or `--deploy` option and the `-r` or `--rtvPath` option.
`-s | --shutdown reason`	Shut down the component with reason reason.
`-U | --update path`	Update the specified Release 2.4 `.rtv` file or files so that they are appropriate for use with this Apama release. path is the pathname of a file or directory. If path specifies a directory, all `.rtv` files in the directory are updated.
`-v | --verbose`	Emit verbose output, including the startup settings (such as `dataPort` and `updateRate`) of the dashboard server connect to.
`-V | --version`	Display program name and version number and then exit.
`-W | --waitFor`	Wait for component to be available.
`-y | --deploy path`	Generate a deployment package for a configuration defined in the dashboard configuration file designated by path. Specify the configuration name with the `-c` or `--config` option. Specify the `.rtv` files to use with the `-r` or `--rtvPath` option.

Managing and stopping the Data Server and Display Server

14 Administering Dashboard Security

Deployed dashboards are protected by the authorization and authentication facilities provided by Apama and your application server.

Apama's dashboard authentication facility prompts users for credentials before allowing any access to deployed dashboards. It gives you the ability to customize authentication by either configuring your application server to use the security realm and authentication service of your choice or by supplying any JAAS-compliant authentication module as a plug-in to the Data Server or Display Server. See "Administering authentication" on page 348.

Apama's authorization facility includes access control that gives you the ability to control who can use a given dashboard. The facility also gives you the ability to control who can use dashboards to gain a given type of access to a given scenario, scenario instance, DataView definition, or DataView item. And it gives the ability to control who can send events from dashboards using the Send Event command. See "Administering authorization" on page 351.

In addition to authenticating and authorizing users, you need to consider how you will protect data sent to dashboards. This is discussed in "Securing communications" on page 358.

Administering authentication

For dashboards deployed as simple Web pages, applets, or Web Start applications, authentication can be performed by your application server.

For dashboards deployed as local applications, or dashboards using the WebSphere application server, authentication is performed both at the dashboard and by the Data Server or Display Server. When a user starts the Dashboard Viewer, a login dialog appears, which prompts the user for a user name and password (as well as for the host and port of the Data Server or Display Server to connect to). The information entered is used to authenticate the user against the authentication service of your choice. Authenticated users are allowed to connect to the Server. See "Authentication for local and WebSphere deployments" on page 348 for more information.

Whenever a dashboard connects to or disconnects from a Data Server or Display Server, the server sends a special notification event to all correlators that are connected to it, provided that your project includes the Dashboard Support bundle. Your monitors or scenarios can make use of these events to implement further authentication-related administration. See "Managing Connect and Disconnect Notification" on page 337 in "Managing the Dashboard Data Server and Display Server" on page 327.

Administering Dashboard Security

Authentication for local and WebSphere deployments

Both the Dashboard Viewer and Data Server or Display Server provide authentication through the Java Authentication and Authorization Service (JAAS). JAAS is the Java implementation of the Pluggable Authentication Module (PAM) framework. It provides a pluggable framework for user authentication and authorization.

The Java JRE provides authentication modules for use with common authentication services such as LDAP and Kerberos. It also supports the development of new authentication modules for use with custom or proprietary authentication services. The Data Server, Display Server, and Viewer will work with any authentication module that complies with the JAAS specification. This openness allows you to integrate dashboards with your existing authentication service.

Important: Default authentication for local deployments uses a no-op implementation of the JAAS login module protocol. All user name/password pairs are authenticated. You can customize authentication for local deployments by supplying your own implementation of the interface `javax.security.auth.LoginModule`.

Administering Dashboard Security

Dashboard Login Modules Provided by Apama

Apama provides the following JAAS login modules in the package `com.apama.dashboard.security`:

- `NoOpLoginModule`: Does no username or password validation. This is used by default for the Dashboard Builder, Viewer, Data Server, and Display Server.
- `UserFileLoginModule`: Loads user and role definitions from an XML file. See "Installing UserFileLoginModule" on page 351 for more information.
- `LdapLoginModule`: Authenticates against an LDAP service.

In addition, the Java Runtime Environment (JRE) that is shipped with Apama includes several JAAS login modules:

- `JndiLoginModule`: The module prompts for a username and password and then verifies the password against the password stored in a directory service configured under JNDI.
- `KeyStoreLoginModule`: Provides a JAAS login module that prompts for a key store alias and populates the subject with the alias's principal and credentials.
- `Krb5LoginModule`: This login module authenticates users using Kerberos protocols.

Authentication for local and WebSphere deployments

Developing custom login modules

When developing your implementation of `LoginModule`, note that the Data Server and Display Server’s built-in `CallbackHandler` currently recognizes only the `NameCallback` and `PasswordCallback`.

See "Example: Implementing LoginModule" on page 359 for a sample implementation of `LoginModule`, which you can also find under `samples\dashboard_studio\tutorial\src` in your Apama installation.

Authentication for local and WebSphere deployments

Installing login modules

`NoOpLoginModule` is used by default for the Dashboard Builder, Viewer, Data Server, and Display Server. To change this, you must install the login module or modules that you want to use instead. To install login modules, do both of the following:

- Specify the login modules to use in the file `JAAS.ini` in the `lib` directory of your Apama installation.
- Create a `jar` file that includes your `LoginModule` implementation or implementations, and add the `jar` or its directory to `APAMA_DASHBOARD_CLASSPATH` (changes to this environment variable are picked up by dashboard processes only at process startup) or else add the `jar` or its directory to

the list of External Dependencies in your project's Dashboard Properties (In Apama Studio, right click on your project and select Properties, expand Apama, select Dashboard Properties, activate the External Dependencies tab, and click the Add External button).

If your login module has dependencies on other `.jar` files, add these `.jar` files to the manifest of the login module `.jar` file.

Apama Studio allows you to sign your `.jar` files when you create a deployment package—see Preparing Dashboards for Deployment in *Using Apama Studio*.

Note: The login module you install can affect the Data Server or Display Server authorization behavior. If you install `UserFileLoginModule`, for example, the default Scenario authority will provide expanded access to users with the `apama_admin` role. For such users, it will grant view, edit, and delete access to all Scenario instances (in addition to granting such access to Scenario-instance owners). See "Providing a login module that supports a Scenario or Event Authority" on page 358 for more information.

If you are installing a login module provided by Apama (see "Dashboard Login Modules Provided by Apama" on page 349), you do not need to create a `jar` file as described above, as this class is provided with your Apama installation and is included in an existing `jar`.

`JAAS.ini` uses the standard JAAS configuration file format. Each entry in the file associates an application with a login module together with a specification of the module's parameter values. Here is a `JAAS.ini` that specifies the `UserFileLoginModule` for the Dashboard Viewer and Data Server applications:

```
/*
* Dashboard Builder security configuration.
*/
builder {
   com.apama.dashboard.security.NoOpLoginModule required
   debug=false;
};
/*
* Dashboard Viewer security configuration.
*/
viewer {
   debug=false;
   com.apama.dashboard.security.UserFileLoginModule required
       debug=false
       userFile="<INSTALL_PATH>\\etc\\dashboard-users.xml"
       refreshDelay="5000";
};
/*
* Dashboard DataServer security configuration.
*/
dataServer {
   debug=false;
   com.apama.dashboard.security.UserFileLoginModule required
       debug=false
       userFile="<INSTALL_PATH>\\etc\\dashboard-users.xml"
       refreshDelay="5000";
};
```

Important: Do not change the login module associated with the Dashboard Builder.

Authentication for local and WebSphere deployments

Installing UserFileLoginModule

In a JAAS.ini file, you specify a module's parameter values with expressions of the form *formal-parameter=actual-parameter*. The login module UserFileLoginModule supports the following parameters:

- debug: true or false. Enable debug logging.
- validateUser: true or false. Set to false to disable password validation. This is for use in Web deployments where authentication is provide by your application server—see "Providing a login module that supports a Scenario or Event Authority" on page 358. In such a case, configure userFile to specify users for your application server. This results in the application server performing authentication and the Data Server handling authorization, in such a way that the application server and the Data Server obtain user-credentials information from the same file.
- userFile: Fully resolved name of the file with user and role definitions
- refreshDelay: Time in milliseconds to check for changes to the definition file (userFile). When it changes it is reloaded. This allows new users to be added.

Note: Installing UserFileLoginModulecan affect the Data Server's authorization behavior. In particular, if you install UserFileLoginModule, the default Scenario authority will provide expanded access to users with the apama_admin role. For such users, it will grant view, edit, and delete access to all Scenario instances (in addition to granting such access to Scenario-instance owners).

Authentication for local and WebSphere deployments

Installing LdapLoginModule

The LdapLoginModule uses JNDI to access naming and directory services. Sun's LDAP provider is used, and hence the InitialContext must be set up properly according to Sun's implementation. The environment settings must be specified in the JAAS.ini file. Here is an example:

```
viewer {
   com.apama.dashboard.security.LDAPLoginModule required
         ProviderURL="ldap://your.own.ldap.server:389"
         Authentication=simple
         Anonymous=false
         DN="uid=%,ou=City,ou=Region,ou=People,o=ACME Corporation"
         TLS=false;
};
```

Authentication for local and WebSphere deployments

Administering authorization

Apama's dashboard authorization facility includes access control that gives you the ability to restrict who can use a given Web-based dashboard—see "Dashboard access control" on page 353.

The example above configures the Dashboard Viewer to use LdapLoginModule.

Following are the supported environment settings:

- `ProviderURL` (required): Specifies the LDAP server and port, which are used to set the `java.naming.factory.initial` property.
- `Authentication` (required): Specifies the authentication mechanism to use. Specify `none`, `simple`, or `sasl_mech`. This value is used to set the `java.naming.security.authentication` property;
- `Anonymous` (optional; defaults to true): Specifies whether the `userPrincipal` and `userCredential` should be used when creating the `LdapContext`.
- `DN` (required): Specifies the user principal to be used when accessing the directory. This value is used (after patching with the user name) to set the `java.naming.security.principal` property. The user entered password is used in `java.naming.security.credentials`.

In the example above, `DN` is set to the following:

```
uid=%,ou=City,ou=Region,ou=People,o=ACME Corporation.
```

The `%` character is replaced by the login name entered by user.

- `TLS` (required): This specifies whether the LDAP server should start the Transport Security Layer extension. Supply `true` to specify that it should be started; supply `false` to specify that it should not be started.
- `Extra` (optional): Allows you to specify any extra parameters for setting the environment before creating the `LdapContext`. The function of these extra parameters is specific to your LDAP server, not the `LdapLoginModule`. Supply a semicolon-separated list of name/value pairs, where each pair has the form

name=value

Consider for example the following:

```
Extra=java.naming.referral=ignore;java.naming.security.protocol=ssl
```

This sets `java.naming.referral` to `ignore` and `java.naming.security.protocol` to `ssl`.

The facility also gives you the ability to control who can use dashboards for each of the following types of Scenario access:

- Viewing a given Scenario instance
- Editing a given Scenario instance
- Deleting a given Scenario instance
- Creating an instance of a given Scenario

In addition, you can control who can use dashboards for view access to DataView items. See "Default Scenario and DataView access control" on page 353 and "Customizing Scenario and DataView access control" on page 354.

You can also control who can send events from dashboards using the Send Event command. See "Send event authorization" on page 356

For Web deployments, some aspects of authorization (in particular, dashboard access control) are centered around the concepts of *users* and *roles*, as defined in the J2EE specification. Users and roles are introduced in "Users and roles" on page 353.

Users and roles

A *user* is an individual in a company or organization, who proves their identity to the Application Server by entering a password known only to them.

A *role* defines a capability to perform an operation or access to some entity on the Application Server, and might typically be held by a number of users.

Each user has zero or more associated roles.

For Web-based deployments, you create users and roles, and associate roles with users by using your application server.

Dashboard access control

By allowing you to associate roles with applet and Web Start dashboards, the dashboard authorization facility gives you the ability to control which users can access which dashboards.

You associate a role with a dashboard when you deploy it, by specifying a value for the Security Role field of the dashboard deployment form. If you don't specify a role in the Security Role field, the predefined role apama_customer is associated with the dashboard. In addition, the apama_admin role is automatically associated with every deployed dashboard.

A user who has logged into the Dashboard Directory Web site can access a given Web-deployed dashboard if and only if one of the dashboard's roles is among the user's roles. If the user's roles include none of the dashboard roles, the user will not see the dashboard in the Dashboard Directory.

For local dashboard deployments, you can restrict dashboard access by doing one of the following:

- Restricting the distribution of deployed dashboard files (the `.zip` file generated by the Deployment Configuration Editor)
- Using system security mechanisms in order to restrict access to the deployed dashboard files.

Default Scenario and DataView access control

By default, only the owner of a Scenario instance or DataView item can access it, unless the owner is specified as "*", in which case all users can access it.

When a user attempts to access a Scenario instance by using a dashboard, view, edit, and delete access are authorized if and only if one of the following is true:

- The user name supplied during application-server or Data-Server login matches the Scenario-instance owner.
- The Scenario instance owner is specified as "*".

Similarly, when a user attempts to access a DataView item by using a dashboard, view access is authorized if and only if one of the following is true:

- The user name supplied during application-server or Data-Server login matches the DataView-item owner.
- The DataView-item owner is specified as "*".

By default, any user can create an instance of any Scenario.

Edit, create, and delete access do not apply to DataView items, but see Send Event Authorization.

Customizing Scenario and DataView access control

You can customize Scenario and DataView access control by supplying a *Scenario Authority*, an implementation of the interface `com.apama.security.IScenarioAuthority`. This interface defines the methods `canView`, `canEdit`, `canDelete`, and `canCreate`, which must be implemented to return `true` or `false` for a given user and Scenario, Scenario instance, or DataView item. See "Providing a Scenario Authority" on page 354.

You might also need to supply a login module, an implementation of `javax.security.LoginModule`, in order to endow the instance of `javax.security.Subject` that represents the current end user with the characteristics that the Scenario or Event Authority requires. See "Providing a login module that supports a Scenario or Event Authority" on page 358.

Providing a Scenario Authority

You can provide an alternative to the default Scenario Authority by doing one of the following:

- Develop and install a custom Scenario Authority. See "Developing a custom Scenario Authority" on page 354 and "Installing a Scenario Authority" on page 356.
- Install `com.apama.dashboard.security.NoOpScenarioAuthority`, which allows full access by any user. This is useful for testing. See "Installing a Scenario Authority" on page 356.

Developing a custom Scenario Authority

In order to develop a a custom Scenario Authority, you must implement each `IScenarioAuthority` method (see "Implementing IScenarioAuthority methods" on page 354), which typically requires the use of `UserCredentials` access methods (see "Using UserCredential accessor methods" on page 357). When you compile your implementation, your classpath must be set appropriately (see "Compiling your Event Authority" on page 357).

Implementing IScenarioAuthority methods

Below is a description of each `IScenarioAuthority` method that you must define, including the following:

- Signature of the method
- When the method is called by the Data Server
- What the method should return

When each method is called depends on whether *authorization caching* is on—see the `-r` or `--cacheUsers` option.

This interface defines the following methods:

```
public boolean canView (
  UserCredentials credentials,
  IScenarioInstance instance
);
```

canView is called the first time (or, if authorization caching is off, every time) in a Data Server or Display Server session that the Server sends data from the specified Scenario instance or DataView item to an end user with the specified credentials. Your implementation should return `true` if the user with the specified credentials is authorized to view the specified instance or item; it should return `false` otherwise.

Note that if caching is off, `canView` is called very frequently, as specified by the update rate for the Data Server (see the description of the `-u` or `--updateRate` option). If your implementation renders calls to `canView` expensive, the performance of your dashboard will be significantly affected.

```
public boolean canEdit (
  UserCredentials credentials,
  IScenarioInstance instance
);
```

canEdit is called the first time (or, if caching is off, every time) a dashboard attempts to edit a Scenario instance. Your implementation should return `true` if the user with the specified credentials is authorized to edit the specified Scenario instance; it should return `false` otherwise. Does not apply to DataView items, but see Send Event Authorization.

```
public boolean canDelete (
  UserCredentials credentials,
  IScenarioInstance instance
);
```

canDelete is called the first time (or, if caching is off, every time) a dashboard attempts to delete a Scenario instance. Your implementation should return `true` if the user with the specified credentials is authorized to delete the specified Scenario instance; it should return `false` otherwise. Does not apply to DataView items, but see Send Event Authorization.

```
public boolean canCreate (
  UserCredentials credentials,
  IScenarioDefinition scenario
);
```

canCreate is called the first time (or, if caching is off, every time) a dashboard attempts to create a Scenario. Your implementation should return `true` if the user with the specified credentials is authorized to create an instance of the specified Scenario; it should return `false` otherwise. Does not apply to DataViews, but see Send Event Authorization.

Using UserCredential Accessor Methods

Your implementation of `IScenarioAuthority` will typically use the following public assessor methods of `com.apama.dashboard.security.UserCredentials`:

```
public String getUsername()
public String getPassword()
public Subject getSubject()
```

Your implementation may also use the following methods of com.apama.services.scenario.IScenarioInstance:

```
public String getOwner()
public Object getValue(String parameterName)
```

Compiling Your Scenario Authority

When you compile your implementation of `IScenarioAuthority`, be sure to put the following `jar` files on your classpath:

- `dashboard_studio.jar`
- `dashboard_client5.1.jar`

- engine_client5.1.jar

These jar files are in the lib directory of your Apama installation.

Installing a Scenario Authority

To install your authorization customization, do both of the following:

- Replace "com.apama.dashboard.security.DefaultScenarioAuthority" with the fully qualified name of your class in the file EXTENSIONS.ini, which is in the lib directory of your Apama installation.
- Create a jar file that contains your IScenarioAuthority implementation, and add the jar or its directory to APAMA_DASHBOARD_CLASSPATH (changes to this environment variable are picked up by dashboard processes only at process startup) or else add the jar or its directory to the list of External Dependencies in your project's Dashboard Properties (In Apama Studio, right click on your project and select Properties, expand Apama, select Dashboard Properties, activate the External Dependencies tab, and click the Add External button).

If your scenario authority has dependencies on other .jar files, add these .jar files to the manifest of the scenario authority .jar file.

Apama Studio allows you to sign your .jar files when you create a deployment package—see Preparing Dashboards for Deployment in *Using Apama Studio*.

If you are installing NoOpScenarioAuthority, you do not need to create a jar file as described above, as this class is provided with your Apama installation and is included in an existing jar.

The EXTENSIONS.ini specifies the scenario authority to use. This file identifies all the user supplied extension classes (including functions and commands). Here is a sample EXTENSIONS.ini:

```
function com.apama.dashboard.sample.SampleFunctionLibrary
command com.apama.dashboard.sample.SampleCommandLibrary
scenarioAuthority com.apama.dashboard.sample.SampleScenarioAuthority
```

This file installs a function library, a command library, and a Scenario authority.

If multiple authorities are specified, a user must be authorized by each.

Sample Custom Scenario Authority

You can find a sample implementation of IScenarioAuthority under samples \dashboard_studio\tutorial\src:

Send event authorization

By default, any user is authorized to send any event. However you can create a custom *event authority* that determines whether a given user is authorized to send a given event. An event authority is a Java class that implements the canSend method of the interface com.apama.dashboard.security.IEventAuthority:

```
boolean canSend (IUserCredentials credentials, Event event);
```

If canSend() returns true the user is allowed to send the event. If it returns false the user is not allowed to send the event and the attempt to send the event is treated as a command failure. Dashboard object property settings determine if this error is displayed to the user.

Using UserCredential accessor methods

Your implementation of IEventAuthority will typically use the following public assessor methods of com.apama.dashboard.security.UserCredentials:

```
public String getUsername()
public String getPassword()
public Subject getSubject()
```

Compiling your Event Authority

When you compile your implementation of IScenarioAuthority, be sure to put the following jar files on your classpath:

- dashboard_studio.jar
- dashboard_client5.1.jar
- engine_client5.1.jar

These jar files are in the lib directory of your Apama installation.

Installing your Event Authority

To install your authorization customization, do both of the following:

- Replace "com.apama.dashboard.security.NoOpEventAuthority" with the fully qualified name of your class in the file EXTENSIONS.ini, which is in the lib directory of your Apama installation.
- create a jar file that contains your IEventAuthority implementation, and place it in the directory %APAMA_WORK%\dashboards_deploy\lib. If you have other custom classes (for example, a custom login module—see "Developing custom login modules" on page 349), you can include them in the same .jar file or in a different .jar file.

If your event authority has dependencies on other .jar files, add these .jar files to the manifest of the event authority .jar file.

Apama Studio allows you to sign your .jar files when you create a deployment package—see Preparing Dashboards for Deployment in *Using Apama Studio*.

Two event authorities are provided with your installation:

- com.apama.dashboard.security.NoOpEventAuthority: Permits all users to send any event.
- com.apama.dashboard.security.DenyAllEventAuthority: Denies all users rights to send any event.

NoOpEventAuthority is the default event authority. Use a custom event authority when deploying your dashboards.

If you are installing DenyAllEventAuthority, you do not need to create a jar file as described above, as this class is provided with your Apama installation and is included in an existing jar.

Here is a portion of EXTENSIONS.ini as shipped:

```
# List of event authorities. An event authority is called to determine
# if a user has rights to send an event to a correlator. Each must implement
# the interface:
##     com.apama.dashboard.security.IEventAuthority
## Multiple authorities can be specified. They will be called in the order
# listed.
## Format:
```

```
##    eventAuthority <classname>
## NoOpEventAuthority - Allows all users to send events
eventAuthority com.apama.dashboard.security.NoOpEventAuthority
# DenyAllEventAuthority - Allows no users to send events
#eventAuthority com.apama.dashboard.security.DenyAllEventAuthority
# eventAuthority <your_class_name_here>
```

Providing a login module that supports a Scenario or Event Authority

When you implement a Scenario or Event Authority, the methods that you implement have a `UserCredentials` argument. Typical implementations retrieve an instance of `javax.security.auth.Subject` from the `UserCredentials` object, and use the `Subject`'s characteristics to determine whether to return `true` or `false` (that is, whether to grant or deny access).

The characteristics of a particular Subject (for example its associated roles, as returned by `Subject.getPrinciples`) are established by a JAAS login module that is called by the Data Server or Display Server. It is this module's responsibility to establish those characteristics on which the Scenario or Event Authority will rely.

For local deployments, this login module is responsible for authenticating the current end user (see "Authentication for local and WebSphere deployments" on page 348) as well as for setting the characteristics of the `Subject`. For Web deployments, this login module is responsible only for setting the characteristics of the `Subject` (since authentication is performed by application server.

For both Web and local deployments, the default Data Server and Display Server login module is `NoOpLoginModule`, which does *not* set any characteristics of the `Subject`. With this module, the `Subject` passed into `IScenarioAuthority` methods has no associated roles.

Typical implementations of `LoginModule` store the `Subject` passed into `LoginModule.initialize` as local state, and set the `Subject`'s characteristics in `LoginModule.commit`.

Note that `UserFileLoginModule` supports Scenario Authorities by setting `Subject` roles at the time of authentication. To use `UserFileLoginModule` in order to support Scenario Authorities for Web-based deployments (where authentication is performed by the application server), set `validateUser` to false when you install `UserFileLoginModule`—see "Installing UserFileLoginModule" on page 351.

For Web-based deployments, the Data Server and Display Server receive only user names (and not passwords) from the application server. This means that you cannot use a JAAS login module that requires both user names and passwords in order to authenticate users and retrieve their roles. To perform role based authorization for Web-based deployments, use a JAAS login module that can retrieve the roles for a user by using only the user name.

Securing communications

For local application deployments, where dashboards communicate directly with the Data Server, your options for securing dashboard communications include:

- Enabling secure sockets (SSL) in the Data Server
- Utilizing a secure channel (SSH) for all dashboard communication
- Utilizing a virtual public network (VPN) for all dashboard access

For applet or Web Start deployments, where dashboards communicate through an application server, your options include the following:

- Enabling HTTPS in the application server
- Utilizing a secure channel (SSH) for all dashboard communication
- Utilizing a virtual public network (VPN) for all dashboard access.

As with all encryption technology, there is a cost in performing the encryption and decryption of data. For applications with a very high frequency of data changes, you should account for this cost when you determine how frequently a dashboard can be updated (see the description of the `-u` or `--update` option).

Administering Dashboard Security

Example: Implementing LoginModule

Below is a sample implementation of `LoginModule`, which you can find under `samples\dashboard_studio\tutorial\src`. See "Authentication for local and WebSphere deployments" on page 348.

```
package com.apama.dashboard.sample;
import java.security.Principal;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;
import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.security.auth.login.FailedLoginException;
import javax.security.auth.login.LoginException;
import javax.security.auth.spi.LoginModule;
/**
 * SampleLoginModule is an example of a custom JAAS login module for
 * Dashboard Builder. Custom JAAS login modules allow you to extend Dashboard
 * Builder to use the authentication mechanism of your choosing.
 * <p>
 * SampleLoginModule authenticates all users, regardless of username
 * and password and adds the Principal "apama_customer" to each.
 *
 * $Copyright(c) 2013 Software AG, Darmstadt, Germany and/or its licensors$
 *
 * @version      $Id: SampleLoginModule.java 84623 2008-06-25 22:41:10Z cr $
 */
public class SampleLoginModule implements LoginModule {
  // Option strings
  private final static String OPT_DEBUG = "debug";
  // Initial state
  private Subject subject;
  private CallbackHandler callbackHandler;
  // True if debug logging turned on
  private boolean debug = false;
  // Authentication status
  private boolean succeeded = false;
  private boolean commitSucceeded = false;
  // Username and password
  private String username;
  private char[] password;
  /**
   * Initialize this LoginModule.
   *
   * @param subject Subject to be authenticated
   * @param callbackHandler CallbackHandler for communicating with the user to
   * obtain username and password
   * @param sharedState Shared LoginModule state
   * @param options Options specified in the login Configuration for this
   * LoginModule.
   */
  public void initialize(
    Subject subject, CallbackHandler callbackHandler, Map sharedState, Map options) {
    this.subject = subject;
    this.callbackHandler = callbackHandler;
    // Process options
    debug = "true".equalsIgnoreCase((String) options.get(OPT_DEBUG));

    // Add additional options and initialization here

    // Must have a callback handler
    if (callbackHandler == null) {
      throw new IllegalArgumentException (
        "Error. Callback handler must be specified.");
    }
  }
  /**
   * Authenticate the user by calling back for username and password.
   *
   * @return true in all cases
   * @exception FailedLoginException if the authentication fails
   * @exception LoginException if unable to perform the authentication
   */
  public boolean login() throws LoginException {
    // Callback to get username and password
    Callback[] callbacks = new Callback[2];
    callbacks[0] = new NameCallback("username: ");
    callbacks[1] = new PasswordCallback("password: ", false);
    try {
      // Perform the callbacks
      callbackHandler.handle(callbacks);

      // Get the user supplied name and password
      username = ((NameCallback) callbacks[0]).getName();
      password = ((PasswordCallback) callbacks[1]).getPassword();
      if (password == null) {
        // Treat a NULL password as an empty password
        password = new char[0];
      }

      // Clear password
      ((PasswordCallback) callbacks[1]).clearPassword();
    } catch (java.io.IOException e) {
      throw new LoginException("UserFileLoginModule. Error performing callbacks. " +
        e.toString());
    } catch (UnsupportedCallbackException e) {
      throw new LoginException("UserFileLoginModule. Error performing callbacks " +
                           e.getCallback().toString() + ".");
    }
    // verify the username/password
    if (validateUser()) {
      if (debug) {
        System.err.println("UserFileLoginModule. User authenticated: " + username);
      }
      succeeded = true;
      return true;
    } else {
      if (debug) {
        System.err.println("UserFileLoginModule. User authentication failed: " +
          username);
      }
      succeeded = false;
      clearState();
      throw new FailedLoginException("UserFileLoginModule. Login failed.");
    }
  }
```

```
/**
 * Called if the LoginContext's overall authentication succeeded
 * (the relevant REQUIRED, REQUISITE, SUFFICIENT and OPTIONAL
 * LoginModules succeeded).
 * <p>
 * Add the user's principals (roles) to the Subject
 *
 * @exception LoginException Commit failed
 * @return true if commit attempts succeeded; false otherwise.
 */
public boolean commit() throws LoginException {
  if (succeeded == false) {
    return false;
  } else {

    // Get the users roles from the user database and add each as a SimplePrincipal
    subject.getPrincipals().addAll(getUserPrincipals());
    clearState();
    commitSucceeded = true;
    return true;
  }
}
/**
 * Called if the LoginContext's overall authentication
 * failed. (the relevant REQUIRED, REQUISITE, SUFFICIENT and OPTIONAL
 * LoginModules did not succeed).
 * <p>
 * Cleans state information.
 *
 * @exception LoginException Abort failed
 * @return true if abort successfule; false otherwise
 */
public boolean abort() throws LoginException {
  if (succeeded == false) {
    return false;
  } else if (succeeded == true && commitSucceeded == false) {
    // login succeeded but overall authentication failed
    succeeded = false;
    clearState();
  } else {
    // overall authentication succeeded and commit succeeded,
    // but another LoginModule's commit failed
    logout();
  }
  return true;
}
/**
 * Logout the user.
 *
 * @return true in all cases
 * @exception LoginException Logout failed
 */
public boolean logout() throws LoginException {
  succeeded = false;
  succeeded = commitSucceeded;
  clearState();
  return true;
}
/**
 * Clear out temporary state used in a single login attempt.
 */
private void clearState () {
  username = null;
  if (password != null) {
    for (int i = 0; i < password.length; i++)
      password[i] = ' ';
    password = null;
  }
}
/**
```

```
   * Validate current username/password pair.
   *
   * @return true if validated.
   */
  private boolean validateUser () {

    //
    // Add user validation here.
    //
    System.out.println("Validate username: " + username + " password: " +
      new String(password));
    return true;
  }

  /**
   * Get the principals (roles) for the current username.
   *
   * @return Set of Principals
   */
  private Set<Principal> getUserPrincipals () {
    HashSet<Principal>  set = new HashSet<Principal>();
    //
    // Add user principals here.
    //

    System.out.println("Add principal username: " + username +
      " principal: apama_customer");
    set.add (new SamplePrincipal("apama_customer"));
    return set;
  }
}package com.apama.dashboard.sample;
import java.security.Principal;
/**
 * SamplePrincipal is an example of a java security principal (role) for
 * use with JAAS authentication.
 * <p>
 * SamplePrincipal provides a simple string based principal.
 *
 * $Copyright(c) 2013 Software AG, Darmstadt, Germany and/or its licensors$
 *
 * @version      $Id: SamplePrincipal.java 84623 2008-06-25 22:41:10Z cr $
 */
public class SamplePrincipal implements Principal {
  // Principal name
  private String name;
  /**
   * Constructor.
   *
   * @param name Principal name
   */
  public SamplePrincipal(String name) {
    this.name = name;
  }
  /**
   * Get the name of the principal.
   */
  public String getName() {
    return name;
  }
}
```

Administering Dashboard Security

15 Using the Apama Component Extended Configuration File

The Apama component extended configuration file is an optional file that you can use to do the following:

- Bind Apama server components to a particular set of addresses.
- Specify that Apama client connections must be from a particular IP address or range of IP addresses.
- Set environment variables for Apama components.
- Set log files and log levels for EPL root, packages, monitors, or events.

You can specify the optional extended configuration file when you start the correlator. If you do, the settings in the extended configuration file apply to the following Apama components:

- Correlator
- IAF
- Sentinel Agent
- `engine_receive` correlator utility

In an extended configuration file, if a line includes a special character that you want to be treated literally you must escape it by inserting a backslash just before it. A character that follows two consecutive backslashes (\\) is treated literally. Single quotes inside double quotes are treated literally. Double quotes inside single quotes are treated literally. See the example in "Sample extended configuration file" on page 366.

Binding server components to particular addresses

To bind Apama server components to a particular address or set of addresses, specify a `BindAddress` line for each address. Specify this in the `[Server]` section of the extended configuration file. For example:

```
[Server]
BindAddress=127.0.0.1:15903
BindAddress=10.0.0.1
```

You can specify as many `BindAddress` lines as you want. Clients can connect to any of the listed addresses.

An IP address is required. If you do not specify a port, the Apama server components use the port that is specified when the correlator is started. This makes it possible to share extended configuration files if you want to restrict connections according to only addresses.

If you do not specify an extended configuration file when you start the correlator, or there are no `BindAddress` entries in the extended configuration file, the Apama components bind to the wildcard address (`0.0.0.0`).

Using the Apama Component Extended Configuration File

Ensuring that client connections are from particular addresses

To ensure that client connections are from particular addresses, add one or more `AllowClient` entries to the extended configuration file in the `[Server]` section. For example:

```
[Server]
AllowClient=127.0.0.1
AllowClient=192.168.128.0/17
```

An `AllowClient` entry takes an IP address, as in the first example above, or a CIDR (Classless Inter-Domain Routing) address range, as in the second example above. With these example entries in the extended configuration file, the Apama components allow connections from either the localhost (`127.0.0.1`) or IP addresses where the first 17 bits match the first 17 bits of `192.168.128.0`. The Apama components to not accept connections from any other IP addresses. This creates a “whitelist” of allowable IP addresses.

If you specify an extended configuration file when you start the correlator, and if there are any `AllowClient` entries in the extended configuration file then the Apama components do not allow connections from any IP address that does not fall within one of the `AllowClient` ranges specified. If you do not specify an extended configuration file when you start the correlator, or there are no `AllowClient` entries in an extended configuration file that you do specify, the Apama components accept connections from any client.

This feature is intended to prevent mistakenly connecting to the wrong server. It is not intended to prevent malicious intruders since it provides no protection against address spoofing.

Using the Apama Component Extended Configuration File

Setting environment variables for Apama components

You can use the extended configuration file to set environment variables. Put environment variable declarations in the `[Environment]` section. For example:

```
[Environment]
LD_LIBRARY_PATH=/usr/local/mydir
```

If you specify an extended configuration file when you start the correlator, and if there are any environment variable entries in the extended configuration file then the Apama components use those environment variable settings. If you do not specify an extended configuration file when you start the correlator, or there are no environment variable entries in an extended configuration file that you do specify, the Apama components use the environment variable settings specified elsewhere.

Note: You must set variables such as `LD_PRELOAD` and `LD_LIBRARY_PATH` before the affected component starts execution. These environment variables are read-only. If you change them after component execution starts, the component does not recognize the change.

Using the Apama Component Extended Configuration File

Setting log files and log levels in an extended configuration file

To set log files and/or log levels for EPL root, packages, monitors, or events, specify entries in the `[ApplicationLogging]` section of the extended configuration file.

To set the default log file and level for the EPL root package, specify two lines in the following format:

```
RootFile=rootLogFilename
RootLevel=ROOTLOGLEVEL
```

Replace *rootLogFilename* with the name of the log file for the EPL root package. No spaces are allowed in the log file name. Replace *ROOTLOGLEVEL* with TRACE, DEBUG, INFO, WARN, ERROR, FATAL, CRIT or OFF. For example:

```
RootFile=apama\\root.log
RootLevel=ERROR
```

Note that you must insert a backslash to escape special characters. In the previous example, there is a filepath that contains a backslash that must be escaped. Hence, the double backslash. You do not need to specify both a log file and a log level; you can specify one or the other. If you do not specify a log file or log level for the root package, it defaults to the correlator’s log file/log level.

To set the default log file and level for an EPL package, monitor, or event, specify two lines in the following format:

```
PackageFile.node=nodeLogFilename
PackageLevel.node=NODELOGLEVEL
```

Replace *node* with the name of the EPL package, monitor, or event you are setting the logging attribute for. If a monitor or event is in a named package and not the root package, be sure to specify the fully qualified name.

Replace *nodeLogFilename* with the name of the default log file for the specified EPL package, monitor or event. No spaces are allowed in the log file name. Replace *NODELOGLEVEL* with TRACE, DEBUG, INFO, WARN, ERROR, FATAL, CRIT or OFF. This is the default log level for the specified node.

For example:

```
PackageFile.com.myCompany.Client=apama\\Client.log
PackageLevel.com.myCompany.Client=DEBUG
```

For a particular node, you do not need to specify both a log file and a log level; you can specify one or the other. If you do not specify a log file or log level for a particular node, it defaults to the settings for a parent node. See "Tree structure of packages, monitors, and events" on page 145.

There is no limit to the number of log settings that can appear in the configuration file. The last line that refers to a given node takes priority over earlier lines. When you set log attributes in the extended configuration file the rules for hierarchical logging apply. See "Setting logging attributes for packages, monitors and events" on page 145.

If you pass an extended configuration file to a correlator when you start that correlator and the configuration file contains an [ApplicationLogging] section, the correlator uses the logging settings in that section. If you do not pass a configuration file when you start a correlator, or there are no settings in the [ApplicationLogging] section, then correlator initialization does not include any log settings except for the default correlator log.

Whether or not you specify an extended configuration file when you start a correlator, any log settings you specify can be overwritten after initialization by invoking the engine_management utility and specifying the setApplicationLogFile and/or setApplicationLogLevel keywords. See "Managing application log levels" on page 146 and "Managing application log files" on page 147.

Using the Apama Component Extended Configuration File

Sample extended configuration file

Save the extended configuration file as a text file. Blank lines are ignored. For example, the contents of ApamaExtendedConfig.txt might be as follows:

```
[Server]
```

```
BindAddress=127.0.0.1:15903
BindAddress=10.0.0.1
AllowClient=127.0.0.1
AllowClient=10.0.0.0/24
[Environment]
LD_LIBRARY_PATH=/usr/local/mydir
[ApplicationLogging]
PackageFile.com.myCompany=apama\\apama.log
PackageLevel.com.myCompany=WARN
PackageLevel.com.myCompany.Client=DEBUG
```

Using the Apama Component Extended Configuration File

Starting a correlator with an extended configuration file

To use an Apama component extended configuration file, specify the `-Xconfig` option with the extended configuration file path when you start the correlator. For example, if both the license file and the extended configuration file are in the same directory as the correlator executable, and the name of the extended configuration file is `ApamaExtendedConfig.txt`:

```
run correlator -l license.txt -Xconfig ApamaExtendedConfig.txt
```

Using the Apama Component Extended Configuration File