Deploying and Managing Apama Applications

Version 10.15.5

June 2024

APAMA

This document applies to Apama 10.15.5 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: PAM-DEP-10155-20240624

Table of Contents

About this Guide

Deploying and Managing Apama Applications describes how to deploy Apama applications using Docker and Kubernetes. It also provides information for improving Apama application performance by using multiple correlators, for managing and monitoring Apama components over REST (Representational State Transfer), and for using correlator utilities and configuration files.

Documentation roadmap

Apama provides documentation in the following formats:

- HTML (available from both the documentation website and the `doc` folder of the Apama installation)
- PDF (available from the documentation website)
- Eclipse help (accessible from Software AG Designer)

You can access the HTML documentation on your machine after Apama has been installed: Display the `index.html` file, which is in the `doc/apama-onlinehelp` directory of your Apama installation directory.

The following guides are available:

Title	Description
Release Notes	Describes new features and changes introduced with the current Apama release as well as earlier releases.
Installing Apama	Summarizes all important installation information.
Introduction to Apama	Provides a high-level overview of Apama, describes the Apama architecture, discusses Apama concepts and introduces Software AG Designer, which is the main development tool for Apama.
Using Apama with Software AG Designer	Explains how to develop Apama applications in Software AG Designer, which is an Eclipse-based integrated development environment.
Developing Apama Applications	Describes the the technology for developing Apama applications: EPL monitors. You can use this technology to implement a single Apama application. In addition, there are C++ and Java APIs for developing components that plug in to a correlator. You can use these components from EPL.
Connecting Apama Applications to External Components	Describes how to connect Apama applications to any event data source, database, messaging infrastructure, or application.
Building and Using Apama Dashboards	Deprecated. Describes how to build and use an Apama dashboard, which provides the ability to view and interact with DataViews. An Apama project typically uses one or more dashboards, which are created in the Dashboard Builder. The Dashboard Viewer provides the ability to use dashboards created in the Dashboard Builder. Dashboards can also be

Title	Description
	deployed as simple web pages. Deployed dashboards connect to one or more correlators by means of a dashboard data server or display server.
Deploying and Managing Apama Applications	Describes how to deploy Apama applications using Docker and Kubernetes. It also provides information for improving Apama application performance by using multiple correlators, for managing and monitoring Apama components over REST (Representational State Transfer), and for using correlator utilities and configuration files.

In addition to the above guides, Apama also provides the following API reference information:

- API Reference for EPL (ApamaDoc)
- API Reference for Java (Javadoc)
- API Reference for C++ (Doxygen)
- API Reference for .NET
- API Reference for Python
- API Reference for Component Management REST APIs

Online Information and Support

Product Documentation

You can find the product documentation on our documentation website at https://documentation.softwareag.com.

In addition, you can also access the cloud product documentation via https://www.softwareag.cloud. Navigate to the desired product and then, depending on your solution, go to "Developer Center", "User Center" or "Documentation".

Product Training

You can find helpful product training material on our Learning Portal at https://learn.softwareag.com.

Tech Community

You can collaborate with Software AG experts on our Tech Community website at https://techcommunity.softwareag.com. From here you can, for example:

- Browse through our vast knowledge base.
- Ask questions and find answers in our discussion forums.

- Get the latest Software AG news and announcements.
- Explore our communities.
- Go to our public GitHub and Docker repositories at https://github.com/softwareag and https://containers.softwareag.com/products and discover additional Software AG resources.

Product Support

Support for Software AG products is provided to licensed customers via our Empower Portal at https://empower.softwareag.com. Many services on this portal require that you have an account. If you do not yet have one, you can request it at https://empower.softwareag.com/register. Once you have an account, you can, for example:

- Download products, updates and fixes.
- Search the Knowledge Center for technical information and tips.
- Subscribe to early warnings and critical alerts.
- Open and update support incidents.
- Add product feature requests.

Data Protection

Software AG products provide functionality with respect to processing of personal data according to the EU General Data Protection Regulation (GDPR). Where applicable, appropriate steps are documented in the respective administration documentation.

1 Security Requirements for Apama

Security model

The Apama security model for correlator and IAF components is that untrusted users must not be given access to any related files or to send or receive data on any of the correlator or IAF network ports. For dashboards, the same applies to the display server's management and data ports, and the data server's management port - although the data server's data port might need to be exposed to end-users if the thick Dashboard Viewer client is used. It is assumed that any user able to access these files or ports is trusted and has permission to make arbitrary changes and read arbitrary data. Such users are also permitted to inject arbitrary code into a correlator and execute it with the permissions of the correlator process.

Security requirements

You must use standard operating system and network tools/configuration to restrict access to the IAF and correlator components to only trusted users. For the dashboard servers, this applies except to the data server's data port.

You must use standard operating system tools to restrict access to all configuration and data files to only trusted users.

You must restrict access to changing the process environment when starting the server processes to only trusted users.

You must not set the correlator, IAF or dashboard server logging to a level higher than `INFO` to have all security-relevant events logged to the log files.

Setting the correlator, IAF or dashboard server logging to a level lower than `INFO` could include security-sensitive information in the log files.

Remember to also fully configure all connected systems to perform adequate authentication and authorization. Select connectivity plug-ins that support authentication and authorization where possible, and make sure these settings are enabled. For example:

- If using the Universal Messaging connectivity plug-in, set appropriate permissions on all channels, and use authentication and a certification authority. For more information, see "Configuring the connection to Universal Messaging (dynamicChainManagers)" in *Connecting Apama Applications to External Components*.

- If using the HTTP server connectivity plug-in, note that it exposes an additional port on the correlator. If deployed in any context where access is not restricted to only trusted users, the HTTP server connectivity plug-in must be configured to use TLS and HTTP basic authentication. It should not be directly connected to the internet. If internet access is required, then the plug-in must be deployed in a DMZ behind a reverse proxy such as Apache or Nginx. For more information, see "Configuring the HTTP server transport" in *Connecting Apama Applications to External Components*.
- If using the HTTP client connectivity plug-in, configure it to use TLS and HTTP basic authentication. For more information, see "Configuring the HTTP client transport" in *Connecting Apama Applications to External Components*.
- If using the MQTT connectivity plug-in, configure it to use SSL/TLS. For more information, see "Configuring the connection to MQTT" in *Connecting Apama Applications to External Components*.
- If using the Kafka connectivity plug-in, configure the Kafka clients to use SSL. For more information, see "Configuring the connection to Kafka (dynamicChainManagers)" in *Connecting Apama Applications to External Components* and the Kafka documentation at https://kafka.apache.org/.

If components must connect across an untrusted network, then either use a standard overlay tool such as a VPN or use a plug-in that supports TLS and authentication.

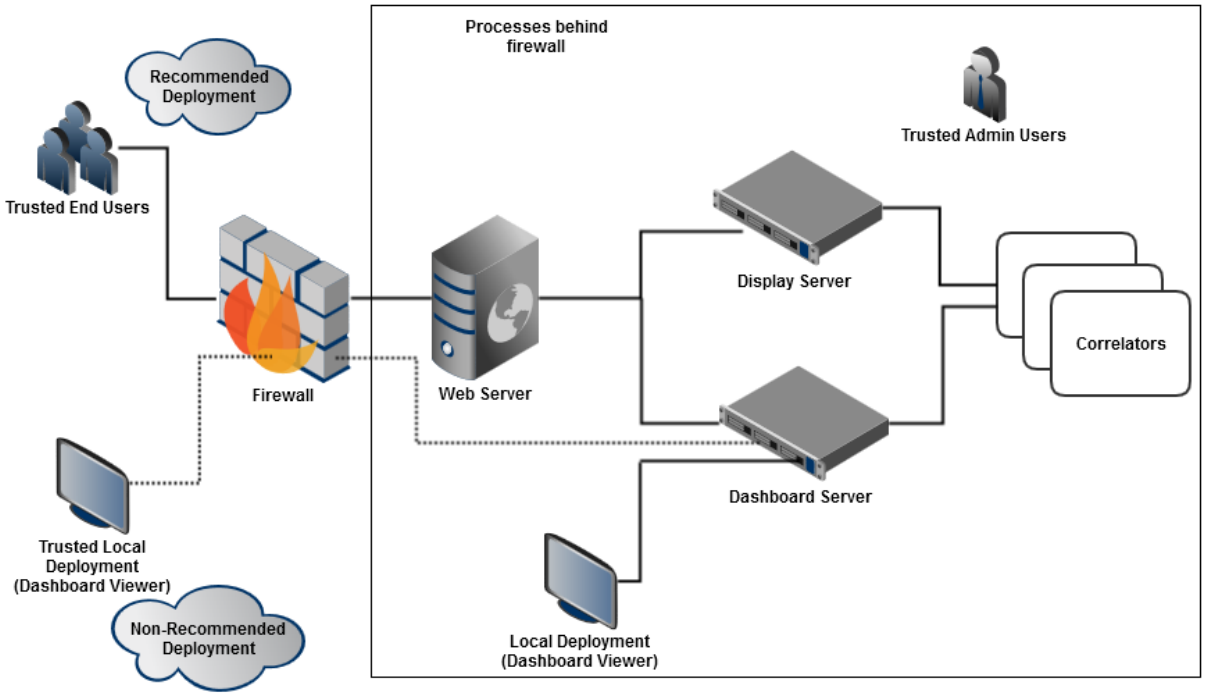
Important:

Ensure that you regularly install the latest Apama fixes, and keep your operating system fully patched to ensure the latest security fixes are present.

Dashboards

For access from untrusted hosts, you should deploy your dashboards to the web server using the display server deployment option. The dashboard server or display server processes should be running behind a firewall, just like the correlators. When accessing the dashboard server by using a standalone Dashboard Viewer outside the firewall, make sure to run the dashboard server with the `--ssl` option, which will ensure secure sockets for client communication using the data port. You must also ensure that the management ports of both servers are not exposed to end users.

The following diagram depicts the recommended dashboard deployment options:



Authentication for the display server is done via JAAS and your authentication mechanism of choice.

Important:

You must customize your own JAAS modules. The out-of-the-box authentication/authorization modules for dashboards cannot be used. This means that there is no authentication by default, and the basic authorization mechanism is shipped. For details on how to configure your own JAAS modules, see "Administering Dashboard Security" in *Building and Using Apama Dashboards*.

For a dashboard audit trail, you must load the Dashboard Support bundle into the correlator processing audit events and handle the `DashboardClientConnected` and `DashboardClientDisconnected` events, logging them in an appropriate manner.

Docker and Kubernetes

Both Docker and Kubernetes offer methods of passing secrets to the correlator. This can be used to securely provide credentials to the correlator. For more information, see the corresponding samples that are mentioned in "Apama samples for Docker" on page 42 and "Apama samples for Kubernetes" on page 51.

Personal data

You need to secure log files, input logs and the persistence database, especially if they contain personal data. On Windows, this would mean setting an inheritable Access Control List (ACL) limiting read access to the contained files. On UNIX systems, this would involve restricting read and execute permissions to only the owning user (that is, "700") and if possible also setting a `umask`

of 0077 on the correlator process to ensure files created by the correlator also have locked down permissions.

See "Protecting Personal Data in Apama Applications" in *Developing Apama Applications* for suggestions to help with identifying how personal data can be protected when building applications on the Apama platform.

2 Overview of Deploying Apama Applications

About deploying Apama applications with an Ant script

The recommended approach for script-based deployment of Apama is to use an Ant script, making use of the Ant macro definitions provided in the `apama-macros.xml` file. You can find this file in the `etc` directory of your Apama installation. See the comments in that file for more detailed information about the available macros.

Software AG Designer has an Ant export wizard that can generate a simple Ant script for deploying your Apama project. See "Exporting to a deployment script" in *Using Apama with Software AG Designer* for more information.

Instead of using Ant to inject the EPL, you can also do this using a YAML configuration file (see "Deploying Apama applications with a YAML configuration file" on page 118). If a YAML configuration file is used for this purpose, then Ant should not inject the EPL.

About deploying Apama applications with Docker

You can use Docker to deploy your Apama applications. See "Deploying Apama Applications with Docker" on page 31 for detailed information.

About deploying Apama applications with Kubernetes

You can use Kubernetes for container orchestration, that is, for automating the deployment of your Apama applications. See "Deploying Apama Applications with Kubernetes" on page 47 for detailed information.

About 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 correlators, see "Correlator Utilities Reference" on page 77.

About deploying dashboards

Dashboard deployment and administration involves the following activities:

- Deployment package installation and configuration. See "Deploying Dashboards" in *Building and Using Apama Dashboards*.
- Data server and display server management. See "Managing the dashboard data server and display server" in *Building and Using Apama Dashboards*.
- Security administration. See "Administering Dashboard Security" in *Building and Using Apama Dashboards*.

Before you perform these tasks, you should familiarize yourself with the deployment and administration concepts described in "Dashboard Deployment Concepts" in *Building and Using Apama Dashboards*.

Deployment options

Dashboards can be deployed as simple thin-client web pages 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" in *Building and Using Apama Dashboards*.

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 DataViews. The data server and display server are introduced in "Data server and display server" in *Building and Using Apama Dashboards*.

Process architecture

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" in *Building and Using Apama Dashboards*.

Builders and administrators

Dashboard deployment involves the use of a dashboard deployment package. 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" in *Building and Using Apama Dashboards*.

About 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 53. The section also contains information about preserving a correlator's runtime state.

Setting up the environment using the Apama Command Prompt

Before you can run any of the Apama tools (such as `engine_send` or `engine_inject`) or any of the Apama servers (such as the correlator or the Integration Adapter Framework which is also known as the IAF) from a normal command prompt, you have to set up your environment correctly. This includes setting the paths to the Apama installation directory, the Apama work directory, the location of the libraries, the Java location, and other environment variables. Apama provides a batch file (Windows) or shell script (UNIX) for this purpose, which is called the "Apama Command Prompt".

- **Windows**

  From the Windows command prompt, run the file `apama_env.bat` which is located in the `bin` directory of your Apama installation.

- **UNIX**

  Invoke the shell script from a Bash shell. Please note csh (C Shell) is not supported. Use the following command from within your Apama installation directory:

  ```
  source bin/apama_env
  ```

  It is important that you use `source` because invoking `apama_env` directly will not work.

  You can add the above command to your shell initialization script (which is `.bashrc` in the case of the Bash shell). If you do so, every shell you use will be an Apama Command Prompt.

Running a single Apama command

Alternatively, you can just run a single Apama command on either Windows or UNIX by using the `apama_env` script as a wrapper. This will not change the environment for your shell and you have to use the `apama_env` wrapper each time you run a command. This is particularly useful if you are invoking Apama commands from another program which has not had the environment set up.

- **On Windows**, you can see the syntax with:

  ```
  apama_env.bat /?
  ```

  To run a single Apama command, provide the command name and arguments after the file name, for example:

  ```
  apama_env.bat correlator --config myproject
  ```

- **On UNIX**, you can see the syntax with:

  ```
  apama_env --help
  ```

  To run a single Apama command, provide the command name and arguments after the script name, for example:

  ```
  apama_env correlator --config myproject
  ```

Using an alternative CA bundle

You can use your own SSL certificate within the Apama Command Prompt. This will affect all processes within the prompt. If you do not specify an alternative certificate file, the default Apama SSL certificate is used.

To use your own SSL certificate, set the environment variable `SSL_CERT_FILE` to the file path of your certificate *before* starting the Apama command prompt.

- **On Windows**, set the environment variable as follows:

  ```
  set SSL_CERT_FILE=path-to-certificate-file
  ```

- **On UNIX**, set the environment variable as follows:

  ```
  export SSL_CERT_FILE=path-to-certificate-file
  ```

If you explicitly wish to not use a CA bundle, set `SSL_CERT_FILE=none`.

3 Deploying and Managing Queries

Note:
Apama queries are deprecated and will be removed in a future release.

As mentioned elsewhere, scaling, both vertically (same machine) and horizontally (across multiple machines), is inherent in Apama query applications. Scaled deployments on multiple machines use distributed cache technology to maintain and share application state. Consequently, deployment of Apama query applications includes setting up a distributed cache (the so-called distributed MemoryStore) as well as some kind of messaging. The topics in this section provide instructions for doing this with the recommended platforms.

Overview of deploying and managing query applications

Typically, query application deployments script the start up and management of all Apama query application components outside of the Apama development environment in Software AG Designer. Apama recommends the use of the The Ant export facility of Software AG Designer to aid in this.

Queries can also be run from Software AG Designer. However, Software AG Designer can run only a single correlator deployment. To run multiple correlator deployments, use Apama macros for Ant.

Queries can be deployed on a single node, but typically would be deployed across multiple nodes, forming a cluster. While involving more components, a cluster provides:

- Scale out across multiple hosts.
- Resiliency against failures.
- Continued availability if some nodes fail.

Using a cluster will involve the following:

- Some number of correlators that are executing queries.
- A distributed MemoryStore for storing event history. Terracotta's TCStore is the recommended MemoryStore driver for Apama queries.

  Note:
  Support for using BigMemory Max for Apama queries is deprecated and will be removed in a future release.

- A JMS bus for distributing events to correlators.

Query application architecture

In a query deployment, incoming events are delivered to correlators, typically via a JMS message bus, such that every event is delivered to one correlator. The correlators store the event history for each query in the distributed MemoryStore. On every event, one correlator reads the latest history for the partition or partitions to which the event belongs, and writes that event to the distributed MemoryStore for access by other correlators. The entire window history is then evaluated against the query patterns.

Queries can make use of the following technologies to provide a scalable platform:

- JMS queues — these are used to distribute events to multiple correlators, which automatically spreads the load across a number of servers.
- TCStore, which is the recommended distributed MemoryStore — this allows state (event history) to be accessed quickly across multiple servers, and replicated to safeguard against hardware failures. This should be configured to give the desired amount of resiliency and scaled appropriately to the deployment.

It is possible to use Apama queries in a standalone mode on a single correlator. This allows easy testing by means of event files. However, all state is stored in-memory, and is lost when the correlator is stopped. Thus, this mode is only recommended for development, not for deployments.

When an event is sent to a cluster of correlators over a JMS queue, the following happens:

1. Each event goes to one correlator.
2. A received event is handled by one of several processing threads within that correlator.
3. The key of the event is extracted based on the definitions of running queries that use that event.
4. The window of events for that key value is retrieved from the distributed MemoryStore.
5. The current event is added to the retrieved window, which is written back to the distributed MemoryStore.
6. The event pattern of interest (what you are looking for) is evaluated against the stored window to determine whether there is a match.

Because events are sent to multiple threads in different correlators, small differences in timing across hosts can result in events being processed out of order. If there is a large number of events in the window, the cost of reading and writing the historic window will be excessive. Events for the same key may be processed by different correlators. Consequently, between events, the only state kept by the system is the window of historic event data.

Upon matching an event pattern, queries may send events to other monitors or to adapters. These can be shared adapters across the cluster, or more typically, adapters local to each correlator.

Deploying query applications

Apama recommends that you use the Ant export facility in Software AG Designer to help you deploy your query application. The general steps for deploying an Apama query application include:

1. In Software AG Designer, enable JMS support and distributed MemoryStore support. See "Correlator arguments" in *Using Apama with Software AG Designer*.
2. In Software AG Designer, generate an Ant deployment script. The generated files are placed in a directory that you specify. See "Exporting to a deployment script" in *Using Apama with Software AG Designer*.
3. Copy the resultant directory onto each host that will run a correlator.

4. If necessary, edit the `environment.properties` file on each correlator host.
5. Ensure that the distributed MemoryStore and JMS servers are running.
6. On each correlator host, run the Ant deployment script to start the correlator.

If the project does not contain a distributed MemoryStore configuration, a local in-process MemoryStore will be used to store events. This is not shared or persistent, so only supports a single correlator deployment. If this correlator stops, it will drop all event history data. Apama recommends Terracotta's TCStore and a corresponding configuration for production use. See "Deploying a Terracotta Server Array (TSA)" on page 23 and "Configuring the TCStore driver" on page 23.

Apama does not recommend running multiple correlators on a single machine. The assumption is that each correlator can use all of the CPU resources available. Also, running multiple correlators on one host does not provide any extra resilience. However, it is possible to run multiple correlators on a single machine. To do so:

1. Copy the exported deployment directory to separate directories on the correlator host machine.
2. Edit the `environment.properties` file to specify a different port number for each correlator and for each (if any) adapter in your project.

Running queries on correlator clusters

The following topics describe how to run queries on correlator clusters.

Deploying queries on multiple correlators

When using multiple correlators to deploy an Apama query application, it is the administrator's responsibility to keep the resources of the exported project up to date. If changes are made to a query, if queries are added or removed from a project, then all correlators should be updated to reflect the new state. It is possible to inject queries into a live running correlator, or delete queries from a correlator. Make sure that the injections and deletions are performed on all correlators in the cluster. Use `engine_delete -F` *`query-name`* to delete a query (see also "Deleting code from a correlator" on page 131). Note that this will also delete any queries using that query's output event (see also "Using the output of another query as query input" in *Developing Apama Applications*).

The queries runtime assumes that all members of a cluster:

- Share access to the same distributed MemoryStore state - by using a TCStore or BigMemory Terracotta Server Array.

  Note:
  Terracotta's TCStore is the recommended MemoryStore driver for Apama queries. Support for using BigMemory Max for Apama queries is deprecated and will be removed in a future release.

- Can connect freely between nodes.

- Run with clocks synchronized to within 1 second of each other. Apama recommends the use of the Network Time Protocol (NTP) to synchronize clocks.

The queries runtime will nominate a single member of the cluster to be primary, which will handle book keeping tasks such as garbage collecting nodes or handling failed cluster nodes.

If a correlator member of a cluster is using external clocking, then some functionality may not be available. The members will be able to share the same data, but an externally clocked node cannot be the primary node and timers will not be failed over from an externally clocked node. In normal operation, external clocking should only be used for testing purposes on a single node (where failover and scalability is not required).

A production deployment of multiple nodes would not use external clocking for routine processing of events. Use the source timestamp feature if the events may be delayed or delivered out of order. For more information, see "Using source timestamps of events" in *Developing Apama Applications*.

Deploying a Terracotta Server Array (TSA)

To deploy a TCStore Terracotta Server Array, see the Terracotta documentation. To deploy a BigMemory Terracotta Server Array, see the BigMemory Max documentation which is part of the Terracotta 4 documentation. Both documentation sets are available from http://documentation.softwareag.com/terracotta/index.htm.

For resilient operations, Apama recommends at least one backup on a separate host. You may want to consider using multiple stripes in order to improve performance. Ensure that the BigMemory Max or Terracotta server is accessible from all cluster members.

Note:
Terracotta's TCStore is the recommended MemoryStore driver for Apama queries. Support for using BigMemory Max for Apama queries is deprecated and will be removed in a future release.

Configuring the TCStore driver

If you want to use TCStore for queries, you need to configure the TCStore driver as described below.

To configure the TCStore driver

1. In Software AG Designer, add the **Distributed MemoryStore** adapter bundle to your Apama project. In the Distributed MemoryStore Configuration Wizard, select **Apama Queries (using TCStore)** as the store provider; **ApamaQueriesStore** is then automatically provided as the store name. See "Configuring a distributed store" in *Developing Apama Applications* for more detailed information.
2. Adapt the list of Terracotta servers in the `storeName-spring.xml` file. See "TCStore (Terracotta) driver details" in *Developing Apama Applications* for more information.

   Important:

You must leave the `useCompareAndSwap` property in its default (`true`) setting for correct behavior of Apama queries.

Configuring the BigMemory Max driver

Note:

Support for using BigMemory Max for queries is deprecated and will be removed in a future release. It is recommended that you now use Terracotta's TCStore for queries.

If you still want to use BigMemory Max for queries in new projects, you can add a BigMemory Max driver to the project as described below. Existing deployments using BigMemory Max for queries are unaffected; this only covers developing new projects in Software AG Designer. Keep in mind that you can no longer select the option **Apama Queries (using BigMemory)**. This option has been replaced by the **Apama Queries (using TCStore)** option.

To configure the BigMemory Max driver in a new project

1. In Software AG Designer, add the **Distributed MemoryStore** adapter bundle to your Apama project. In the Distributed MemoryStore Configuration Wizard, select **BigMemory Max** as the **Store provider** and specify "ApamaQueriesStore" as the store name. See "Configuring a distributed store" in *Developing Apama Applications* for more detailed information.

   Note:

   If you specify a different store name or do not specify a name at all, an in-process only memory store will be used.

2. Check that the cluster name is set correctly for the host/port pairs of all of the BigMemory Terracotta Server Array.
3. Set the `providerDir` property to the Terracotta installation directory.
4. Optionally, edit the on-heap and off-heap storage and other parameters as needed (see "BigMemory Max driver details" in *Developing Apama Applications*).

   Important:

   You must leave the `useCompareAndSwap` property in its default (`true`) setting for correct behavior of Apama queries.

Using JMS to deliver events to queries running on a cluster

When running queries across multiple correlators in a cluster, as well as configuring all correlators to access the same distributed MemoryStore, Apama recommends that all events are delivered into the cluster using a JMS queue. By using a JMS queue, each correlator will pull events from the JMS queue unless it has a full input queue (that is, it is behind on processing events) or has stopped running (for example, shut down for maintenance or suffered a hardware failure). In either case, events will continue to be processed by other correlators in the cluster. Correlators can

also be added to or removed from the cluster to scale the cluster capacity if desired. It is also possible to use per-correlator adapters for incoming events, but the adapters must co-ordinate so that every event is sent to only one correlator, and should one adapter/correlator pair fail, then other adapters process events that the failed node would have processed. Each event should only be delivered to one correlator, else multiple correlators will store the event in the shared cache, which can result in erroneous matches. Using JMS queues, this happens automatically, giving an "elastic" system that can be scaled and continues running in the face of failure.

To run queries across multiple correlators in a cluster:

- Configure each correlator to access the same distributed MemoryStore. This is a requirement.
- Use a JMS queue to deliver events into the cluster. This is a recommendation.

When the cluster uses a JMS queue, each correlator pulls events from the queue. If the input queue of one correlator in the cluster becomes full and it cannot pull events from the JMS queue, the other correlators continue to do so and continue to process events. A correlator may stop pulling events because the correlator is behind on processing events or because it has stopped running, perhaps for maintenance or because of a hardware failure.

Using a JMS queue makes it easy to scale the cluster capacity by adding or removing correlators.

An alternative to using a JMS queue is to use an adapter for each correlator. For example, by having an IAF-based adapter connected to each correlator, it is possible to send messages to and from a query application without using JMS. A disadvantage of using per-correlator adapters is that the adapters must coordinate the following:

- Each event goes to only one correlator in the cluster. If an event goes to more than one correlator, then multiple correlators store the same event in the shared cache. This can result in erroneous matches.
- Should one adapter/correlator pair fail, then the other adapters process the events that the failed node would have processed.

Use of a JMS queue automatically ensures that an event goes to only one correlator and that all received events are processed. The result is an "elastic" system that can be scaled and that continues to run even if a node fails.

Similar to using multiple contexts in a correlator, delivering events through JMS can result in events that occur close together in time being processed in an order that is different than the order in which they were created or sent to the JMS message bus.

Messages may be lost in the event of node failure, unless you have configured JMS for reliable message delivery (see also "Handling node failure and failover" on page 26).

Configure your JMS bus to have one or more queues, and configure a static JMS receiver connection. See "Getting started with simple correlator-integrated messaging for JMS" in *Connecting Apama Applications to External Components*. You will also need to provide mapping for all event types that flow into the queries. See "Mapping Apama events and JMS messages" also in *Connecting Apama Applications to External Components*.

The queries runtime ensures that after all queries have been injected into the correlator and started, they automatically start to receive events from JMS queues. There is no need to explicitly call

`jms.onApplicationInitialized()` as described in "Using EPL to send and receive JMS messages" in *Connecting Apama Applications to External Components*.

For all applications that do not consist entirely of queries, for example, applications that contain additional EPL monitors or Java monitors, then it may be required to delay starting JMS until the application and queries are both ready to process events. The auto-starting of JMS behavior of queries can be controlled by sending a `QueriesShouldNotAutoStartJMS()` event to the main context. This event can be routed by an application's `onload()` method. If this is done, then a monitor in the main context should listen for a `QueriesStarted()` event and should wait until both the application and queries have started. The monitor can then call `jms.onApplicationInitialized()` directly. For example, the following monitor delays starting JMS until queries are started and a `StartMyApp()` event has been processed:

```
using com.apama.queries.QueriesShouldNotAutoStartJMS;
using com.apama.queries.QueriesStarted;
event StartMyApp {
}
monitor MyApp {
   import "JMSPlugin" as jms;
   action onload() {
      route QueriesShouldNotAutoStartJMS();
      on QueriesStarted() and StartMyApp() {
         jms.onApplicationInitialized();
      }
   }
}
```

Mixing queries with monitors

It is possible to have both monitors and queries in a project.

Events that are to be processed by queries should be sent to the `com.apama.queries` channel from monitors. Queries may send events to any channel which EPL monitors may be subscribed to.

While queries will automatically scale and share state across a cluster, EPL monitors will not. Thus, be aware that a query may process subsequent events matching a pattern on different nodes. On different nodes, monitors with potentially different state will be executing. Similarly, the state of EPL monitors is not automatically stored in the distributed MemoryStore.

Both EPL monitors and Apama queries can make use of actions defined on events, subject to some limitations on the use of `spawn`, `die`, and event listeners. See "Restrictions in queries" in *Developing Apama Applications*.

Handling node failure and failover

A node may stop processing events from time to time. This may be because it is stopped for planned maintenance, or the node failed in some way. In these cases:

- Events that have been delivered to the node but not yet processed will be lost. This will typically be a small window of events.

This does not apply if you are sending and receiving events via JMS where you have configured JMS for reliable messaging. See "Avoiding message loss with JMS" on page 27 for more information.

- If using JMS, then events continue to be delivered to and processed by other correlators in the cluster. The failed correlator will not hold up processing on other nodes. Other nodes continue processing events, including matching against events that the failed node had previously received (if they had been processed).
- Any clients connected to the failed correlator will need to re-connect to another correlator. The same set of parameterized query instances is kept in synchronization across the cluster. See "Managing parameterized query instances" on page 28.

Similarly, nodes running a Terracotta Server Array may fail. For this reason, TCStore or BigMemory Max should be configured with sufficient backups to ensure no data is lost in this case. Consult the Terracotta documentation.

Avoiding message loss with JMS

If all of your incoming and outgoing events are received/sent via correlator-integrated JMS (see also "Correlator-Integrated Support for the Java Message Service (JMS)" in *Connecting Apama Applications to External Components*) and if this has been configured with `APP_CONTROLLED` receivers and `BEST_EFFORT` senders (see also "Sending and receiving reliably without correlator persistence" in *Connecting Apama Applications to External Components*), then no events are lost in the event of a node failure. Any events that have been delivered from JMS to queries on that node are then handled by another node if they had not been fully processed before the failure. Any events sent to JMS by queries on that node are delivered by another node if they had not been successfully delivered before the failure.

- This only works if the queries (or a chain of queries) are receiving events directly from JMS receivers and are sending their output directly to JMS senders. There are no guarantees if EPL monitors are processing query input or output, interposing themselves between the queries and JMS.
- No EPL monitors in the same correlator should be performing acknowledgments to `APP_CONTROLLED` receivers themselves, as those receivers are entirely under the control of the queries runtime.
- Incoming events may be delivered twice or be delivered out of order during the failover window. This is the time between the node failure and the cluster (including the JMS broker) detecting the failure/disconnection. It is your responsibility to make sure that your queries are not sensitive to duplicates or re-ordering within this failover window.
- Outgoing events may also be delivered in duplicate during the failover window.
- Queries using source timestamps (see "Using source timestamps of events" in *Developing Apama Applications*) cannot make use of JMS reliable messaging.

You should also ensure that the JMS broker does not lose messages in the case of a broker failure. Make sure that all JMS senders have their `messageDeliveryMode` property set to `PERSISTENT`, as well as doing any necessary broker-specific configuration on the broker itself.

Note:
Reliable messaging will not take effect unless your queries are exclusively using correlator-integrated JMS as their message source and destination. It does not apply when using connectivity plug-ins as your event source or destination (even if they support reliable messaging).

Managing parameterized query instances

When using parameterized queries, Apama recommends that you use one Scenario Service client at a time to manage parameterizations. Use of more than one client can result in undefined behavior if they both attempt to edit a parameterized instance concurrently. You can connect to any correlator in the cluster, and Apama will automatically synchronize the state of parameterized instances across the cluster. This assumes that the same query definitions have been injected into the correlators on all cluster nodes. If a node fails, you will need to connect to another correlator in the cluster.

Creating new query instances by setting parameter values

Use the Scenario Browser to set parameter values for a parameterized query and thus create new parameterized query instances, also referred to as parameterizations. See also "Using the Scenario Browser view" in *Using Apama with Software AG Designer*.

Changing parameter values for queries that are running

Use the Scenario Browser to change the parameter values for a running parameterized query instance, also referred to as a parameterization. See also "Using the Scenario Browser view" in *Using Apama with Software AG Designer*.

Monitoring running queries

To help you monitor queries that are running on a given correlator, Apama provides data about active queries in DataViews. To display the information provided by these DataViews, you can create a dashboard in which an end user can:

- Monitor query runtime performance.
- Determine whether a query is behaving as intended. For example, you can see how incoming events are distributed across partitions. If you are expecting a particular send and match rate, you can see if you are getting the results you expect.
- Ensure that the window size (the number of events in the window) is not too large. The expectation is that your application is designed so that partitioning keeps any given window size as small as possible.

The `Queries_Statistics_Sample` that is provided with Apama (located in the `\samples\queries` directory of your Apama installation) contains such a dashboard. It shows you how to build a dashboard that allows you to monitor the performance of running queries.

For information about exposing DataViews in dashboards, see "Building Dashboard Clients" in *Building and Using Apama Dashboards*.

A running query is either a non-parameterized query instance or a parameterization. For each running query, there is a DataView for each of its input event types. For example, if a query instance has two input event types, then there are two DataViews that provide statistics for that query, one for each input event type.

Each DataView:

- Contains data about the activity during the last second of one running query and one of its input event types.
- Contains the fields described in the table below. The value contained in each field is an exponentially weighted moving average (EWMA).
- Is updated every 10 seconds by default if the information has changed since the last update.

  By sending a `SetQueryStatisticsPeriod` event, you can control the frequency of the statistics gathering or disable query statistics entirely. For example, to update the query statistics every second:

  ```
  com.apama.queries.SetStatisticsUpdatePeriod(1,1)
  ```

  To disable query statistics entirely:

  ```
  com.apama.queries.SetStatisticsUpdatePeriod(0,0)
  ```

Statistical Field	Description
`% Threads Active EWMA`	Apama uses multiple threads to process a given query. This is the percentage of those threads that were used within the last second to process the input event type that this DataView provides information for. While there is not a linear correlation, as this percentage goes down, the reliability of the rest of the statistics becomes weaker. This is because a smaller proportion of threads are contributing information.
`Avg. Inbound Event Rate/s EWMA`	The average rate per second at which events of this type are being processed.
`Avg. % of Successful Matches EWMA`	The average percentage of the number of received events that cause a match.
`No. Unique Keys Processed EWMA`	The number of unique query partitions that were accessed for this event type within the past second.

Statistical Field	Description
`Avg. Window Size/Key EWMA`	The average window size (number of events that it contains) of each unique partition that was accessed within the past second.

The display name of these DataViews is **Correlator Query Statistics**.

After a non-parameterized query is injected into the correlator, Apama provides a DataView for each input event type and begins writing data to it. After a non-parameterized query is deleted, Apama no longer makes the DataViews for that query instance available.

For a parameterized query, after a parameterization is created, then Apama adds new DataViews and begins populating them. When a parameterization is deleted, then Apama no longer provides the DataViews that correspond to that parameterization. If the definition of a parameterized query is deleted, then Apama no longer provides DataViews for any parameterization of that query.

To help you monitor queries that are running across multiple correlators in a cluster, Apama also provides the same type of performance statistics provided for a given correlator but where the underlying data has been aggregated across all the clustered correlators running those queries.

The display name of these DataViews is **Cluster-Wide Query Statistics**.

This means that for each query running on a correlator, two types of monitoring data are provided:

- Statistics generated from data from only that correlator.
- Statistics generated from data aggregated across all correlators in the cluster running that same query.

4 Deploying Apama Applications with Docker

Introduction to Apama in Docker

Below is a brief overview of Docker, however, no familiarity with Docker commands is assumed in this part of the documentation. You need to install Docker before you can make use of the Apama images. See https://docs.docker.com/get-started/ for a more detailed overview of Docker and how to use it.

Images

Images are the base units of the Docker system, providing templates that are used to create the containers, which in turn form the running applications. Docker software runs on various operating systems and can run containers built from the Apama images.

Note:
Apama supports building images on Linux only.

Docker

Docker is a technology that allows an organization to remove the complexity of configuring and deploying software applications within the infrastructure it uses. The Docker platform achieves this by running the applications in what are called *containers*, which isolate the environment of an application from the deployment environment and provide tools for portability and scalability. As long as the organization infrastructure can run the Docker software, it can run the containers and consequently any applications contained in them. Use cases for Docker containers include modernization of legacy applications, migration to a cloud infrastructure, services, and continuous integration and deployment.

Docker Compose

Docker handles single containers and getting them built and deployed. It is best practice for a container to perform one role, but often systems or services are composed of multiple applications interacting with each other. Docker Compose is the method for handling multiple containers and their interdependencies. It is an application that marshals the build, deployment, and runtime characteristics of one or more containers using a configuration file to control the process.

Docker swarm

Docker allows clusters of machines running Docker to be grouped into *swarms*. Swarms allow the containers to be distributed and load-balanced seamlessly using the Docker command line.

Docker stack

A Docker *stack* deals with deployment, runtime characteristics, containers and their dependencies. This is called *orchestration*. A Docker stack is a group of interrelated services that share dependencies and can be orchestrated and scaled together. Similar to the swarm, a Docker command line provides a way to manage and interact with stacks.

Apama and Software AG images

Software AG has a presence on the Docker Hub image repository. Several images are available for Software AG products. The offering from Apama is an image which will run an instance of the correlator application. See https://hub.docker.com/search?q=softwareag%2F&type=image for the published images.

Alternatively, images for several products, including Apama, may be built from an installation using scripts included in the installation.

Published Apama container images

Software AG produces several different variations of the Apama product as Docker images, depending on your use case. Each Apama image has a corresponding *builder* image for use in multi-stage builds.

Image	Use
softwareag/apama-correlator	Image for the correlator, including support for Java and Python. Does not contain connectivity to other Software AG products.
softwareag/apama-correlator-minimal	Smallest image for the correlator, aimed at pure-Apama use cases. Does not contain Java or Python support, or connectivity to other Software AG products.
softwareag/apama-correlator-suite	Image for the correlator, containing support for Java, Python, and connectivity to the rest of the Software AG product suite and JMS.
softwareag/apama-cumulocity-jre	Base image for custom Cumulocity IoT microservices, containing the correlator and connectivity to Cumulocity IoT.
softwareag/apama-builder	Project build and test tools for deploying and testing projects to be used with the `apama-correlator` and `apama-minimal` images in a multi-stage Docker build.
softwareag/apama-builder-suite	Project build and test tools (including Apache Ant) for deploying and testing projects to be used with the `apama-correlator-suite` image in a multi-stage Docker build.
softwareag/apama-cumulocity-builder	Project build and test tools for deploying and testing projects to be used with the `apama-cumulocity-jre` image in a multi-stage Docker build.

For more details of the content of each image, see the descriptions on the appropriate Docker Hub page. You must ensure that you select the appropriate image containing the components you require.

For information on how to use the builder images in a multi-stage build, see "Building Apama projects during the Docker build" on page 39.

Licensing Apama in Docker

For on-premise installations of Apama, license files are typically added at installation time. However, the Docker image does not come with a license file by default. You have to provide it by yourself. There are several ways to accomplish this, depending on your deployment architecture and whether you may have different license files or may need to update them while the correlator is running.

The Docker server mounts the license file at runtime

The simplest method is to provide the license file on the Docker server when it launches correlator containers. To do this, you mount a copy of the license file into the correlator container at the standard location of `/apama_work/license/ApamaServerLicense.xml`. You can either do this from a local file or using a cluster-wide shared file system.

To do this with a simple `docker run` command from a node-local file system, run:

```
docker run \
  -v/path/to/ApamaServerLicense.xml:/apama_work/license/ApamaServerLicense.xml \
  imagename
```

Mounts can also be provided in Docker Compose, Docker Stack or Kubernetes orchestration files. See the documentation for your orchestrator for more details.

The Docker server provides the license file via a configuration object

Both Docker Stack and Kubernetes support creating configuration objects in the stack, whose contents can be provided as a file mounted into the container at runtime.

Using Docker Stack, you need to create a stack configuration file which defines configuration. For example:

```
configs:
   apama_license:
      file: ./ApamaServerLicense.xml
```

Then you need to load the configuration to a particular location in your container:

```
services:
   apama:
      image: imagename
      configs:
         - source: apama_license
           target: /apama_work/license/ApamaServerLicense.xml
```

The application image contains the license file

Alternatively, if you do not need to provide different license files at runtime (for example, while developing or testing), you can compile the license file into your application image. To do this, add a `COPY` line to your project Dockerfile which copies the license file to the standard location. For example:

```
COPY --chown=sagadmin:sagadmin ApamaServerLicense.xml \
  $APAMA_WORK/license/ApamaServerLicense.xml
```

If you ever need to change the license file, you will need to rebuild your application image with the new license file.

The project contains the license file

You can also store the license file inside your Software AG Designer project and load it via configuration, rather than having it picked up automatically by the correlator. This also has the disadvantage of not being able to have a different license file in production, but means that your license file is automatically available wherever you run the project, whether in Docker or not, even if you did not initially install Apama with the license file.

To do this, you need to add a section to the Apama configuration file in your project, with a relative path to the license file which has been added to your project. For example:

```
correlator:
   licenseFile: ${PARENT_DIR}/../ApamaServerLicense.xml
```

With this option, the license file and configuration will automatically be included in the application image using the default Dockerfile provided by Software AG Designer.

For more details, see "YAML configuration file for the correlator" on page 102.

Quick start to using an Apama image

The Apama image that we are using here is available on Docker Hub (https://hub.docker.com/r/softwareag/apama-correlator/). You must purchase it (free of charge) so that you can use it. After you have done this, proceed as described below.

1. Log in to Docker Hub using your credentials:

   ```
   docker login
   ```

2. Obtain the image from the repository so that it will be available to use:

   ```
   docker pull softwareag/apama-correlator:version
   ```

   where *version* is the current Apama version number (a two-digit number such as `10.15`).

   Note:

   If you logged in successfully but get an error during the `docker pull` command, it is likely that there is a spelling mistake or that the image you specified is missing.

3. List the images available to you:

   ```
   docker images
   ```

   You should see the image that you pulled in the output of the above command (keep in mind there may be many images if you are using a shared machine).

4. Create a container using the image and run it in Docker:

```
docker run -d --name container --rm softwareag/apama-correlator:version
```

The above command will “detach” after running the container. You can see it running using the following command:

```
docker ps
```

You can interact with the running image in many ways (see the Docker documentation for the appropriate commands), but we will now retrieve the logs from the running container and then stop the container.

5. To examine the log of the running container, enter the following command using the name that was set with the `--name container` argument of the `docker run` command.

   ```
   docker logs container
   ```

   When the above command is executed, the container identifier or the name which you can find in the output of the `docker ps` command is taken as a parameter.

6. You can now stop the container:

   ```
   docker stop container
   ```

   This also removes the container since it was started with the `--rm` option.

Apama image for Cumulocity IoT microservices

If you are building a microservice for use within Software AG's Cumulocity IoT, then we provide a smaller base image designed for use in that environment (https://hub.docker.com/r/softwareag/apama-cumulocity-jre). You can use the `softwareag/apama-cumulocity-jre`*version* image anywhere that you could use `apama-correlator`. Existing Dockerfiles which use `apama-correlator` but provide `APAMA_IMAGE` as a build argument can be built using `apama-cumulocity-jre` instead, using an argument to the `docker build` command:

```
docker build -t projectimage \
 --build-arg APAMA_IMAGE=softwareag/apama-cumulocity-jre:version projectdir
```

The base image for Cumulocity IoT only provides a JRE and not a full JDK. It does not have Python support built-in and it only provides connectivity with Cumulocity IoT and not any other Software AG product.

Running as the sagadmin user within a container

Software AG images are configured not to run processes as root by default within containers. This is standard container practice. All images create a user `sagadmin` with the user identifier and group identifier of `1724`. By default, all processes run in the container, and all `RUN` commands in Dockerfiles using these images as a base run with that user identifier.

`COPY` commands may need to specify writing as the `sagadmin` user via the `--chown` argument. It is recommended to continue using this user for all of your commands within Docker.

If you need to run as another user, you need to add `USER` statements to your Dockerfile or the appropriate options to your docker `run` command.

Building an Apama image from the current installation

As an alternative to using the image from Docker Hub, the Apama installation provides a Dockerfile which enables you to build an image. You can find it in the `samples/docker/image` directory of your Apama installation. See the `README.txt` file in the `samples/docker` directory for detailed instructions.

1. To build the image, run the following command from your Software AG installation directory (by default, this is `/opt/SoftwareAG`):

   ```
   docker build --tag registrytag -f ./Apama/samples/docker/image/Dockerfile .
   ```

   If you wish to publish the image to a registry, then *registrytag* should be of the following form:

   registryhost/organization/imagename:tag

   Note that the image will be stored locally, but it can be published to your registry as described below.

2. Enter the following command to publish the image:

   ```
   docker push registrytag
   ```

3. Enter the following commands to use the image:

   ```
   docker pull registrytag

   docker run -d --name container --rm registrytag

   docker logs container

   docker stop container
   ```

   You can now use the image in the same way as described in "Quick start to using an Apama image" on page 35, by referring to the registry name on the command line. You can also refer to it in the Dockerfile; see "Apama samples for Docker" on page 42 for examples of how to refer to images in Dockerfiles.

Deploying an Apama application in Docker

The examples below reference the image in Docker Hub. However, you can also use an image built from your Apama installation; the results will be the same.

The base image simply provides an empty running correlator, with the management port exposed. You can use the command line tools described in "Correlator Utilities Reference" on page 77 to inject EPL and manage the correlator, or you can connect to the running container and use the management commands within the container. The expected use case is the deployment of a user application in an image derived from the base image.

1. Typically, the first step is to create the Dockerfile that references the required image of the Apama correlator. The following is an example Dockerfile:

```
# Reference the Apama image at softwareag/apama-correlator:<TAG>.
# The tag refers to the version number of the Apama correlator.
FROM softwareag/apama-correlator:version

# Copy files from the local app directory to /app in the resulting image.
COPY --chown=sagadmin:sagadmin app/* /app/

# This is the command we run - it references the internal directory.
CMD ["correlator","--config","/app"]
```

The above example file produces an image that contains a root level directory called `/app` that has copies of local files in it. These files will be owned by the user the correlator runs as, called `sagadmin`.

2. When the correlator image runs, it is directed to read the configuration file `init.yaml` that is present within the directory. An example of this from the `Simple` sample application is shown below. It can be found in the `samples/docker/applications/Simple` directory of your Apama installation, along with the `HelloWorld.mon` file that it references.

   Example `init.yaml`:

   ```
   correlator:
       initialization:
           list:
               - ${PARENT_DIR}/HelloWorld.mon
   ```

   At correlator startup, the above configuration file injects the monitor file `HelloWorld.mon` from the same directory. The image build process will copy the `HelloWorld.mon` and `init.yaml` files into the image. Thus, the application image can run without outside dependencies.

3. You build the image using the `docker build` command. The following example assumes that the command is run from the directory containing the Dockerfile. See the Docker documentation for details on the available options.

   ```
   docker build --tag organization/application .
   ```

4. Once the image is built, it is stored locally. You can view it using the `docker images` command. Note that the image is not running at this point.

   ```
   docker images | grep application
   ```

5. To run the image as a container, enter the following command:

   ```
   docker run -d --rm --name container organization/application
   ```

   The options `-d` and `--rm` are used to detach and remove the container after shutdown.

6. To examine the running process, enter the following command:

   ```
   docker ps
   ```

7. To examine the logs, enter the following command:

   ```
   docker logs container
   ```

8. To stop the container, enter the following command:

```
docker stop container
```

Now the `docker ps` command should no longer show your container.

9. To remove the image, enter the following command:

```
docker rmi organization/application
```

This untags and deletes the image.

Note that the application is "baked" into the image you create. The files copied into the image will not change and any output will not persist outside the container. The management port is exposed, and therefore the correlator can be manipulated remotely through the Management interface (see also "Using the Management interface" in *Developing Apama Applications*). However, you can interact with the running container in many ways, including starting a shell, examining logs, and running commands in the container. See the Docker documentation for details on the available options.

Developing an Apama application using the Docker image

The Dockerfile that you created in "Deploying an Apama application in Docker" on page 37 fixes or bakes in the `app` directory and its contents. Thus, you would have to recreate the image every time the configuration file or the EPL files change, which is inconvenient during the development phase of the application where the EPL changes frequently. You can address that by mapping in a local directory when running the container and use that to hold your application. For example:

```
docker run -d --rm --name container -v /local/path:/app \
  softwareag/apama-correlator:version correlator --config /app
```

Note:

The above command is broken over two lines via the `\` escape character at the end of line. You can either copy this line verbatim or construct it to be a single line command.

Keep in mind that you are using the official image and not creating your own. Because you are mounting in a local directory containing the application, there is no need to build a custom image. It will be read from the */local/path* which is mounted in instead.

Now you can make changes to the EPL or `init.yaml` file and simply restart the container to pick up the changes. When you have completed your changes and want to deploy, you can add a Dockerfile to bake the files into the image again. It is possible to define and map multiple volumes, allowing flexible usage of the image.

Building Apama projects during the Docker build

Some Apama applications, particularly those developed using Software AG Designer or those with custom plug-ins, require additional build steps when creating a Docker image, such as running the `engine_deploy` tool (see also "Deploying a correlator" on page 126). For those using the sample packaging kit to create base images from an installation, this can be done directly in a standard Dockerfile. For those using the official Docker Hub image, a second builder image can be used for project build steps. This is in order to keep the runtime image as small as possible. The builder image can be used as part of a Docker multi-stage build.

A typical multi-stage Dockerfile looks like this:

```
FROM buildbase as builder

COPY source /source
RUN buildstep

FROM runtimebase

COPY --from=builder /buildoutput /buildoutput

CMD ["/buildoutput"]
```

For a typical Software AG Designer-based Apama project, your Dockerfile looks something like this:

```
ARG APAMA_VERSION=version
ARG APAMA_BUILDER=softwareag/apama-builder:${APAMA_VERSION}
ARG APAMA_IMAGE=softwareag/apama-correlator:${APAMA_VERSION}
FROM ${APAMA_BUILDER} as builder

COPY --chown=sagadmin:sagadmin MyProject ${APAMA_WORK}/MyProject
RUN engine_deploy --outputDeployDir ${APAMA_WORK}/MyProject_deployed \
${APAMA_WORK}/MyProject

FROM ${APAMA_IMAGE}

COPY --from=builder --chown=sagadmin:sagadmin ${APAMA_WORK}/MyProject_deployed \
${APAMA_WORK}/MyProject_deployed

WORKDIR ${APAMA_WORK}

CMD ["correlator", "--config", "MyProject_deployed"]
```

In Software AG Designer, you can add Docker support to your project as described in "Adding Docker support to Apama projects" in *Using Apama with Software AG Designer*. When you do this, a Dockerfile similar to the one above is automatically created in the project. Therefore, a project with Docker support can be built into an image using the following command:

```
docker build MyProject
```

For most projects, the provided Dockerfile will be sufficient. If you have additional build steps (such as building custom plug-ins), you can add them to the Dockerfile in your project. A default Dockerfile with the name `Dockerfile.project` is provided in the `etc` directory of your Apama installation. You can copy this file manually into the root of any project which can be deployed using the `engine_deploy` tool.

Also, note the use of build arguments in the Dockerfile. This allows you to use `--build-arg` to specify the name of an alternative builder or runtime image. If you want to use the automatically generated Dockerfile with your own image created from the packaging kit, you need to set the build arguments appropriately:

```
docker build -t appimage --build-arg APAMA_BUILDER=apamaimage
  --build-arg APAMA_IMAGE=apamaimage MyProject
```

Alternatively, you can just change the version of Apama that is used from Docker Hub:

```
docker build -t appimage --build-arg APAMA_VERSION=version
```

Note:

Each time you import an Apama project from a previous version into the current version, you have to update the version in the Dockerfile, or you have to run `docker build` with the appropriate build arguments to override the version. Software AG Designer will warn you if your Dockerfile is not up to date with the current version.

By default, the Dockerfile added by Software AG Designer uses the `softwareag/apama-builder` and `softwareag/apama-correlator` images, which do not contain components for connection to JMS, Cumulocity IoT, or other Software AG suite components. If your project uses this functionality, you may need to use one of the other pairs of published images instead, using the above-mentioned build arguments. For more details of the images available, see "Published Apama container images" on page 33. For exact details of the contents of each image, see the corresponding pages on Docker Hub. However, in general, the *builder* images contain the following additional tools:

- `engine_deploy` - to convert a project (either from Software AG Designer, or a collection of monitors created outside of Software AG Designer) into a configuration directory which can be used to start a correlator. See also "Deploying a correlator" on page 126.
- `engine_package` - to create CDP (correlator deployment package) files from monitor and event files. See also "Packaging correlator input files" on page 134.
- `apama_project` - to manipulate an Apama project outside of Software AG Designer. See also "Creating and managing an Apama project from the command line" on page 122.
- PySys - to run automated tests on your application before deployment. See also the *API Reference for Python*.

The image also contains a Java compiler. It does not by default contain a C++ compiler. If you want to compile C++ code, then you need to install a C++ compiler as part of your build step using a multi-stage build. This is only included while you are building, not in the final image, as long as you do it in the build part of a multi-stage build.

```
ARG APAMA_BUILDER=softwareag/apama-builder:version
ARG APAMA_IMAGE=softwareag/apama-correlator:version
FROM ${APAMA_BUILDER} as builder

COPY --chown=sagadmin:sagadmin MyProject ${APAMA_WORK}/MyProject
RUN engine_deploy --outputDeployDir ${APAMA_WORK}/MyProject_deployed \
${APAMA_WORK}/MyProject
RUN yum install gcc make && make -C MyProject

FROM ${APAMA_IMAGE}

COPY --chown=sagadmin:sagadmin --from=builder \
${APAMA_WORK}/MyProject_deployed ${APAMA_WORK}/MyProject_deployed
COPY --chown=sagadmin:sagadmin --from=builder \
${APAMA_WORK}/MyProject/libMyLib.so ${APAMA_WORK}/lib/libMyLib.so

WORKDIR ${APAMA_WORK}

CMD ["correlator", "--config", "MyProject_deployed"]
```

Using Docker Compose with Apama

Docker Compose uses a configuration file to define how to build and then how to run the one or more images and containers it defines. Where you have previously used individual commands for each image and container, you can now use Docker Compose to initiate the build or running of systems of containers.

A simple sample is provided in the `samples/docker/applications/Simple` directory of your Apama installation. Using that sample, you can see the basic workflow for all of the samples.

When run, the `docker-compose` command reads a configuration file that defines what it will build and how it should behave. This configuration file is named `docker-compose.yml` by default. It is recommended that you read the Docker documentation for the `docker-compose` command to get details on what the entries in the configuration file are.

It is important to note that the correlator may read all configuration with the extension of `.yaml` contained in a specified directory. You must take care over the placement and naming of any files in YAML format with an extension of `.yaml` to avoid unexpected behavior in the correlator. The following command uses the `docker-compose.yml` file in the current directory to build all the images needed for the containers defined in the `docker-compose.yml` file and then runs them as directed.

```
docker-compose up -d --rm
```

When run in the `Simple` sample directory, the above command builds and then runs the sample. You can use the following command to see output from the correlator running a simple "Hello world" application in the container:

```
docker-compose logs
```

To stop the container, enter the following command from the `Simple` sample directory:

```
docker-compose down
```

For more complex examples, see "Apama samples for Docker" on page 42. These examples demonstrate communications between containers, persisting container data, sharing between containers, and exposing the containers as services.

Apama samples for Docker

There are a number of samples that can be found in the `samples/docker/applications` directory of your Apama installation. The `Simple` sample that is referenced in "Using Docker Compose with Apama" on page 42 contains a "Hello world" application. The other samples in the above directory cover more complex use cases. These samples build upon the base image and demonstrate how to use a dockerized Apama correlator to build your own application or service.

The `README.txt` files that are provided in the `samples/docker/applications` directory and in each of the individual sample subdirectories guide you through the process of building and running. The samples demonstrate various ways in which Docker can be used to deploy Apama and related Software AG products. They also demonstrate interactions between Docker containers and normal applications such as dashboards.

Weather

This sample deploys Apama's Weather demo. It demonstrates that various Apama components can be run in distinct containers. The sample creates a correlator container and a separate container for the dashboard server which connects to the correlator just as in the non-Docker Weather sample. This sample also creates a Compose network called “front” which is used to allow both containers to communicate.

You can view this network using the following command:

```
docker network ls
```

Adapter

This sample starts an IAF in a container, which connects to a correlator running in another container. The IAF is running the File transport. An EPL application is deployed into the correlator, which requests the contents of a file `inputFile.txt` from the File adapter, and then directs it to write those contents back out to another file `outputFile.txt`.

The output file is stored within the container, but can be retrieved via Docker:

```
docker cp adapter_iaf_1:/apama_work/Adapter/outputFile.txt .
```

MemoryStore

Earlier samples showed how to use Dockerfiles to create derived images that give Apama components from the base image access to configuration and EPL code. However, this approach only suffices for static data that can be reproduced by copying in files from a canonical source. For containers to share dynamic data, they need a live view on that data. This is provided by a Docker feature called “volumes”, which allows containers to share parts of their file system with each other.

This sample contains a toy application `MemoryStoreCounter.mon` that makes use of the MemoryStore to lay down persistent state on disk in the form of a number that increments each time the monitor is loaded. While the correlator container is the one reading and writing to the MemoryStore, the persistent file is on a Docker volume which is persisted between container restarts.

You can view the created volume using the following command:

```
docker volume ls
```

Docker volumes give you the ability to manage data that has a different lifecycle to the container that uses it. In an application like this, you can replace the other containers with equivalents that are based on a newer version of Apama. After bringing them up again, the correlator will still have access to the MemoryStore data that it wrote in a previous iteration, as this data is owned by the volume.

Universal Messaging

This sample demonstrates two Apama correlators communicating with each other via a Universal Messaging realm server, all in separate containers. This sample requires an installation of Universal Messaging.

Secrets

This sample demonstrates how to use Docker secrets to set variables in correlator configuration files. These can then be loaded at runtime into a correlator.

Using the Apama image with the Docker stack

Current versions of Docker include swarm mode for natively managing a cluster of Docker engines called a *swarm*. You can use the Docker command line interface to create a swarm, deploy application services, and manage behavior. See the Docker documentation for available commands and more detailed information regarding swarms and their management.

The top of the hierarchy of distributed applications is the *stack*. A stack is a group of interrelated services that share dependencies, and can be orchestrated and scaled together. A single stack is capable of defining and coordinating the functionality of an entire application (though very complex applications may want to use multiple stacks).

Using stacks is an extension of creating a Compose file and then using the `docker stack deploy` command. The majority of samples described in "Apama samples for Docker" on page 42 use single service stacks running on a single host, which is not usually what takes place in production.

1. Use the following command to enable swarm mode:

   ```
   docker swarm init
   ```

   The `init` command outputs a token which can be used to add extra processing workers to the swarm using the following command:

   ```
   docker swarm join --token token workeraddress
   ```

2. The `docker-compose.yml` configuration file is the key element to creating the deployment. To use stack commands, the version of the configuration file must be greater than 3. The image specified in the configuration file must exist because (unlike Docker Compose) the stack commands ignore build sections in the configuration file. As a consequence, building the image is a separate step and must be done before attempting to start the application.

3. Use the following command to run the application:

   ```
   docker stack deploy stack --compose-file docker-compose.yml
   ```

   The *stack* element in the above command is a name which will be prefixed to the elements of the deployment that the above command creates. It is also used in the commands for interrogating the running system, for example:

   ```
   docker stack ps stack
   ```

```
docker stack services stack
```

4. Run the following command to cleanly shut down the running application when you are finished:

```
docker stack rm stack
```

5 Deploying Apama Applications with Kubernetes

Introduction to Apama in Kubernetes

Kubernetes is an open-source system that provides an alternative for orchestrating containers. The Apama images as described in "Deploying Apama Applications with Docker" on page 31 can be used within Kubernetes, allowing an alternative to deploying and controlling a user application. See https://kubernetes.io/ for a more detailed overview of Kubernetes and how to use it.

Images

Much like Docker, images form the basis of the containers that are run and controlled by Kubernetes. Creating and obtaining images is identical to Docker. That is, you can either get the Apama image from Docker Hub as described in "Quick start to using an Apama image" on page 35 or you build your own image as described in "Building an Apama image from the current installation" on page 37.

The images are templates that are used to create the containers. Kubernetes software runs on various operating systems and can run containers built from the Apama images.

Note:
Apama supports building images on Linux only.

Kubernetes

Kubernetes uses a different command line application and terminology from Docker. For example, the simplest unit is a *pod*. A pod corresponds to a running process on the cluster, but can be more than one *container*. The command line interface is `kubectl` and should be used instead of `docker`. See the Kubernetes documentation for more details on the various command-line options.

Quick start to using Apama in Kubernetes

For this quick start, you have to build the image from the `Simple` sample that can be found in the `samples/docker/applications/Simple` directory of your Apama installation. See the `README.txt` file in the `samples/docker/applications` directory for detailed instructions.

Once you have built the image, proceed as described below.

1. Define a deployment YAML file which references the image you wish to run as a container. An example of this is the `kubernetes.yml` file which can be found in the `samples/docker/applications/Simple` directory of your Apama installation.

   This example YAML file creates a pod. However, Kubernetes can be used to define much more complex objects and behaviors; see the Kubernetes documentation for details of the possible configurations.

2. Use the Kubernetes command line tool (`kubectl`) to create the container and run it (for example, using the above example YAML file):

   ```
   kubectl create -f kubernetes.yml
   ```

The `create` command starts the container under Kubernetes control. You can now use the `kubectl` command line to interrogate the pod that has been created, see below.

3. To return the name of the pod that corresponds to running your Apama container:

```
kubectl get pods
```

4. To examine the logs from the container in the pod:

```
kubectl logs podname
```

5. Once you are finished, you can shut down everything and remove the containers. To do so, you use the same YAML file that has been used to start the process:

```
kubectl delete -f kubernetes.yml
```

The power of Kubernetes comes with more complex setups involving multiple containers and hosts. Some of these features are covered in more complex samples; see "Apama samples for Kubernetes" on page 51 for further information.

Deploying an Apama application using Kubernetes

You will either need to create an image or you will already have images for the application you wish to deploy using Kubernetes. To illustrate the concepts and basic process of getting the image running and interrogating the running containers, this topic describes how to use the `Adapter` sample which can be found the `samples/docker/applications/Adapter` directory of your Apama installation. The `README.txt` files that are provided in the `samples/docker/applications` directory and in the subdirectory for the `Adapter` sample guide you through the process of building and running.

1. To create the images, enter the following commands from the `samples/docker/applications/Adapter` directory (see also "Deploying Apama Applications with Docker" on page 31 for details on building images):

```
cd deployment
docker build -t registrytag-correlator .
cd ../iaf
docker build -t registrytag-iaf .
cd ..
docker push registrytag-correlator
docker push registrytag-iaf
```

   You must replace the tags *`registrytag-correlator`* and *`registrytag-iaf`* with tags that are valid for a registry that you can write to, in the following form:

   `registryhost/organization/repository``:`*`image`*

   To use Kubernetes, you must publish the images in a registry and not just in a local Docker server. You must also include these tags in the Kubernetes configuration file that is used to create the objects. If you are using the `kubernetes.yml` file from the `Adapter` sample (which is mentioned below), you must replace the `correlator-image` and `iaf-image` lines with your tags.

2. Once you have determined which images you want to use, and know their location and tag name, you can create the YAML configuration file for Kubernetes. This configuration file

defines the state that you want from a running system, indicating the runtime characteristics you want Kubernetes to adhere to. This file makes no mention of where things run on a cluster, but it can be used to determine behavior like restarting, replica containers, load balancing and resource restrictions.

Apama provides the example configuration file `kubernetes.yml` which can be found in the `samples/docker/applications/Adapter` directory of your Apama installation. Detailed descriptions of the possible contents of a Kubernetes configuration file can be found on the Kubernetes website (https://kubernetes.io/).

The `Adapter` sample contains two pods and a service tying the running containers together. A service in Kubernetes is an abstraction which enables a loose coupling between dependent pods. Although each pod has a unique IP address, these IP addresses are not exposed outside the cluster without a service. Services allow your applications to receive traffic. Whenever you have a pod which depends on another service, it is recommended that you use an init container to ensure that the service is ready before starting the dependant pod. All the samples involving multiple pods use this pattern.

3. To create the Kubernetes objects (pods and service):

   ```
   kubectl create -f kubernetes.yml
   ```

   After you have created the objects defined in the `kubernetes.yml` file, you can list the pods and the service, and you can examine the details of the running objects, see below.

4. To list the pods:

   ```
   kubectl get pods
   ```

5. To list the service:

   ```
   kubectl get services
   ```

6. To examine the details of the running objects (image, container, volumes, and status events):

   ```
   kubectl describe pod engine
   ```

7. To examine the logs in the correlator instance:

   ```
   kubectl logs engine
   ```

8. To examine the IAF logs:

   ```
   kubectl logs iaf
   ```

9. You can run commands in the pod. For example, to run a single command:

   ```
   kubectl exec engine 'ls'
   ```

   Or to open a shell on the running container:

   ```
   kubectl exec -it engine bash
   ```

10. To shut down the application, you use the same YAML file that has been used to start the process:

```
kubectl delete -f kubernetes.yml
```

For more samples of Kubernetes configurations applied to Apama images and applications, see "Apama samples for Kubernetes" on page 51.

Apama samples for Kubernetes

There are a number of samples that can be found in the `samples/docker/applications` directory of your Apama installation. The `Simple` sample has already been referenced in "Quick start to using Apama in Kubernetes" on page 48. The other samples in the above directory cover more complex use cases. These samples build upon the base image and demonstrate how to use an Apama correlator in a Kubernetes environment to build your own applications.

The `README.txt` files that are provided in the `samples/docker/applications` directory and in each of the individual sample subdirectories guide you through the process of building and running. The samples demonstrate various ways in which Kubernetes can be used to deploy Apama and related Software AG products. They also demonstrate interactions between containers and normal applications such as dashboards.

Weather

This sample deploys Apama's Weather demo. It demonstrates the use of separate pods and connecting services to allow communication between them. A correlator pod and a dashboard server pod connect just as in the non-Docker Weather sample. The engine and weather services expose the required ports allowing connections to be made.

Adapter

This sample starts an IAF pod which connects to a correlator pod connected by a service. The IAF is running the File transport. An EPL application is deployed into the correlator, which requests the contents of a file `inputFile.txt` from the File adapter, and then directs it to write those contents back out to another file `outputFile.txt`.

The configuration for the correlator pod defines a `readinessProbe` that checks that the correlator is running before the pod is marked ready. In a similar way, the IAF pod configuration defines an `initContainer` to make sure the service it uses to connect to the correlator is present before continuing.

See also " Deploying an Apama application using Kubernetes" on page 49 which refers to this sample.

MemoryStore

For containers to share dynamic data, they need a live view on that data. This is provided by a Kubernetes feature called "volumes", which allows containers to share parts of their file system with each other. We use the Kubernetes `PersistentVolume`, `PersistantVolumeClaim` and `Deployment` objects to implement the sample.

This sample contains a toy application `MemoryStoreCounter.mon` that makes use of the MemoryStore to lay down persistent state on disk in the form of a number that increments each time the monitor

is loaded. While the correlator container is the one reading and writing to the MemoryStore, the persistent file is on a volume which is persisted between container restarts.

Kubernetes volumes give you the ability to manage data that has a different lifecycle to the container that uses it. In an application like this, you can replace the other containers with equivalents that are based on a newer version of Apama. After bringing them up again, the correlator will still have access to the MemoryStore data that it wrote in a previous iteration, as this data is owned by the volume.

Universal Messaging

This sample demonstrates two Apama correlators communicating with each other via a Universal Messaging realm server. Therefore, Universal Messaging must be installed to run. A service is created that allows the correlators to communicate with the Universal Messaging pod, and the correlator pod uses an `initContainer` element in the Kubernetes configuration to ensure that the Universal Messaging service exists before continuing.

Secrets

This sample demonstrates how to use Kubernetes secrets to set variables in correlator configuration files. These can then be loaded at runtime into a correlator.

6 Tuning Correlator Performance

This section addresses how to scale up Apama to improve upon the performance of a single correlator. It describes the Apama features you can use to send events to multiple 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 correlator component and the scalability features of the correlator platform.

Should it prove necessary, capacity can further be increased by using multiple correlators on multiple hosts. To facilitate such multi-process deployments, Apama provides features to enable connecting components to pass events between them. It is recommended that each correlator is run on a separate host, to assist in the configuration of scaled-up topologies. However, it is possible to run multiple correlators on a single host. There are two methods of configuration:

- Using the configuration tools from the command line or Apama macros for Ant.
- 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. Java plug-ins can be used if invocation of Java code is required on multiple threads, either directly from EPL or by registering an event handler. See "Using EPL plug-ins written in Java" in *Developing Apama 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*.

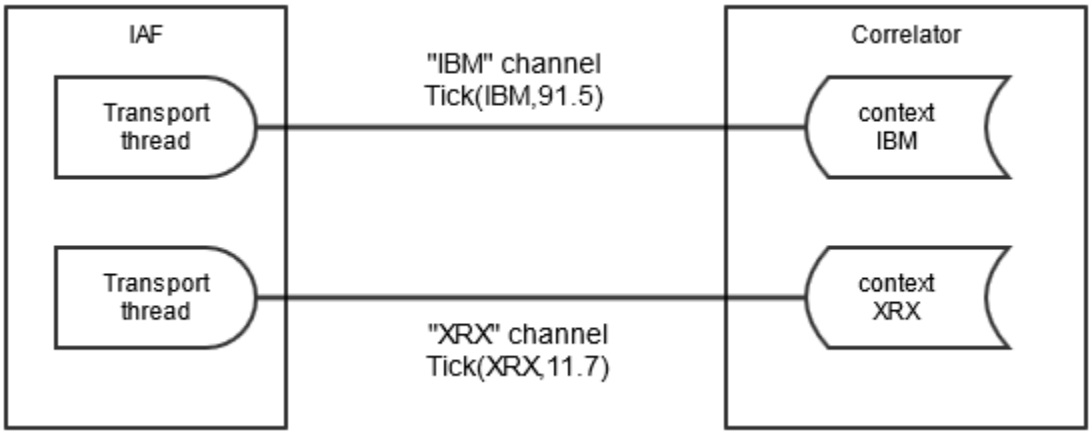
The core event processing and matching service offered by Apama is provided by one or more correlator processes. In a simple deployment, Apama comprises a single correlator connected directly to at least one input event feed and output event receiver. Although this arrangement is suitable for a wide variety of applications (the actual size depending on the hardware in use, networking, and other available resources), for some high-end applications it may be necessary to scale up Apama by deploying multiple correlator processes on multiple hosts to partition the workload across several machines.

Partitioning strategies

Using the patterns and tools described in this guide it is possible to configure the arrangement of multiple contexts within a single correlator or multiple 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 contexts, spawned monitors expected and the inter-dependencies between monitors and events. The following examples illustrate this.

The `stockwatch` sample application (in the `samples\epl` folder of your Apama installation directory) monitors for changes in the values of named stocks and emits an event should a stock of interest fall below a certain value. The stocks to watch for and the prices on which to notify are set up by initialization events, which cause monitors that contain the relevant details to be spawned. In this example, the need for partitioning arises from a very high event rate (perhaps hundreds of thousands of stock ticks per second), which is too high a rate for a single context to serially process.

A suitable partitioning scheme here might be to split the event stream in the adapter, such that different event streams are sent on different channels. The illustration below shows how this can be accomplished:



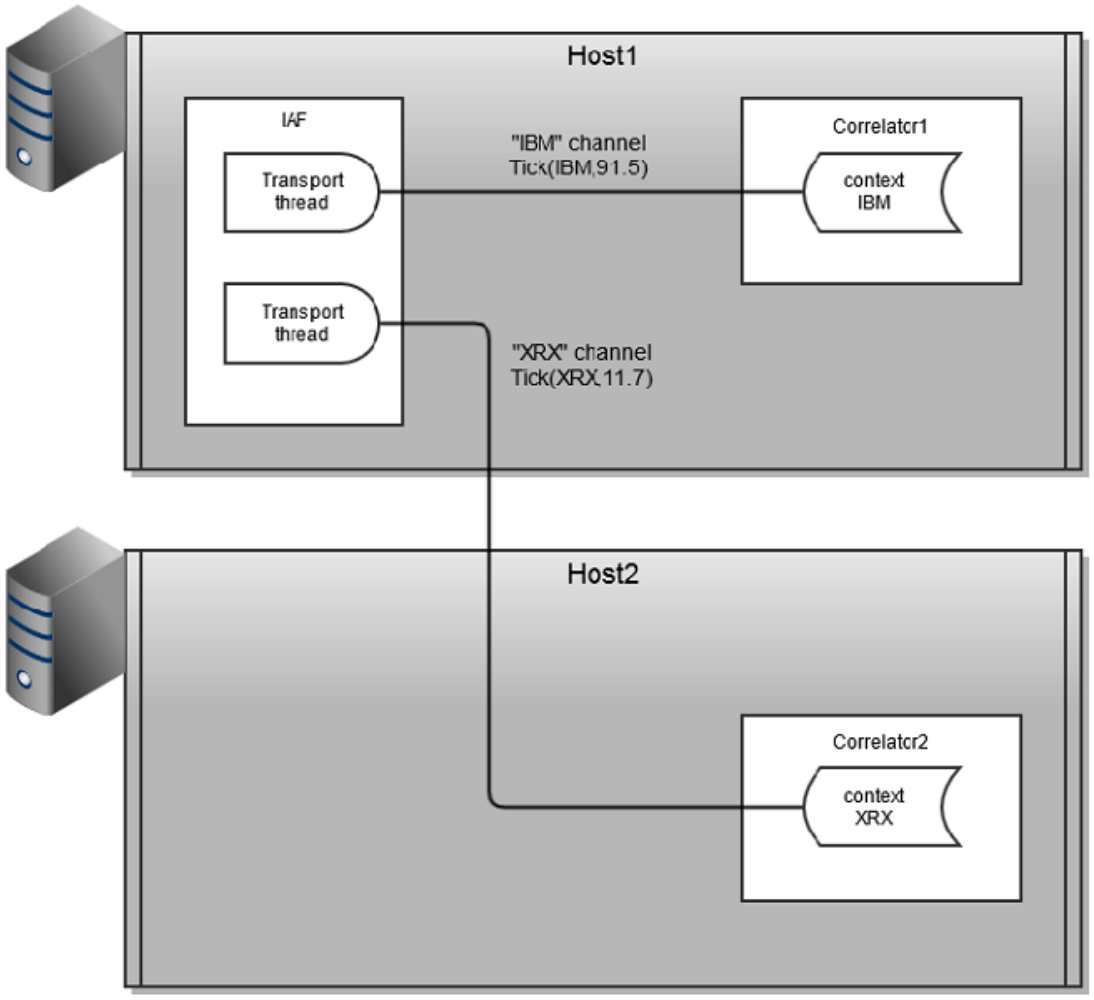
This diagram shows an adapter sending events to different channels based on the symbol of the stock tick. The adapter transport configuration file would specify a `transportChannel` attribute for the stock event that named a field in the `NormalisedEvent` that specified the stock symbol. Either a thread per symbol or a single thread (which could become a bottleneck) managed by the transport, depending on what the system the transport is connecting to allows, is used to send `NormalisedEvents` to the semantic mapper to be processed. The IAF thus sends the events on the channel in the stock symbol value in the `NormalisedEvent`.

In this example, the stock symbol is either “IBM” or “XRX”. The IAF will send events to all sinks (typically one) that are specified in the IAF's configuration file. In the correlator, all monitors interested in events for a given symbol would need to set up listeners in a context where a monitor has subscribed to that symbol. To achieve good scaling, the application is arranged so that each context is subscribed to only one symbol. For the `stockwatch` application, a separate context per symbol would be created, and the `stockwatch` monitor spawns a new monitor instance to each context. In each context, the monitor instance would execute `monitor.subscribe(stockSymbol);` where `stockSymbol` would have the value `"IBM"` or `"XRX"` corresponding to the stock symbol it is interested in. This application will scale well, as each event stream (for the different stock symbols) can run in parallel on the same host; this is referred to as scale-up.

Listeners in each context would listen for events matching a pattern, such as `on all Tick(symbol="IBM", price < 90.0)`.

If the number of stock symbols is very large and the amount of processing for each stock symbol is large, then it may be required to run correlators on more than one host to use more hardware resources than are available in a single machine. This is referred to scale-out. To achieve scale-out, connections per channel need to be made between the Apama components using the `engine_connect`

tool (or the equivalent call from Ant macros or the client API). The `engine_connect` tool can connect any two Apama components, either correlator to correlator, or IAF to correlator. For best scaling, multiple connections are required between components, which `engine_connect` provides in the parallel mode. The following image shows a scaled out configuration.



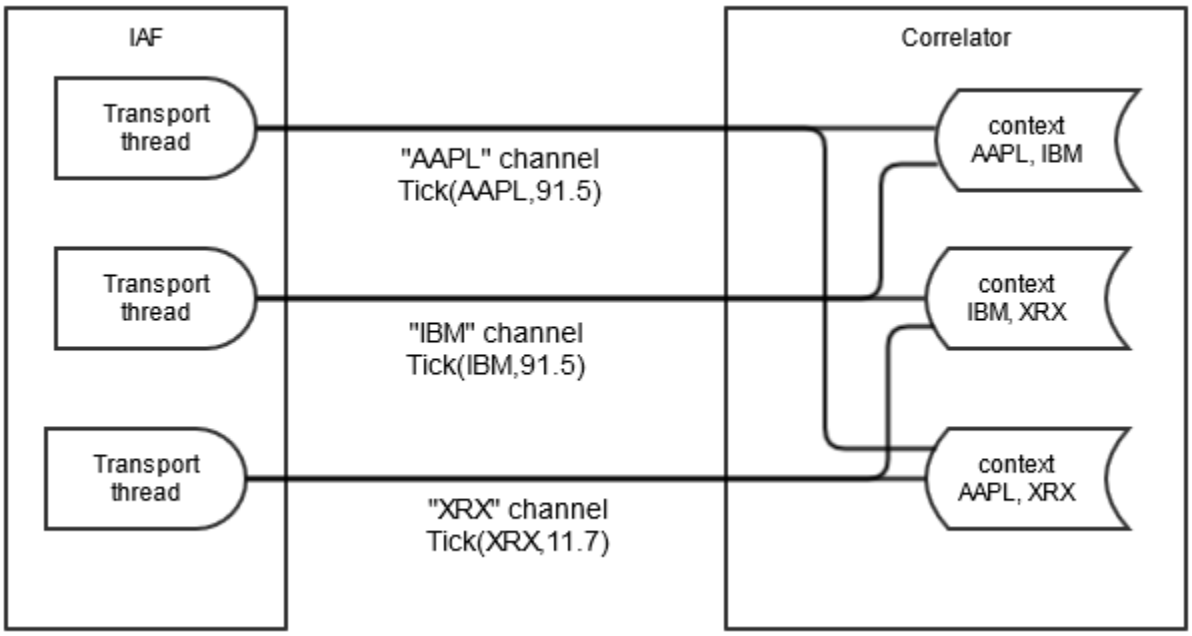
This configuration allows many contexts to run on two hosts and requires use of `engine_connect` to set up the topology.

Now 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. 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 context to perform the necessary recalculations for each stock tick, thus requiring the application to be partitioned across multiple contexts.

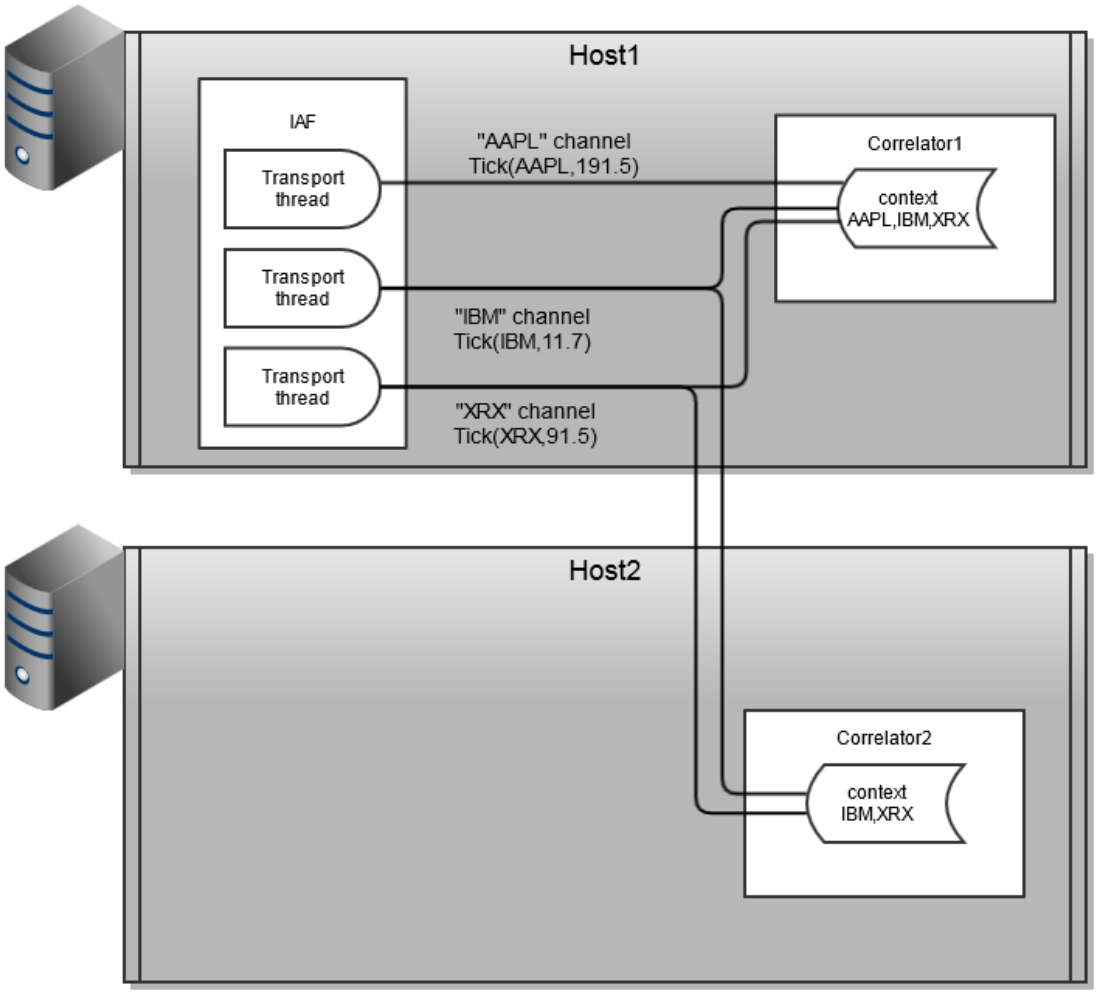
Unlike the `stockwatch` application, it is not possible to achieve scaling to larger numbers of portfolios by splitting the event stream. Each portfolio can contain multiple stocks, and stocks can be dynamically added and removed, thus one event may be required by multiple contexts. In this

case, a suitable partitioning scheme is to partition the monitor instances across contexts (as with `stockwatch`) but to duplicate as well as split the event stream to each correlator. The following images shows the partitioning strategy for the portfolio monitoring application.



Again, each monitor instance is spawned to a new context and subscribes to the channels (stock symbols in this application) that it requires data for. Note that while the previous example would scale very well, this will not scale as well. In particular, if one monitor instance requires data from all or the majority of the channels, then it can become a bottleneck. However, there may be multiple such monitor instances running in parallel if they are running in separate contexts.

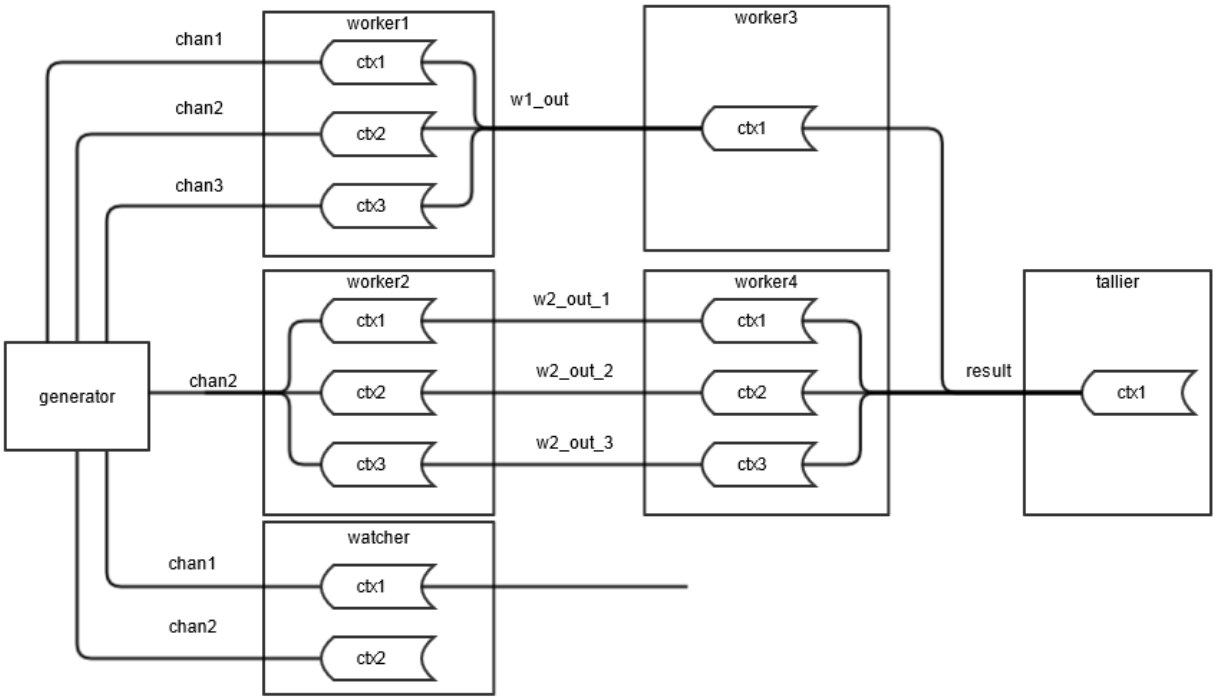
Similar to the `stockwatch` application, the portfolio monitoring application may require scale-out across multiple hosts, as shown below.



In summary, the partitioning strategy can be thought of as a formula for splitting and duplicating monitors and/or events between correlators while preserving the correct behavior of the application. In some circumstances, it may be necessary to re-write monitors that work correctly on a single correlator to allow them to be partitioned across correlators, as the following section describes.

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 correlators on separate channels, such that events sent by a source correlator on a specific channel find their way to the correct contexts in the target correlator. A set of two or more correlators connected in this way is known as a correlator pipeline, as shown in the following image. 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.



In this image, a correlator can perform the function of each of the 7 nodes (generator, 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, sent on the channels as marked on the diagram. `tallier` collates the results from these correlators for forwarding to any registered receivers. A final correlator, `watcher`, monitors the events emitted by `generator` 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 the previous figure, it is assumed that 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 sent events, which contexts in the connected correlators receive by subscribing to the appropriate channels.

To realize this application structure requires coding each chunk of processing as one or more separate monitors, which send 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 that grows beyond the capacity of a single 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.

Correlator pipelining

To implement engine topologies comprising multiple correlators requires a method of connecting correlators 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 60.
- Indirectly using Software AG's Universal Messaging message bus. For complex deployments where parts of the application may be moved between Apama correlators, this is likely to be the best alternative. When using Universal Messaging, each correlator connects to the same Universal Messaging realm. See "The Universal Messaging Transport Connectivity Plug-in" in *Connecting Apama Applications to External Components*.
- Programmatically via the client API, see "Configuring pipelining through the client API" on page 67.
- Using a custom launch configuration in Software AG Designer. See "Connecting correlators" in *Using Apama with Software AG Designer*.

Configuring pipelining with engine_connect

The `engine_connect` tool allows direct connection of running correlator instances. The executable for this tool is located in the `bin` directory of the Apama installation. Running the tool in the Apama Command Prompt or using the `apama_env` wrapper (see "Setting up the environment using the Apama Command Prompt" on page 15) ensures that the environment variables are set correctly.

Note:

Some of the functions of the `engine_connect` tool can be performed from within EPL. For more information, see "Using the Management interface" in *Developing Apama Applications*.

Synopsis

To configure pipelining, run the following command:

```
engine_connect [ options ]
```

When you run this command with the `-h` option, the usage message for this command is shown.

Description

`engine_connect` connects a source correlator (the sender) to a target correlator (the receiver). The target correlator will receive events from the specified channel(s) of the source correlator. Source and target correlators must already be running.

Alternatively, if you specify the `-f` option, `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 67 for details of the file format). The `engine_connect` tool 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 a hyphen (`-`), then connection information is read from the standard input device (`stdin`) until end-of-file.

The connection between the source and target correlators is persistent. When one of the correlators stops running, then when that correlator restarts it automatically reconnects with the other correlator.

The tool is silent by default unless an error occurs. For verbose progress information, use the `-v` option.

Options

The `engine_connect` tool takes the options listed below.

Note:

Many of these options can also be specified as elements of a YAML configuration file (with different element names). If an option is specified in both the command line and a YAML configuration file, then the command line takes precedence. For further information, see "Configuring the correlator" on page 101 and especially the topic "Setting up connections between correlators in a YAML configuration file" on page 117.

Option	Description
`-h | --help`	Displays usage information.
`-sn` *`host`* `| --sourcehost` *`host`*	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`* `| --sourceport` *`port`*	Port on which the source (event sending) correlator is listening. The default is `15903`.
`-tn` *`host`* `| --targethost` *`host`*	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`* `| --targetport` *`port`*	Port on which the target (event receiving) correlator is listening. The default is `15903`.
`-c` *`channel`* `| --channel` *`channel`*	Named channel on which to send/receive events. You can specify the `-c` option multiple times to send/receive events on

Option	Description
	multiple channels. You must specify the `-c` option at least once for each sender/target pair. Until you do, no events emitted by the sender correlator are received by the target correlator. An event is discarded if it is sent on a channel for which you did not specify the `-c` option.
`-m` *mode* | `--mode` *mode*	Indicates whether there is one connection (`-m legacy`) between the sender and target correlators or one connection for each specified channel (`-m parallel`). The default behavior is that there is one connection between the sender and target correlators. The tool uses the same connection for every channel. Events sent on any channel are delivered to the default channel in the target correlator and all events are delivered in order. You can specify default behavior by specifying `-m legacy` or `--mode legacy`. To create a connection for each specified channel, specify `-m parallel` or `--mode parallel`. Events sent on a named channel are delivered to the same named channel in the target correlator. Events sent on the same channel are delivered in order. Events sent on different channels may be re-ordered. You also specify the `-m` option when you specify the `-x` option to disconnect. If you are using a separate connection for each channel, you should specify `-m parallel` when you specify the `-x` option. If you are using one connection for all channels, you should specify `-m legacy` when you specify the `-x` option. See also "Avoid mixing connection modes" on page 64.
`-x | --disconnect`	When you specify the `-x` option, the behavior depends on whether you also specify the `-c` option. If you specify the `-x` option and you do not also specify the `-c` *channel* option, then the source correlator stops sending events to the target correlator. Each connection between the source correlator and the target correlator is terminated. If you specify the `-x` option and the `-c` *channel* option and the tool is using one connection for each channel, then the source correlator terminates only the connection(s) it was using for the channel(s) you specify. Any other connections being used for other channels continue to be used. You can specify the `-x` option with one or more instances of the `-c` *channel* option. Remember to also specify `-m parallel`. If you specify the `-x` option and the `-c` *channel* option and the tool is using one connection for all channels, then the source

Option	Description
	correlator stops sending events on only the channel(s) you specify. The source correlator continues to send events on any other channels it was already sending events on. If there are no other channels, then the source correlator no longer sends events to the target correlator. However, the connection between the two correlators remains in place. Remember to also specify `-m legacy`.
`-s | --qdisconnect`	Disconnect if slow (only takes effect on the first connection).
`-f` *file* `| --filename` *file*	Read connection information from the named file. If this option is specified, the options `-sn`, `-sp`, `-tn`, `-tp` and `-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 | --utf8`	Indicates that the connection information file is in UTF-8.
`-v | --verbose`	Requests verbose output during `engine_connect` execution.
`-V | --version`	Displays version information for the `engine_connect` tool.

Exit status

The `engine_connect` tool returns the following exit values:

Value	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.

Comparison of legacy and parallel connection modes

Legacy connection mode	Parallel connection mode
0 or 1 connection between two correlators.	Any number of connections between correlators.
Events sent on different channels are delivered in the order in which they are sent.	Events sent on different channels may be delivered in a different order from the order in which they were sent.

Legacy connection mode	Parallel connection mode
Sending an event to a named channel delivers the event to the default channel.	Sending an event to a named channel delivers it to only that channel.
Unlike Universal Messaging for passing events between correlators.	Similar to Universal Messaging for passing events between correlators.
Same behavior as releases earlier than Apama 5.2.	New behavior starting with Apama 5.2.

Universal Messaging has no mechanism for enforcing ordering among events sent on different channels. However, Universal Messaging is the better alternative when you want to use a large number of channels to send events between components. Without Universal Messaging, the use of two TCP connections with threads on both ends of the connection might reach the limit of how many channels can have dedicated connections.

Avoid mixing connection modes

Successive command lines that specify the same source/target hosts/ports build on each other. While this makes it possible to mix the legacy and parallel connection modes, you should avoid doing that. Mixing connection modes can cause an event to be delivered twice to the same channel. For example:

```
engine_connect -tp 15902 -sp 15903 -c channelA -c channelB
engine_connect -tp 15902 -sp 15903 -c channelA -c channelC -m legacy
```

The result of the first command is that there is one (legacy) connection for sending/receiving events on `channelA` and `channelB`. The result of the second command is that there is a dedicated connection for sending/receiving events on `channelA` and a dedicated connection for sending/receiving events on `channelC`. Events sent on `channelA` would be delivered twice: once on the legacy connection and once on the dedicated connection.

Examples

Because you can specify command lines that build on each other, you could set up a connection and add named channels later. You can also unsubscribe the channels you have added so that no events are sent or received. The connection remains and you can re-add channels at a later time. However, until you specify the `-c` option for a given connection, no events emitted by the source correlator are received by the target correlator. Consider the following command line:

```
engine_connect -sn host1 -sp 15903 -tn host2 -tp 15904
```

The correlators on `host1` and `host2` are connected but no channels have been subscribed and therefore no events are sent/received. To send and receive events, specify the `-c` option as in the following command line:

```
engine_connect -sn host1 -sp 15903 -tn host2 -tp 15904 -c CHAN1 -c CHAN2
```

Now the connected correlators can use `CHAN1` and `CHAN2` to send/receive events. To add another channel, execute this command:

```
engine_connect -sn host1 -sp 15903 -tn host2 -tp 15904 -c CHAN3
```

The correlators are now using `CHAN1`, `CHAN2`, and `CHAN3` to send/receive events. To stop using `CHAN2`, execute the following command. The correlators continue to use `CHAN1` and `CHAN3`.

```
engine_connect -sn host1 -sp 15903 -tn host2 -tp 15904 -x -c CHAN2
```

To stop sending and receiving events, execute the following command. Note that the correlators remain connected until one of them stops. There is no penalty for this connection.

```
engine_connect -sn host1 -sp 15903 -tn host2 -tp 15904 -x
```

In this example, the following command is equivalent to the previous command.

```
engine_connect -sn host1 -sp 15903 -tn host2 -tp 15904 -x -c CHAN1 -c CHAN3
```

Connection configuration file

`engine_connect` can take connection information from a file for connecting and disconnecting correlators. A sample of such a configuration file is shown below, which defines the topology shown in "Engine topologies" on page 58.

```
# comments are allowed prefixed by a '#' - the rest of the line
# is ignored
generator:dopey.apama.com:1234
worker1:sleepy.apama.com:1234:generator{chan1,chan2,chan3}
worker2:grumpy.apama.com:1234:generator{chan2}
worker3:sneezy.apama.com:1234:worker1{w1_out}
worker4:bashful.apama.com:1234:worker2{w2_out_1,w2_out_2,w2_out_3}
tallier:happy.apama.com:1234:worker3{result},worker4{result}
watcher:doc.apama.com:1234:generator{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 `generator`, 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 `generator` 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 -m parallel -sn sneezy.apama.com -sp 1234 -tn happy.apama.com
    -tp 1234 -c result
>engine_connect -m parallel -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 -m parallel -f topology.dat
>engine_connect -m parallel -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 -m parallel -sn dopey.apama.com -sp 1234 -tn sleepy.apama.com
    -tp 1234 -c foo
>engine_connect -m parallel -f topology.dat
```

will first add a connection from correlator `generator` to `worker1` on channel `foo`, then (from the configuration file) extend that connection so that `worker1` also receives all events from `generator` 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`.

Configuring pipelining through the client API

Apama provides client software development kits (SDKs) that can be used to interface with a running correlator or group of correlators. You can use `attachAsConsumerOfEngine` of the Engine Client API in Java and .NET (or `attachAsConsumerOfEngine` of the lower-level Engine Management API in Java, .NET, C++ and C). For more information, see "Developing Custom Clients" in *Connecting Apama Applications to External Components*.

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. Use the `engine_inject` tool described in "Injecting code into a correlator" on page 119, or by means of the relevant functions of the client library, to load the relevant monitors directly on to the appropriate correlators, specifying the host and port for each correlator.

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. Use the IAF `transportChannel` attribute to specify the channel an event is sent to, and connect that channel to the appropriate correlators.

Using jemalloc to optimize memory usage

Apama supports the jemalloc memory allocator as an optional alternative to its standard memory allocator. See also http://jemalloc.net/.

Note:
This option is only available on Linux platforms and is not available on Windows at this time.

jemalloc typically consumes less memory overall. Therefore, it is advisable to use it when the process needs excessive memory or is running out of memory. But keep in mind that there may be throughput, latency or overall performance hits if you use this option.

To enable jemalloc as the memory allocator, set the following environment variable before starting the correlator:

```
AP_ALLOCATOR=jemalloc
```

7 Restricting Correlator Resource Usage with Control Groups

On Linux platforms, control groups (cgroups) can be used to limit the resource usage of processes, for example, by limiting the number of CPUs a process can detect. Apama uses the number of CPUs to determine how many threads to create. So you can use cgroups to limit this.

All cgroup restrictions that are currently active are logged at startup by the correlator (see also "Starting the correlator" on page 78).

The number of CPUs and memory available to the correlator are exposed through various means under the labels `physicalCores` and `availableMemoryMB`:

- Management interface. See also "Using the Management interface" in *Developing Apama Applications*.
- Apama's REST API. See also "Managing and Monitoring over REST" on page 71.
- Prometheus (under the metric `sag_apama_correlator_metadata`). See also "Monitoring with Prometheus" on page 75.

8 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, or receiving JSON 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 a document with the current status of the correlator. The format of this document is depicted by the header set in the request, that is, `application/xml` for XML and `application/json` for JSON.

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, but may only support XML and not JSON. 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. However, the `/request` URI will accept JSON or XML documents via `PUT`.

All requests and responses over these interfaces have the same, simple elements:

- In XML, these elements are:

  ```
  prop

  map

  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.

- In JSON, these elements are:

  ```
  map {}

  list []
  ```

  All elements have a name-value pair. Name and value are separated by a colon (:) with the name to the left and the value to the right of the colon. The `map` element, which is represented by curly brackets, is an unordered list of named elements which might be any of `map` or `list` elements. See the `/info` URI as an example. The `list` element, which is represented by square brackets, is very similar to the `map` element except that here the order is typically regarded as significant. If there is an error, then a response with the error message is returned, for example, `{"apamaErrorMessage":"Not found"}`.

For both formats, the `/connections` URL is a good example of all these elements being used together:

- In XML, the top-level element is a `map` which has two children, both `list` elements, called `senders` and `receivers`. Each list contains a `map` element for each sender and receiver. Each sender or receiver has a set of `prop` elements.
- In JSON, the top-level element is a `map {}` which has two children, both `list []` elements, called `senders` and `receivers`. Each list contains a `map {}` element for each sender and receiver. Each sender or receiver has a set of name-value pairs.

For detailed information on the supported URIs, see the *API Reference for Component Management REST APIs*.

Two examples are provided below, one for JSON and another for XML. Each example shows only a possible hierarchy of a response. To get the actual format of the response for each request, it is recommended that you actually make the request.

Example for JSON:

```
{
  "Key1": [
    {"Key1.1.1":"Value1.1.1","Key1.1.2":"Value1.1.2"},
    {"Key1.2.1":"Value1.2.1","Key1.2.2":"Value1.2.2"}
   ],
  "Key2":[],
  "Key3":
   [
    {"Key3.1.1":"Value3.1.1","Key3.1.2":[]}
   ],
  "Key4":[]
}
```

Example for XML:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="/resources/transform.xslt"?>
<map name="apama-response">
  <list name="Key1">
    <map name="Key1.1">
      <prop name="Key1.1.1">Value1.1.1</prop>
      <prop name="Key1.1.2">Value1.1.2</prop>
    </map>
```

```
    <map name="Key1.2">
      <prop name="Key1.2.1">Value1.2.1</prop>
      <prop name="Key1.2.2">Value1.2.2</prop>
    </map>
  </list>
  <list name="Key2"/>
  <list name="Key3">
    <map name="Key3.1">
      <prop name="Key3.1.1">Value3.1.1</prop>
      <list name="Key3.1.2"/>
    </map>
  </list>
  <list name="Key4"/>
</map>
```

9 Monitoring with Prometheus

Prometheus is an open-source project that is used for monitoring application state. See https://prometheus.io/ for detailed information on Prometheus and how to use it.

Standard correlator status

Apama exposes many internal correlator statistics as Prometheus metrics over HTTP on the `/metrics` endpoint. For the full list of built-in metric names, see "List of correlator status statistics" on page 143.

A sample demonstrating basic Prometheus usage, such as what metrics are exposed and how to define new metrics, can be found in the `samples/prometheus/basic` directory of your Apama installation.

User-defined status

In addition, any user-defined status can be exposed over Prometheus, provided the status name is acceptable and the value is lexically equivalent to a number. All user-defined status names first go through a simple escaping scheme where each comma (,), period (.) and hyphen (-) character is replaced by an underscore (_), and then checked against the Prometheus metric name regex. If this passes, the name is considered acceptable.

User status metrics with names containing labels in the form `{keyA=valueA,keyB=valueB}` are converted to Prometheus metrics with those labels in the Prometheus labels. For example, if you have `somename{key=value}` as part of the user status name, this is converted to a Prometheus label on the metric `somename`.

Note:

Each user-defined status that is exposed as a metric is of the `gauge` type.

See also "User-defined status reporting from connectivity plug-ins" in *Connecting Apama Applications to External Components*.

Prometheus metadata metrics

The following Prometheus metrics use Prometheus labels attached to the metric to provide additional information:

- `sag_apama_correlator_licensedata`

  This metric always returns a value of zero, but its labels give information about the status of the current license.

- `sag_apama_correlator_metadata`

  This metric always returns a value of zero, but its labels give information about the current configuration.

10 Correlator Utilities Reference

The Apama 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 correlators.

For information about EPL, event definitions, monitors, namespaces and packages, see "Getting Started with Apama EPL" in *Developing Apama Applications*.

Starting the correlator

The `correlator` executable starts the correlator. This is located in the `bin` directory of the Apama installation. Running the executable in the Apama Command Prompt or using the `apama_env` wrapper (see "Setting up the environment using the Apama Command Prompt" on page 15) ensures that the environment variables are set correctly.

Synopsis

To start the correlator, run the following command:

```
correlator [ options ]
```

When you run this command with the `-h` option, the usage message for this command is shown.

Description

By default, the `correlator` executable starts a 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 a 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 161. 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.

Note:

If a license file cannot be found, the correlator will run with reduced capabilities. See "Running Apama without a license file" in *Introduction to Apama*.

Options

The `correlator` executable takes the options listed below.

Note:

Many of these options can also be specified as elements of a YAML configuration file (with different element names). If an option is specified in both the command line and a YAML configuration file, then the command line takes precedence. For further information, see

"Configuring the correlator" on page 101 and especially the topic "YAML configuration file for the correlator" on page 102 which lists the available elements.

Option	Description
`-V | --version`	Displays version information for the correlator.
`-h | --help`	Displays usage information.
`-p port | --port port`	Specifies the port on which the correlator should listen for monitoring and management requests. The default is `15903`. Alternatively, you can specify the port in a YAML configuration file. See "Specifying the correlator port number" on page 109 for details.
`-f file | --logfile file`	Specifies the path of the main log file that the correlator writes messages to. The default is `stdout`. See also "Specifying log filenames" on page 90, "Descriptions of correlator status log fields" on page 93 and "Text internationalization in the logs" on page 96.

`-v level | --loglevel level`
 	
or
 	
`-v category=level | --loglevel category=level`
	Specifies the level of information that the correlator sends to the main correlator log file. Specify one of the following:
	
	■ A log level which is to apply to all messages written to the log file.
	
	■ A category with the log level for that category. This option can be provided multiple times.
	
	You can also specify the log level in a YAML configuration file. See "Setting correlator and plug-in log files and log levels in a YAML configuration file" on page 113 for details. This topic also lists the category names that can be specified.
	
	A log level can be one of the following (in increasing order of verbosity): `ERROR`, `WARN`, `INFO`, `DEBUG`, `TRACE`. The default is `INFO`.
	
	The use of `DEBUG` and `TRACE` is not recommended in a production environment as the amount of logging information will impact performance.
	
	The use of `ERROR` and `WARN` is not recommended. These log levels may make it impossible to provide support due to the lack of information

Option	Description
	in the log file. If one of these log levels is used, a warning is printed at the top of the correlator log file. **Note:** If `OFF`, `CRIT` or `FATAL` is specified, the log level is reset to `ERROR` and a warning message is printed at the top of the correlator log file.
`-t | --truncate`	Specifies that if the main correlator log file already exists, the correlator should empty it first. The default is to append to it.
`-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 license file.
`-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 97.
`-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 97. The default is that the correlator does not disconnect slow receivers.
`--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:**

Option	Description
	Setting `logQueueSizePeriod` to `0` turns logging off. Without correlator logging information, it is impossible to effectively troubleshoot problems. See also "Descriptions of correlator status log fields" on page 93.
`--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*.
`--jmsConfig` *dir*	Specifies that correlator-integrated 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 correlator-integrated messaging for JMS, see "Configuration files" in *Connecting Apama Applications to External Components*.
`-j | --java`	Enables support for Java applications, which is needed to inject a Java application or plug-in using `engine_inject -j`.

`-J` *option* `| --javaopt` *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` or `--javaopt` option multiple times in the same `correlator` command line. For example:
	`-J "-Da=value1" -J "-Db=value2" -J "-Xmx400m"`
	You can use `-J` as a prefix. In this case, you have to specify it without a space: `-J`*option*. For example:
	`-J-Dproperty=value`
	You can also specify JVM options in a YAML configuration file. If the same JVM option is

Option	Description
	specified in both the command line and the configuration file, the command line takes precedence. See also "Specifying JVM options" on page 117. **Note:** It is not possible to configure the correlator's system classpath using the `CLASSPATH` environment variable. We recommend that you set the classpath on a per-plug-in basis, for example, in the descriptor file for an EPL plug-in (see "Specifying the classpath in deployment descriptor files" in *Developing Apama Applications*) or in the configuration file for a connectivity plug-in, JMS or distributed MemoryStore library. For cases where you really need to set the global system classpath, you can use `-J-Djava.class.path=`*`path`*. When you use this option to pass the classpath to the JVM, Apama prepends the correlator-internal JAR files to the path you specify.
`--inputLog` *`file`*	Specifies the path of the input log file. The 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 try to diagnose the problem. An input 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. See "Replaying an input log to diagnose problems" on page 203.
`--configLog` *`file`*	Specifies the path to the configuration log file. The configuration log file contains the correlator's configuration, that is, the contents of all YAML configuration files and properties files as well as the correlator startup arguments and environment. The configuration log is a good way to capture useful diagnostic information from the correlator's startup in performance-critical situations where it is not acceptable to enable the input log.

Option	Description
`-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 using an input log to capture events and messages coming into a correlator. If you need to reproduce correlator behavior from that input log, you will want the correlator to generate the same random numbers as it generated when the original input was captured.
`--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` tool in conjunction with the status messages in the main correlator log file.
`-g | --nooptimize`	Disables correlator optimizations that hinder debugging. Specify this option when you plan to run the `engine_debug` tool. You cannot run the `engine_debug` tool if you did not specify the `-g` option when you started the correlator. Software AG Designer automatically uses the `-g` option when it starts a correlator from a debug launch configuration. However, if you are connecting to an externally-started correlator, and you want to debug in that correlator, you must ensure that the `-g` option was specified

Option	Description
	when the externally-started correlator was started.
`-P | -Penabled=`*`boolean`*	`-P` or `-Penabled=true` 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*. 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. You can also enable and configure correlator persistence using a YAML configuration file. See "Configuring persistence in a YAML configuration file" on page 116 for details. `-Penabled=false` disables correlator persistence, overriding any value that was specified in a YAML configuration file.
`-PsnapshotIntervalMillis=`*`interval`*	Specifies the period between persistence snapshots. The default is 200 milliseconds.
`-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.
`-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.
`-PstoreName=`*`file`*	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

Option	Description
	`-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 input 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 input log when it restarts a persistence-enabled correlator. The default is that the correlator does copy the recovery datastore to the 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 input log.
`--pidfile` *file*	Specifies the name of the file that contains the process identifier. This file is created at process startup and can be used, for example, to externally monitor or terminate the process. The correlator will remove that file after a clean shutdown. It is recommended that the file name includes the logical name of the correlator and/or port number to distinguish different correlators (for example, `my-correlator-15903.pid`). **CAUTION:** If the correlator process is terminated uncleanly or if the operating system is restarted, the pidfile usually remains on disk. However, the process identifier it contains is no longer valid

Option	Description
	or may match some other newly started process. To prevent mistakenly sending signals to the wrong process, ensure that the pidfile is explicitly deleted after a restart of the operating system or after an unclean termination of the correlator. Alternatively, you can configure the pidfile to be written to a transient location such as `/run` that is automatically deleted when the operating system is started. The correlator takes an exclusive lock on the pidfile while it is running. This means that if you have another correlator running using the same pidfile, the second correlator will fail to start up. You should not run multiple correlators from the same configuration using the same pidfile. In other cases, existing pidfiles will be overwritten, even if they contain a process identifier of a running process.
`--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 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. Using the compiled runtime requires that the binutils package is installed on the Linux system. Other than performance, the behavior of the two runtimes is the same except for the following:

Option	Description
	■ 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. **Note:** If you are using both correlator persistence and the compiled runtime (`--runtime compiled` option), we recommend the use of the `--runtime-cache` option to improve recovery times.
`--runtime-cache` *dir*	Enables caching of compiled runtime objects in the specified directory. Subsequent injections of the same files to any correlator using that cache will be quicker. For more information, see "Injection time of compiled runtime" on page 100.
`--frequency` *num*	Instructs the correlator to generate clock ticks at a frequency of *num* 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 *num* above the rate at which your operating system can generate its own clock ticks internally. On UNIX and some Windows machines this is 100 and on other Windows machines it is 64.
`-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 100.

Option	Description
`--config file`	Used to configure the correlator. Specifies one of the following: ■ The name of a `.properties` file. See also "Using properties files" on page 108. ■ The name of a `.yaml` file. See also "Using YAML configuration files" on page 101. ■ The name of a directory containing `.yaml` files and `.properties` files. In this case, the files are processed in alphabetical order. ■ A semicolon-delimited list of `.properties` files, `.yaml` files and directories. This option can be specified multiple times. The options are processed in the same sequence in which they are specified on the command line. If multiple `.yaml` files are specified, they are merged together based on the contents of the top-level maps they each contain. For example, if two files have a top-level map called `connectivityPlugins`, the merged document has a single `connectivityPlugins` map with all keys and values. Both maps, however, must not contain the same keys, otherwise an error will occur. For more information, see "Configuring the correlator" on page 101.
`-Dkey=value`	Specifies a value for a substitution property to be used by the Apama configuration files. The `-D` arguments always take precedence over any arguments defined in a `.properties` file. Therefore, they are processed before any `--config` arguments, regardless of the order on the command line. Using this option is similar to specifying the substitution using a `.properties` file for YAML (see also "Using properties files" on page 108). Correlator-integrated messaging for JMS and the distributed MemoryStore have their own properties files, which are Spring files. Keep in mind that any properties that are specified with

Option	Description
	the `--config file` or `-Dkey=value` option take precedence and override any properties already defined in a Spring properties file. They will *not* take effect for any properties that were not already defined in a Spring properties file. See also "Configuration files for JMS" in *Connecting Apama Applications to External Components* and "Configuration files for distributed stores" in *Developing Apama Applications*. `-D` properties are not related to Java system properties. If your intention is to set a Java system property, use `-J-Dkey=value` (and not `-Dkey=value`).
`--applicationLogLevel level`	Specifies the level of information that the correlator sends to the EPL log file. The log level must be one of the following (in increasing order of verbosity): `OFF`, `CRIT`, `FATAL`, `ERROR`, `WARN`, `INFO`, `DEBUG`, `TRACE`. The default is the current setting of the `-v` (or `--loglevel`) option. You can also specify the log level in a YAML configuration file. See "Setting EPL log files and log levels in a YAML configuration file" on page 111 for details.
`--applicationLogFile file`	Specifies the path of the EPL log file that the correlator writes messages to. The default is the current setting of the `-f` (or `--logfile`) option. You can also specify the EPL log file in a YAML configuration file. See "Setting EPL log files and log levels in a YAML configuration file" on page 111 for details.
`--shutdownTimeoutSecs number`	Specifies the maximum time allowed for the correlator to shut down after all persistence activities have been completed. When this time has elapsed, the correlator shuts down forcibly, ignoring any transport or plug-in shutdown activities. This option is especially useful to prevent indefinite hangs caused by plug-ins. If this option is not provided, a default value of 90 seconds is used.

Option	Description
`--python=boolean`	`--python` or `--python=true` enables Python support. By default, Python support is not enabled. You can also enable Python support using a YAML configuration file. For more information, see "Using Python plug-ins" in *Developing Apama Applications*. `--python=false` disables Python support, overriding any value that was specified in a YAML configuration file.

Exit status

The `correlator` executable returns the following exit values:

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

Specifying log filenames

A correlator can send information to the following log files:

- **Main correlator log file.** Upon correlator startup, the default behavior is that the correlator logs status information to `stdout`. To send this information to a file, specify the `-f file` or `--logfile file` option and replace *`file`* with a log filename. The format for specifying a log filename is described below.

  Before you specify a log filename, you should consider your log rotation policy, which can determine what you want to specify for the log filename. See "Rotating correlator log files" on page 185.

- **EPL log files.** You can create log files for packages, monitors, and events in your application. The format you use to specify these log filenames is the same as for the main correlator log file. For details about how to create EPL log files, see "Setting EPL log files and log levels in a YAML configuration file" on page 111 and "Setting EPL log files and log levels dynamically" on page 182.

- **Correlator input log file.** When you start a correlator, you can specify the `--inputLog file` option so that the correlator maintains a special log file for all inputs. Again, before you specify a log filename, you should consider the rotation policy for your input log files. See "Rotating an input log file" on page 204.

Note:
The correlator input log file is slightly different from the other log files, and it is not intended to be read by a human.

- **Correlator configuration log file.** When you start a correlator, you can specify the `--configLog` *`file`* option so that the correlator maintains a special log file for all of the correlator's configuration.

The format for specifying a log filename is as follows:

```
file[${START_TIME}][${ROTATION_TIME}][${ID}][${PID}].log
```

The following table describes each part of a log filename specification. You cannot include spaces. You can separate the parts of the filename specification with underscores. You can specify `${START_TIME}`, `${ROTATION_TIME}`, `${ID}` and/or `${PID}` in any order. For examples, see "Examples for specifying log filenames" on page 92.

Element	Description
`file`	Replace *`file`* with the name of the file that you want to be the log file. If you specify the name of a file that exists, the correlator overwrites it on Windows or appends to it on UNIX. Required. If you also specify `${START_TIME}` and/or `${ROTATION_TIME}` and/or `${ID}` and/or `${PID}`, the correlator prefixes the name you specify to the time the correlator was started and/or the time the log file was rotated (logging to a new file began) and/or an ID beginning with "001" and/or the process ID. Be sure to specify a location that allows fast access.
`${START_TIME}`	Tag that indicates that you want the correlator to insert the date and time that it started into the log filename. Optional, however you probably want to always specify either this option or `${ROTATION_TIME}` to avoid overwriting log files.
`${ROTATION_TIME}`	Tag that indicates that you want the correlator to insert the date and time that it starts sending messages to a new log file into the log filename. Optional. If you specify `${ROTATION_TIME}` and this log filename specification appears in a correlator start-up command then the name of the initial log file contains the time the correlator started.
`${ID}`	Tag that indicates that you want the correlator to insert a three-digit ID into the log filename. The ID that the correlator inserts first is "001".

Element	Description
	Optional. The log ID increment is related only to rotation of log files. See "Rotating correlator log files" on page 185 and "Rotating an input log file" on page 204. The ID allows you to create a sequence of log files. Each time the log file is rotated, the correlator increments the ID. A sequence of log files have the same name except for the ID. If you also specify `${ROTATION_TIME}` then a sequence of log files have the same name except for the rotation time and the ID. Restarting the correlator always resets the ID portion of the log filename to "001".
`${PID}`	Tag that indicates that you want the correlator to insert the process ID into the log file name. Optional. The process ID will be constant for the lifetime of the process. Therefore, if you start multiple processes with the same arguments, they get different file names.

If you plan to rotate log files then be sure to specify `${ROTATION_TIME}` or `${ID}`. You can also specify both.

Examples for specifying log filenames

This topic provides examples of specifying log filenames. The format for specifying a log filename is the same in the following cases:

- Starting the correlator and specifying a main correlator log file with the `--logfile` option.
- Starting the correlator and specifying a correlator input log file with the `--inputLog` option.
- Invoking `engine_management --setLogFile` to rotate the main correlator log.
- Invoking `engine_management --setApplicationLogFile` to create an EPL log file for a package, monitor or event.

The following specifies that the name of the main log is "correlator_status.log":

```
--logfile correlator_status.log
```

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

```
--logfile correlator_status_${START_TIME}.log
```

The above command opens a log with a name something like the following:

```
correlator_status_2015-03-12_15:12:23.log
```

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

```
correlator_status_${START_TIME}_${ID}.log
```

The initial name of the log file is something like the one on the first line below. If you then rotate the log file then the correlator closes that file and opens a new file with a name like the one on the second line:

```
correlator_status_2015-03-12_15:12:23_001.log
correlator_status_2015-03-12_15:12:23_002.log
```

To specify an EPL log filename for messages generated in `com.example.mypackge`, you can specify the log filename as follows:

```
mypackage_${ID}_${ROTATION_TIME}.log
```

With that specification, messages generated in `com.example.mypackage` will go to a file with a name such as the one on the first line below. The time in the initial EPL log filename is the time that the initial log file is created. If you rotate the logs every 24 hours at midnight then the names of subsequent EPL log files will be something like the names in the second and third lines below.

```
mypackage_001_2015-03-21_18:42:06.log
mypackage_002_2015-03-22_00:00:00.log
mypackage_003_2015-03-23_00:00:00.log
```

If you want to run multiple correlators with the same arguments but with separate log files, you can use the process ID to differentiate them:

```
--logfile correlator_${PID}.log
```

The above command will produce a log file with a name such as the following, where each correlator will have a unique log file:

```
correlator_23487.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 --inputLog input_\${START_TIME}_\${ID}.log
```

Descriptions of correlator status log fields

The correlator sends information to the main correlator log file every five seconds (the default behavior) or at an interval that you specify with the `--logQueueSizePeriod` option when you start the correlator.

There are one or two lines in the log file, depending on whether persistence is enabled or not:

- The line starting with `Correlator Status:` is always shown. See "Correlator status" on page 94 for detailed information on the log fields that are shown in this line.
- The line starting with `Persistence Status:` is only shown when persistence has been enabled. See "Persistence status" on page 96 for detailed information on the log fields that are shown in this line.

Note:

Correlators with correlator-integrated messaging for JMS enabled send additional information to the main log file of the correlator. For details on this information, see "Logging correlator-integrated messaging for JMS status" in *Connecting Apama Applications to External Components*.

Correlator status

When logging at `INFO` level, this information contains the following:

```
Correlator Status: sm=11 nctx=1 ls=60 rq=0 iq=0 oq=0 icq=0 lcn="<none>" lcq=0 lct=0.0
rx=5 tx=17 rt=31 nc=2 vm=325556 pm=81068 runq=0 si=915.3 so=0.0 srn="<none>" srq=0 jvm=0
```

Where the fields have the following meanings (see "List of correlator status statistics" on page 143 for more information):

Field	Meaning	Trend
`sm`	The number of monitor instances, also known as sub-monitors.	Steady
`nctx`	The number of contexts in the correlator, including the main context.	Steady
`ls`	The number of listeners in all contexts. This includes on statements and active stream source templates.	Steady
`rq`	The sum of routed events on the route queues of all contexts.	Low
`iq`	The number of executors on the input queues of all contexts. As well as events, this can include clock ticks, spawns, injections and other operations. A context in an infinite loop will grow by 10 per second due to clock ticks. Every context has an input queue, which by default is a maximum of 20,000 entries.	Low
`oq`	The number of events waiting on output queues to be dispatched to any connected external consumers/receivers.	Low
`icq`	The number of events on the input queues of all public contexts. See also "About context properties" in *Developing Apama Applications* for information on the receiveInput flag.	Low
`lcn`	The name of the slowest context. This may or may not be a public context.	None

Field	Meaning	Trend
`lcq`	The number of events on the slowest context's queue, as identified by the name of the slowest context.	Low
`lct`	For the context identified by the slowest context name, this is the time difference in seconds between its current logical time and the most recent time tick added to its input queue.	Low
`rx`	The number of events that the correlator has received from external sources since the correlator started. This includes connectivity plug-ins, correlator-integrated JMS, engine_send, other correlators connected with engine_connect, dashboard servers, the IAF, and events that are not parsed correctly. This number excludes events sent within the correlator from EPL monitors or EPL plug-ins.	Increasing
`tx`	The number of events that have been delivered to external consumers/receivers. This counts for each external consumer/receiver an event is sent to. It counts the number of deliveries of events.	Increasing
`rt`	The number of events that have been routed across all contexts since the correlator was started.	Increasing
`nc`	The number of external consumers/receivers connected to receive emitted events. This includes connectivity plug-ins, correlator-integrated JMS, engine_receive, or correlators connected using engine_connect.	Steady
`vm`	Virtual memory in kilobytes.	Steady
`pm`	Physical memory in kilobytes.	Steady
`runq`	The number of contexts on the run queue. These are the contexts that have work to do but are not currently running.	Low
`si`	The number of pages per second that are being read from swap space. If this is greater than zero, it may indicate that the machine is under-provisioned, which can lead to reduced performance, connection timeouts and other problems. Consider adding more memory, reducing the number of other processes running on the machine, or partitioning your Apama application across multiple machines.	Low
`so`	The number of pages per second that are being written to swap space. If this is greater than zero, it may indicate that the machine is under-provisioned, which can lead to reduced performance, connection timeouts and other problems. Consider adding more memory, reducing the number of other processes running on the machine, or partitioning your Apama application across multiple machines.	Low
`srn`	The name of the consumer/receiver with the largest number of incoming events waiting to be processed. This is the slowest	None

Field	Meaning	Trend
	non-context consumer/receiver of events, which can be an external receiver or an EPL plug-in.	
`srq`	The number of events on the slowest consumer's/receiver's queue, as identified by the name of the slowest consumer/receiver.	Low
`jvm`	The sum of all memory used by the Java virtual machine (JVM) which is embedded in the correlator (that is, the used heap memory, the used non-heap memory, and the used buffer pool memory). The value is in megabytes. If the JVM is disabled, the value will be 0.	Steady

Persistence status

If persistence is enabled, information such as the following is also shown when logging at `INFO` level:

```
Persistence Status: numSnapshots=25 lastSnapshotTime=1516203192
snapshotWaitTimeEwmaMillis=0.029071 commitTimeEwmaMillis=3.459181
lastSnapshotRowsChangedEwma=8
```

Where the fields have the following meanings (see "List of correlator status statistics" on page 143 for more information):

Field	Meaning	Trend
`numSnapshots`	The number of persistence snapshots taken since the correlator started.	Increasing
`lastSnapshotTime`	The UNIX timestamp of the last completed snapshot.	Increasing
`snapshotWaitTimeEwmaMillis`	An exponentially weighted moving average (EWMA) of the time in milliseconds taken to wait for a snapshot.	Varies
`commitTimeEwmaMillis`	An exponentially weighted moving average (EWMA) of the time in milliseconds taken to commit to the database.	Varies
`lastSnapshotRowsChangedEwma`	An exponentially weighted moving average (EWMA) of the number of rows changed per snapshot.	Varies

Text internationalization in the logs

The information given here applies for the correlator, and also for the IAF and dashboard servers.

The strings in the logs are encoded in two different ways, depending on whether they are written to a log file or to the console:

- **Log file**

  Apama log files are always encoded as UTF-8. Any non-ASCII strings in a log file (such as Chinese characters, German umlauts, some currency symbols, and so on) will only look correct if your text editor is configured to view the text as UTF-8. This is irrelevant on most Linux installations, where the default encoding is usually UTF-8.

- **Console**

  If the Apama component is logging to the console, the output is encoded according to the locale of your system. In practice, this means that any strings that are logged to the console (including non-ASCII strings) will look correct.

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 *Developing Apama Applications*.

Description of slow receivers

The correlator uses two strategies to detect slow receivers: time-based, and size-based.

Time-based (maxOutstandingSecs) slow consumer detection

Events that the correlator sends to Apama clients, IAFs or other correlators are acknowledged by the receiver after the event has been delivered to the receiver. 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.

For most systems, and assuming that the underlying network connections are not prone to drop-outs, the default setting of 10 seconds is usually adequate.

You can control the length of time within which the receiver must acknowledge an event before it is marked as a slow receiver. To do so, you can specify `maxOutstandingSecs` in the YAML configuration file that is used when starting the correlator. See "YAML configuration file for the correlator" on page 102.

For example, if you specify `maxOutstandingSecs: 15.0` in the YAML configuration file, the correlator marks a receiver as slow if the correlator does not receive an acknowledgment within 15 seconds. If you do not specify this element, 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 `maxOutstandingSecs`, the slow designation takes effect between 1 and 1.5 times the value of `maxOutstandingSecs`.

Size-based (maxOutstandingKb) slow consumer detection

`maxOutstandingKb` can also be specified in the YAML configuration file that is used when starting the correlator. See "YAML configuration file for the correlator" on page 102.

The correlator keeps track of the events that have been sent to each receiver but have not yet been acknowledged, based on the amount of memory taken up by those events. The correlator marks a receiver as being slow if the size of the events waiting to be acknowledged goes above the `maxOutstandingKb` limit, which is 10MB by default.

Note that the size-based slow consumer detection operates completely independently of the time-based (`maxOutstandingSecs`) slow consumer detection.

It is rare for the size-based limit to be exceeded unless the events being transmitted are very large.

Connectivity plug-ins slow consumer detection

Connectivity chains use time-based slow consumer detection similar to `maxOutstandingSecs`. The size of the time window is currently set at 10 seconds (not configurable). A chain that has not processed a message for more than 10 seconds is logged as slow, but is not disconnected.

How frequently slow receivers occur

In practice, sending acknowledgments 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 acknowledgment 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.

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.

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 main 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.

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's Data Player in Software AG Designer 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`.

See also "Understanding time in the correlator" in *Developing Apama Applications*.

Injection time of compiled runtime

Injection times for systems using the compiled runtime can be very long - significantly longer than if using only the interpreted runtime. Subsequent injection times can be improved by using the `--runtime-cache` *`dir`* option, which specifies a directory where the correlator can cache the compilation state (see "Starting the correlator" on page 78). This stores the results of compiling EPL code on disk to be used for subsequent injections of the same code.

The compiled EPL code is specific to the system on which it was compiled and the version of Apama that was used to compile it. This means that while a cache can be moved or shared between machines to improve startup on a new machine, it must be identical to the original. Otherwise, the cached version cannot be used and it must be recompiled

An injection is able to use a cached version of a previous injection if all of the following are the same as in a previous injection:

- The EPL source code.

- The source code of all files that contain any type that an injection depends on.
- The correlator version.
- The host operating system.
- The CPU model.

The results of injections can be affected by any of the above. Therefore, if any change occurs, the correlator will re-compile the EPL.

The cache is designed to be used to speed up re-injection on production systems and not for quicker development cycles, which should typically use the interpreter for faster injection times. If there are identical user acceptance testing (UAT) and production environments, then the UAT environment can prime a cache which can then be used by the production correlator to improve initial startup times. However, the two systems must be identical. The strings used to disambiguate systems are logged at correlator startup when using the compiled runtime and can be used to compare the systems.

The cache contents are never removed by the correlator. If you change your source, correlator versions or platform, then the cache may grow and contain stale items which are no longer needed. If cache sizes become a problem, then we either recommend deleting the entire cache, or just the cache files with the oldest timestamps. The correlator will transparently recompile any needed files which are missing from the cache.

Configuring the correlator

You can configure the correlator using YAML configuration files and properties files. See the topics below for detailed information on these types of files, and on how to configure certain aspects of the correlator.

YAML configuration files are also used to configure connectivity plug-ins. See "Configuration file for connectivity plug-ins" in *Connecting Apama Applications to External Components* for further information.

Using YAML configuration files

You can specify one or more YAML files using the `--config` option when you start the correlator. See “Starting the correlator” on page 78.

For detailed information on YAML, see http://www.yaml.org/spec/1.2/spec.html. A quick overview is given below.

YAML configuration files can contain maps, lists or simple values:

- A map contains a string key, followed by a colon and space, followed by a value (which can be a map, a list or a simple value). Typically, an entry with a simple value is written on one line, and collections (maps and lists) are written on a following line with indentation.
- A list contains a number of values, each of which is written on a new line with a preceding dash and space ("- ").

- A simple value includes a string or number. It is typically written on a single line. A string may be enclosed in quotes if needed, but this is not mandatory.

  Some characters in YAML have special significance at certain positions. For example, a value ending with a colon (:) is treated as a key in a dictionary, so if you want a string value to end with a literal colon (:), you should quote it.

- Nesting is expressed using spaces to indent different levels of object. Tabs are forbidden in YAML files, all indentation must be performed using spaces.

For example:

```
myTopLevelMap:
  mykey: myvalue
  mylist:
    - item 1 # comment
    - item 2
    - "a quoted string value"
```

YAML is a superset of JSON. Thus, any valid JSON is also usable in the YAML configuration file. This is helpful if there is ambiguity in the way YAML expresses configuration.

YAML documents should be saved with the standard UTF-8 character encoding.

YAML configuration file for the correlator

The following sample shows the format of a YAML configuration file for the correlator.

```
correlator:
  javaApplicationSupport: false
  pythonSupport: true
  randomSeed: number
  initialization:
    list: [ ... ]
  persistence:
    enabled: false
    snapshotIntervalMillis: number
    adjustSnapshot: true
    storeLocation: path
    storeName: string
    clear: false
  runtime: interpreted
  runtimeCacheDir: path
  licenseFile: path
  jmsConfigDir: path
  distMemStoreConfigDir: path
  inputLogFile: path
  externalClock: false
  timerFrequency: 10
  truncateLogFile: false
  disableOptimizations: false
  includeDatabaseInInputLog: true
  statusLogIntervalSecs: 5.0
  inputQueueSize: 20000
  logFile: path
  shutdownTimeoutSecs: number
```

```
server:
  pidFile: path
  port: 15903
  name: string
  bindAddress: [...]
  permittedRootURIs: [...]
  allowClient: [...]
  maxOutstandingSecs: 10.0
  maxOutstandingKb: 10240
  disconnectSlowConsumers: false

jvmOptions: [...]

eplLogging:
  .root:
    level: loglevel
    file: path
  string:
    level: loglevel
    file: path
correlatorLogging:
  string:
    level: loglevel

environment:
  string: string

engineConnect:
  - sourceHost: string
    sourcePort: integer
    channels:
      - string
    mode: string
    disconnectIfSlow: boolean

includes: [...]

# For configuring EPL plug-ins:
eplPlugins: [...]

# For configuring connectivity plug-ins:
connectivityPlugins: [...]
startChains: [...]
dynamicChainManagers: [...]
dynamicChains: [...]
```

For paths, we recommend using replacements like `${PARENT_DIR}` or `${APAMA_WORK}` to make the configuration files portable. See also the list of predefined properties in "Using properties files" on page 108.

`[...]` denotes that this can be a list.

You can use multi-line and single-line syntax for maps (for example, in the `eplLogging` case).

Many of the elements in the above sample configuration file correspond to command-line options of the `correlator` executable and have the same syntax and options. See the tables below, and see the descriptions of these command-line options in "Starting the correlator" on page 78 for detailed information.

Note:

If an element in a YAML configuration file is defined more than once, the first definition that is loaded by the correlator takes precedence over all definitions that are loaded later. An `INFO` message is then logged for all later definitions that are being ignored. Any corresponding options that are specified directly on the command line, however, take precedence over any other later definitions.

The following table lists the elements that can be specified in the `correlator` section of the YAML configuration file (see the separate table below for the `correlator/persistence` section).

Note:

When developing using Software AG Designer, add any elements you need to set to the `config/CorrelatorConfig.yaml` file. See also the description of the **Configuration** option in "Correlator arguments" in *Using Apama with Software AG Designer*.

This element	corresponds to this command-line option
`javaApplicationSupport`	`-j | --java`
`pythonSupport`	`--python`
`randomSeed`	`-XsetRandomSeed`
`runtime`	`--runtime`
`runtimeCacheDir`	`--runtime-cache`
`licenseFile`	`-l | --license`
`jmsConfigDir`	`--jmsConfig`
`distMemStoreConfigDir`	`--distMemStoreConfig`
`inputLogFile`	`--inputLog`
`externalClock`	`-Xclock | --externalClock`
`timerFrequency`	`--frequency`
`truncateLogFile`	`-t | --truncate`
`disableOptimizations`	`-g | --nooptimize`
`includeDatabaseInInputLog`	`--noDatabaseInReplayLog`
`statusLogIntervalSecs`	`--logQueueSizePeriod`
`inputQueueSize`	`--inputQueueSize`
`logFile`	`-f | --logfile`
`shutdownTimeoutSecs`	`--shutdownTimeoutSecs`

The following table lists the elements that can be specified in the `correlator/persistence` section of the YAML configuration file:

This element	corresponds to this command-line option
`enabled`	`-P | -Penabled=true`
`snapshotIntervalMillis`	`-PsnapshotIntervalMillis`
`adjustSnapshot`	`-PadjustSnapshot`
`storeLocation`	`-PstoreLocation`
`storeName`	`-PstoreName`
`clear`	`-Pclear`

The following table lists the elements that can be specified in the `server` section of the YAML configuration file:

This element	corresponds to this command-line option
`pidFile`	`--pidfile`
`port`	`-p | --port`
`name`	`-N | --name`
`maxOutstandingSecs`	`-m | --maxoutstandingack`
`maxOutstandingKb`	`-M | --maxoutstandingkb`
`disconnectSlowConsumers`	`-x | -- qdisconnect`

The following table lists the remaining sections that can be specified in the YAML configuration file:

This element	corresponds to this command-line option
`jvmOptions`	`-J | --javaopt`
`eplLogging/.root/level`	`--applicationLogLevel`
`eplLogging/.root/file`	`--applicationLogFile`
`correlatorLogging/level`	`-v | --loglevel`

Other elements are described in topics under "Configuring the correlator" on page 101, and this also includes additional information for some of the elements that correspond to the command-line options. These are:

- In the `correlator` section:

  - `initialization`. See "Deploying Apama applications with a YAML configuration file" on page 118.
  - `persistence`. See "Configuring persistence in a YAML configuration file" on page 116.
- In the `server` section:
  - `bindAddress` and `permittedRootURIs`. See "Binding server components to particular addresses" on page 110.
  - `allowClient`. See "Ensuring that client connections are from particular addresses" on page 110.
- `jvmOptions`. See "Specifying JVM options" on page 117.
- `eplLogging`. See "Setting EPL log files and log levels in a YAML configuration file" on page 111.
- `correlatorLogging`. See "Setting correlator and plug-in log files and log levels in a YAML configuration file" on page 113.
- `environment`. See "Setting environment variables for Apama components" on page 111.
- `engineConnect`. See "Setting up connections between correlators in a YAML configuration file" on page 117.
- `includes`. See "Including YAML configuration files inside another YAML configuration file" on page 106.

The `eplPlugins` element is used for configuring EPL plug-ins. There is no equivalent command-line option. See "Writing EPL Plug-ins in Python" in *Developing Apama Applications* for detailed information.

The following elements are used for configuring connectivity plug-ins. There are no equivalent command-line options. See "Using Connectivity Plug-ins" in *Connecting Apama Applications to External Components* for detailed information.

- `connectivityPlugins`
- `startChains`
- `dynamicChainManagers`
- `dynamicChains`

Including YAML configuration files inside another YAML configuration file

Instead of specifying a list of configuration files on the command line with the `--config` option, you can also specify these files in the `includes` section of another configuration file. For example:

```
includes:
   - myCodec.yaml
   - myTransport.yaml
```

When the configuration file that is specified with the `--config` option is processed, the list in the `includes` section is expanded recursively, and the information inside the included configuration files is merged into the main configuration. The same rules are used as for the `--config` option; for details, see the description of that option in "Starting the correlator" on page 78.

You do not need to worry about multiple file inclusions (such as cyclical or diamond references), but you must still be careful not to have duplicate keys in top-level maps.

Using an `includes` section can be useful for specifying a connectivity chain in a modular way. For example, you may have a main configuration file with the following contents:

```
startChains:
  myChain:
     - apama.eventMap
     - MyCodec
     - MyTransport
includes:
   - myCodec.yaml
   - myTransport.yaml
```

where the included files have the following contents:

- `myCodec.yaml`:

  ```
  connectivityPlugins:
    MyCodec:
      classpath: ${myJarVersion}
      class: com.example.my.Codec
  includes:
     - propDir
  ```

  You can include further files in an included file. Instead of a file name, you can also specify the name of a directory in the `includes` section. All files that can be found in this directory are then included.

  The `includes` section in the above example assumes that there is a directory named `propDir` which contains a `jarVersions.properties` file with the following content:

  ```
  myJarVersion=myJar316.jar
  ```

- `myTransport.yaml`:

  ```
  connectivityPlugins:
    MyTransport:
      classpath: myOther.jar
      class: com.example.my.Transport
  ```

Properties defined in included files are only valid for later files. For example, the deployment with the following files will work:

- `1.yaml`:

  ```
  includes:
     - my.properties
     - 2.yaml
  ```

- `my.properties`:

```
3=3.yaml
```

- `2.yaml`:

```
includes:
   - ${3}
```

The deployment with the following file, however, will not work:

- `1.yaml`:

```
includes:
   - my.properties
   - 2.yaml
   - ${3}
```

  In this case, you should specify the `my.properties` file as an argument with the `--config` option, in addition to the `1.yaml` file.

See "Using properties files" on page 108 for more information.

Using properties files

Properties files (with the file extension `.properties`) can be used to either specify values for substitution variables in YAML files (see also "Using YAML configuration files" on page 101) or to configure EPL. A properties file must be either in ISO-8859-1 encoding or in UTF-8 encoding if it begins with an UTF-8 byte order mark (BOM).

You can specify one or more properties files using the `--config` option when you start the correlator. See "Starting the correlator" on page 78. The properties files are processed in the order in which they appear on the command line. Each properties file can refer to properties that have already been defined by a previously processed properties file, using `${varname}` syntax.

Note:
If the same property is defined more than once, the first definition that is loaded by the correlator takes precedence over all definitions that are loaded later. Messages are then logged for all later definitions that are being ignored.

The properties file format is the same as the standard Java `.properties` file format, with `#` for comments and `\\` used to escape any occurrences of `\`. For example:

```
# my comment line
myplugin.mykey=c:\\my directory
```

Properties files are by default parsed using the ISO-8859-1 encoding (Latin-1). To use characters from other character sets, you have two options. You can use a `\uXXXX` escape sequence, which uses the UCS-2 character codes. This allows you to encode any character from the Basic Multilingual Plane (BMP). However, we do not support UTF-16 surrogate pairs to access characters outside the BMP. Alternatively, you can provide a UTF-8 BOM as the first characters of the file. In this case, you can use any Unicode sequence in UTF-8 directly in the file without any escaping.

You can use the following predefined properties:

Property	Description
`${PARENT_DIR}`	The absolute normalized path of the directory containing the properties file or YAML file currently being processed.
`${APAMA_HOME}`	The path to the Apama installation.
`${APAMA_WORK}`	The path to the Apama work directory.
`${$}`	The literal $ sign.

All properties are applied to all YAML files, although conventionally, there is often a `.properties` file named the same as each `.yaml` file. Properties files, however, are not actually tied in to YAML files in any way. It is therefore recommended that you prefix each property key with a unique string, such as the identifier of the chain to which it applies.

Instead of using a properties file or in addition to using a properties file, you can also use the `-D` option of the `correlator` executable. See "Starting the correlator" on page 78.

Properties defined either through properties files or on the command line are available to EPL via the Management interface. For more details, see "Using the Management interface" in *Developing Apama Applications* and the *API Reference for EPL (ApamaDoc)*.

Correlator-integrated messaging for JMS and the distributed MemoryStore have their own properties files, which are Spring files. Keep in mind that any properties that are specified with the `--config file` or `-Dkey=value` option of the `correlator` executable take precedence and override the properties defined in a Spring properties file. See also "Configuration files in JMS" in *Connecting Apama Applications to External Components* and "Configuration files for distributed stores" in *Developing Apama Applications*.

Runtime parameterization of configuration

Some components allow parameterization of configuration at runtime, for example, connectivity plug-in chains created from a manager or from EPL. Variables which must be replaced at runtime are denoted with `@{varname}` instead of `${varname}`. Strings using this syntax may need to be escaped with double quotes in the YAML configuration file. Instead of being supplied via the command line or via properties files, these substitutions are provided within the subsystem using that part of the configuration. See also "Creating dynamic chains from EPL" in *Connecting Apama Applications to External Components*.

Specifying the correlator port number

You can specify the port on which the correlator should listen for monitoring and management requests. To do so, you specify a `port` definition in the `server` section of the YAML configuration file. For example:

```
server:
  port: 15903
```

This is identical to specifying the `--port` option when starting the correlator. See also "Starting the correlator" on page 78.

If the port is specified with the `--port` option when starting the correlator, this value is used. Else, if it is specified in the YAML configuration file, that value is used. If a port is not specified at all, the default value 15903 is used.

Binding server components to particular addresses

To bind Apama server components to a particular address or set of addresses, specify a `bindAddress` definition for each address. Specify this in the `server` section of the YAML configuration file. For example:

```
server:
  bindAddress:
    - 127.0.0.1:15903
    - 10.0.0.1
```

You can specify as many `bindAddress` definitions 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 YAML configuration files if you want to restrict connections according to only addresses.

If you do not specify a YAML configuration file when you start the correlator, or there are no `bindAddress` definitions in the YAML configuration file, the Apama components bind to the wildcard address (`0.0.0.0`).

Using the correlator web interface with non-default addresses

The correlator web interface sometimes uses the client's host header for redirects. For security, this is checked against a list of accepted *`host:port`* aliases for this component. By default, all of the addresses of the machine on the default or overridden correlator port are part of this list.

If the correlator has a custom bind address configuration, or is being hidden behind another component which is redirecting the web interface, the default list of permitted addresses may need to be overridden. This can be done with the `permittedRootURIs` configuration entry:

```
server:
  permittedRootURIs:
     - ${EXTERNAL_HOSTNAME}:${EXTERNAL_PORT}
```

This can either be a single string or a list of multiple permitted URIs. This list replaces the automatically calculated defaults, it does not append.

Ensuring that client connections are from particular addresses

To ensure that client connections are from particular addresses, add one or more `allowClient` definitions to the YAML configuration file in the `server` section. For example:

```
server:
```

```
allowClient:
  - 127.0.0.1
  - 192.168.128.0/17
```

An `allowClient` definition 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 YAML 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 do not accept connections from any other IP addresses.

If you specify a YAML configuration file when you start the correlator, and if there are any `allowClient` definitions in the YAML 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 a YAML configuration file when you start the correlator, or there are no `allowClient` definitions in a YAML configuration file that you do specify, the Apama components accept connections from any client.

Important:
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.

Setting environment variables for Apama components

You can use the YAML configuration file to set environment variables. Put environment variable declarations in the `environment` section. For example:

```
environment:
  MY_ENV_VAR: myvalue
```

If you specify a YAML configuration file when you start the correlator, and if there are any environment variable entries in the YAML configuration file, then the Apama components use those environment variable settings. If you do not specify a YAML configuration file when you start the correlator, or there are no environment variable entries in a YAML configuration file that you do specify, the Apama components use the environment variable settings specified elsewhere.

Note:

You cannot use this feature to set variables such as `LD_PRELOAD` and `LD_LIBRARY_PATH` because UNIX dictates that they are set before the affected process starts execution. These environment variables should therefore be considered read-only.

Setting EPL log files and log levels in a YAML configuration file

This topic describes how to configure logging for individual EPL packages. For information about configuring the log level of the whole correlator and plug-ins running inside it, see "Setting correlator and plug-in log files and log levels in a YAML configuration file" on page 113.

You can configure per-package logging in several ways:

- Statically, in the YAML configuration file when starting the correlator, as described in this topic.
- Dynamically, using the following options of the `engine_management` tool:

  ```
  --setApplicationLogFile

  --setApplicationLogLevel
  ```

  See "Setting EPL log files and log levels dynamically" on page 182 for detailed information.
- Dynamically from within EPL. See "Using the Management interface" in *Developing Apama Applications* for detailed information.

To set log files and/or log levels for EPL root, packages, monitors, or events, specify entries in the `eplLogging` section of the YAML configuration file.

To set the default log file and level for the EPL root package, specify this in the following format:

```
eplLogging:
  .root:
     file: rootLogFilename
     level: 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:

```
eplLogging:
  .root:
     file: apama/root.log
     level: ERROR
```

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 this in the following format:

```
eplLogging:
  node:
     file: nodeLogFilename
     level: NODELOGLEVEL
```

Replace *`node`* with the name of the EPL package, monitor, or event for which you are setting the logging attribute. 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:

```
eplLogging:
  com.myCompany.Client:
    file: apama/Client.log
    level: DEBUG
```

```
  com.myCompany.Internal: { level: ERROR }
```

The above example shows both multi-line and single-line syntax. The single-line syntax is more compact when you are just setting either the log level or the file, but not both.

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 183.

When you set log attributes in the YAML configuration file, the rules for hierarchical logging apply. See "Setting EPL log files and log levels dynamically" on page 182.

If you pass a YAML configuration file to a correlator when you start that correlator and the configuration file contains an `eplLogging` 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 `eplLogging` section, then correlator initialization does not include any log settings except for the default correlator log.

Whether or not you specify a YAML configuration file when you start a correlator, any log settings you specify can be overwritten after initialization by invoking the `engine_management` tool and specifying the `--setApplicationLogFile` and/or `--setApplicationLogLevel` options. See "Managing EPL log levels" on page 183 and "Managing EPL log files" on page 185.

Note:

Regularly rotating log files and storing the old ones in a secure location may be important as part of your personal data protection policy. For more information, see "Recommendations for logging by Apama application code" in *Developing Apama Applications*.

Setting correlator and plug-in log files and log levels in a YAML configuration file

In a YAML configuration file, you can configure the log file and/or log level individually for plug-ins, as well as for components of the correlator itself. You can do this either for the whole correlator or for individual categories. This includes setting the log level for individual connectivity plug-ins and EPL plug-ins.

Note:

You can also set the log levels individually on the command line (see "Starting the correlator" on page 78). If a log level is specified on the command line, it overrides any setting in the YAML configuration file.

To set the log levels in the YAML configuration file, specify entries in the `correlatorLogging` section. The syntax for this section is:

```
correlatorLogging:
   .root:
      level: ROOTLOGLEVEL
      file: ROOTLOGFILE
   category: CATEGORYLOGLEVEL
```

You can specify either the log level directly, or as a map with the key level. These are synonymous. To set the log file, you must use the map syntax.

`.root` is used to specify the default log file/level for the whole correlator. We do not recommend specifying a log level higher than `INFO` for the default log level, since important information may be lost from the log files.

Valid log levels are `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL`, `CRIT` or `OFF`.

You can use replacement tokens in a log file. See "Specifying log filenames" on page 90 for more information.

The categories for which the logging can be configured are listed in the table below. Note that log categories are regarded as a hierarchy. For example, setting the log level for `apama` will be inherited by `apama.status`.

Example - the default log level for the whole correlator is set to `INFO`, and the log level for the connectivity plug-ins framework is set to `DEBUG`:

```
correlatorLogging:
   .root:
      level: INFO
   apama.connectivity: DEBUG
```

This category	controls the following
`apama.applicationEvents`	Correlator application event logging. Provides detailed output of the inner workings of the correlator, such as context-state changes, event triggering, spawning, routing, etc. To enable application event logging, you have to set the log level of this category to `DEBUG`. See also "Viewing garbage collection and application events information" on page 170.
`apama.connectivity`	Connectivity plug-ins - framework.
`apama.debughandler`	Correlator EPL debugger.
`apama.jms`	Correlator-integrated messaging for JMS.
`apama.messaging`	Internal messaging-related messages.
`apama.socket`	Socket-level communications.
`apama.status`	Correlator status lines.
`apama.streams`	Stream queries within EPL.
`apama.verboseGC`	Correlator garbage collection messages for all monitors. To enable garbage collection logging, you have to set the log level of this category to `DEBUG`. See also "Viewing garbage collection and application events information" on page 170.

This category	controls the following
`apama.verboseGC.`*`MonitorName`*	Correlator garbage collection messages for the specified monitor. To enable garbage collection logging, you have to set the log level of this category to `DEBUG`. See also "Viewing garbage collection and application events information" on page 170.
`com.apama.correlator.jms`	Correlator-integrated messaging for JMS.
`com.apama.adapters`	Correlator-integrated messaging for JMS - mapping rules.
`com.apama.jmon`	JMon framework. **Note:** Apama's in-process API for Java (JMon) is deprecated and will be removed in a future release.
`com.softwareag.connectivity`	Connectivity plug-ins - Java framework.
`connectivity.`*`PluginName`*`.` *`ChainName`*	Connectivity plug-ins - messages from a specific plug-in. Applies to connectivity plug-ins written in both C++ and Java.
`connectivity.`*`TransportName`*`.` *`ManagerName`*	Connectivity plug-ins - messages from managers.
`connectivity.apama.` *`hostPluginName`*`.`*`ChainName`*	Connectivity plug-ins - messages from the specified host plug-in. The valid host plug-in names (such as `eventMap`) are listed in "Host plug-ins and configuration" in *Connecting Apama Applications to External Components*.
`connectivity.chain.`*`ChainName`*	Connectivity plug-ins - chain-related messages.
`plugins.`*`PluginName`*	EPL plug-ins - in C++, Java or Python. Note that handling of Java logging is slightly different to EPL and Python. By convention, we recommend Java EPL plug-ins should specify `plugins.`*`PluginName`* when creating the `Logger` object, for example, `com.apama.util.Logger.getLogger("plugins.MyPlugin")`. However, this is only a convention, and if some other string or a Java class is specified instead, then that will be used as the correlator's log category. Keep in mind that the logging for the EPL plug-ins is about logging from Java, Python and C++ plug-ins that can be called from EPL. For information on logging from EPL itself, see "Setting EPL log files and log levels in a YAML configuration file" on page 111.

Note:

It is not possible to dynamically change the correlator and plug-in log levels. Only EPL log levels can be dynamically changed (see also "Setting EPL log files and log levels dynamically" on page 182).

Configuring persistence in a YAML configuration file

You can enable and configure correlator persistence in the following ways:

- Using a YAML configuration file as described here.
- On the command line, using the persistence options of the `correlator` executable. See "Starting the correlator" on page 78 for more information on these options.

On the command line, the persistence options are given as `-Poption=value`.

In a configuration file, they are given as follows:

```
correlator:
  persistence:
    option: value
```

All of persistence options for the command line can also be specified in a configuration file. Special notations are required for the following options:

- `-P` without further options or `-Penabled=true` enables persistence. In a configuration file, this is specified as follows:

  ```
  correlator:
    persistence:
      enabled: true
  ```

- `-Pclear` does not have a value; it is implicitly set to `true`. In a configuration file, this is specified as follows:

  ```
  correlator:
    persistence:
       clear: true
  ```

The following is a list of all the options that you can specify in a configuration file:

- `enabled: boolean`
- `snapshotIntervalMillis: interval`
- `adjustSnapshot: boolean`
- `storeLocation: path`
- `storeName: filename`
- `clear: boolean`

The following sample shows the format of a YAML configuration file that is used to specify the persistence options:

```
correlator:
```

```
  persistence:
    enabled: true
    snapshotIntervalMillis: 12000
    storeLocation: ${PARENT_DIR}/store
    storeName: mystore.db
    clear: false
```

For detailed information on correlator persistence, see "Using Correlator Persistence" in *Developing Apama Applications* and especially its subtopic "Enabling correlator persistence" which provides more information on the different persistence options.

Setting up connections between correlators in a YAML configuration file

Rather than separately invoking the `engine_connect` tool, you can also define connections between correlators during correlator startup. These connections are defined as a list of entries in the `engineConnect` section of the YAML configuration file. For example:

```
engineConnect:
  - sourceHost: localhost
    sourcePort: ${FIRST_PORT}
    channels:
      - myChannel
    mode: parallel
    disconnectIfSlow: false
  - sourceHost: localhost
    sourcePort: ${SECOND_PORT}
    channels:
      - myChannel
    mode: parallel
    disconnectIfSlow: true
```

The above elements correspond to command-line options of the `engine_connect` tool and have the same syntax and options. See the descriptions of these command-line options in "Configuring pipelining with engine_connect" on page 60 for detailed information, but keep in mind that only the options shown in the above example are supported.

On startup, the configuration specified in the `engineConnect` section always uses the current correlator as the target correlator. Unlike the `engine_connect` tool, you cannot explicitly specify an arbitrary target correlator with `engineConnect`.

On startup, `engineConnect` is not available to correlators running in persistent mode.

Unlike the `engine_connect` tool, when the `mode` is not specified for an entry in the `engineConnect` YAML, the default mode is `parallel`, meaning that there is one connection for each specified channel.

Specifying JVM options

In a YAML configuration file, you can specify JVM options which the correlator is to pass to the embedded JVM. To do so, you provide a list of JVM options with the `jvmOptions` key. If an option has a leading hyphen, you have to enclose the option in quotes. But you can also specify the option

without quotes by leaving out the leading hyphen; in this case, the correlator will automatically add the hyphen. Thus, you can specify the JVM options as shown in the following example:

```
jvmOptions:
 - "-Dkey1=value1"
 - Dkey2=value2
 - "-Xms100m"
 - Xmx500m
```

You can specify JVM options in multiple configuration files. The options from all the files are then appended together in the order in which they have been specified. If the same JVM option is specified in the command line as well as in a configuration file, the command line takes precedence. For more details, see the description of the `-J` option in "Starting the correlator" on page 78.

Deploying Apama applications with a YAML configuration file

Instead of having another process inject code and send events into a correlator at startup, it is also possible to use a YAML configuration file to list files to be loaded by the correlator at startup. This is useful for Docker containers or other minimal environments where only part of an Apama installation is available or it is not practical to run Java tools to perform the injections. It is also a better fit for Docker use cases as the correlator does not require any other coordination process for startup. For typical installations not using such environments, use of Ant macros is recommended instead, which perform the injections after starting the correlator.

The YAML configuration file for the correlator is specified using the `--config` option when starting the correlator (see also "Starting the correlator" on page 78). The YAML file itself contains the following:

```
correlator:
    initialization:
        list:
            - ${PARENT_DIR}/bin/myPlugin.jar
            - ${PARENT_DIR}/eventdefinitions/evtdef.mon
            - ${APAMA_HOME}/monitors/ConnectivityPlugins.mon
            - ${PARENT_DIR}/monitors/app.mon
            - ${PARENT_DIR}/events/start.evt
        encoding: UTF8
```

It is recommended to use a substitution variable such as `${APAMA_HOME}` or `${PARENT_DIR}` rather than absolute or relative paths. This makes the configuration independent of the correlator's current working directory.

The `list` entries must have one of the following extensions:

- `.mon` for EPL monitor, aggregate and event definition source.
- `.jar` for EPL plug-ins written in Java.
- `.cdp` for correlator deployment packages.
- `.evt` for event files.

Apama queries (`.qry`) are not supported in source form.

Note:

If you use the `engine_deploy` tool, EPL code is automatically generated from query files. For further information, see "Deploying a correlator" on page 126.

Apama queries are deprecated and will be removed in a future release.

The `encoding` entry is optional. If `UTF8` is specified, all of the text input files (`.mon`, `.evt`) are read as UTF-8. If `local` is specified or if the `encoding` entry is not specified at all, the text files are assumed to be in the local encoding unless they start with a UTF-8 byte order mark (BOM) in which case they are treated as UTF-8.

This mechanism separates the build-time (calculating injection order, generating EPL) steps from the deployment-time steps, so no build steps are required in the environment where the correlator is running. This does mean that any changes to the project potentially require rewriting the YAML list and then redeploying, however, it allows separation of these concerns.

Note:

The correlator port is opened before the injections have completed. This allows monitoring tools to connect while the injections occur, but this also means that the correlator may not be entirely ready when a client connects. You can use the `flushAllQueues` request (see "Shutting down and managing components" on page 161) to wait for the injections to complete.

Injecting code into a correlator

To inject EPL files, EPL plug-ins in Java, JMon 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 or using the `apama_env` wrapper (see "Setting up the environment using the Apama Command Prompt" on page 15) ensures that the environment variables are set correctly.

Note:
Apama's in-process API for Java (JMon) is deprecated and will be removed in a future release.

Synopsis

To inject applications into the correlator, run the following command:

```
engine_inject [ options ] [ file1 [ file2 ... ] ]
```

When you run this command with the `-h` option, the usage message for this command is shown.

Description

The `engine_inject` tool reads application definitions from the specified file(s) and injects them into a correlator. If you do not specify a filename, or if you specify a hyphen (-) 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 Event Processing Language (EPL). For more information on EPL, see "Introduction to Apama Event Processing Language" in *Developing Apama Applications*. Alternatively, you can specify the `-j` or `-c` options. The `-j` option specifies that you will inject an application or plug-in 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 EPL plug-in written in Java or JMon application. For more information, see "Writing EPL Plug-ins in Java " and "Overview of JMon Applications" in *Developing Apama Applications*.

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 correlator input files" on page 134.

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 JMon applications, the 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 file `2.mon` contains an error, then `engine_inject` successfully injects `1.mon` and then terminates when it 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.

Note:

If a license file cannot be found, the correlator does not allow the injection of user-generated CDPs. See "Running Apama without a license file" in *Introduction to Apama*.

Options

The `engine_inject` tool takes the following options:

Option	Description
`-h | --help`	Displays usage information.
`-n` *`host`* `| --hostname` *`host`*	Name of the host on which the correlator is running. The default is `localhost`. Non-ASCII characters are not allowed in host names.

Option	Description
`-p port | --port port`	Port on which the correlator is listening. The default is `15903`.
`-v | --verbose`	Requests verbose output during `engine_inject` execution.
`-u | --utf8`	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 this option to override this assumption. The `engine_inject` tool then assumes that all input files are in UTF-8.
`-V | --version`	Displays version information for the `engine_inject` tool.
`-j | --java`	Indicates that each operand is a Java archive file (JAR file) that contains a single JMon application or an EPL plug-in written in Java.
`-c | --cdp`	Indicates that each operand is a correlator deployment package (CDP) file.
`-s | --hashes`	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:

Operand	Description
`[ file1 [ file2 ... ] ]`	The names of zero or more files that contain application data in Apama EPL, JMon, or correlator deployment package (CDP) 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:

Value	Description
`0`	All definitions were injected into the correlator successfully.
`1`	No connection to the correlator was possible or the connection failed.

Value	Description
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.

Creating and managing an Apama project from the command line

The `apama_project` tool can be used to create and manage an Apama project outside of Software AG Designer. It lets you perform the following actions:

- create an Apama project,
- list all supported bundles that can be added to a project,
- list all bundles/instances that have already been added to a project,
- add connectivity, adapter and EPL bundles and their instances to a project,
- remove bundles/instances from a project.

Each Apama project generated by this tool is compatible with Software AG Designer and can seamlessly be imported into Software AG Designer as an Apama project. For more information, see "Working with Projects" in *Using Apama with Software AG Designer*.

The executable for this tool is located in the `bin` directory of the Apama installation. It is recommended that you run this tool in the Apama Command Prompt or using the `apama_env` wrapper (see "Setting up the environment using the Apama Command Prompt" on page 15); otherwise you have to provide the full path the `apama_project` executable each time you run it.

Synopsis

To use this tool, run the following command:

```
apama_project command [options]
```

When you run this command with the `help` (or `--help`, `-help` or `-h`) option, the usage message for this command is shown. For example:

- To display help for all commands:

```
apama_project help
```

- To display help for a specific command:

```
apama_project command help
```

Description

You can only specify one command at a time.

Apart from the command for creating a project, all other commands require you to run the tool from the project directory.

The `apama_project` tool takes the following commands and options:

Command	Option	Description
`create`	*projectname*	Creates a new project directory with the specified name in the current directory. If the project name is to consist of more than one word, you have to enclose it in quotes. The new project directory has the same structure and content as an Apama project that you create with Software AG Designer. If desired, you can import it into Software AG Designer. **Note:** We recommend committing your project into a version control system, so that changes are recorded and you can compare or revert changes across versions if needed.
`list bundles`	*not applicable*	Lists all connectivity, adapter and EPL bundles that are available for adding to the project. Also lists the bundles with instance information that are already present in the project. Some bundles support adding several instances to the same project if desired. For example, you can add separate HTTP Client connectivity bundle instances to talk to different servers. Other bundles support adding only one instance. Such "singleton" bundles, once added to the project, will no longer show up in the list of bundles that can be added. Index numbers are shown for the bundles which can still be added. They can be used as shortcuts instead of typing the full name. Note that the index numbers are subject to change; they differ

Command	Option	Description
		each time a bundle has been added. So if you want to use the number shortcut, list the bundles once more in order to find the correct number.
`add bundle`	*`bundlename`* `[--instance` *`instancename`*`]`	Adds the specified bundle to the project (which was either created using the `apama_project` tool or Software AG Designer). The *`bundlename`* can be a display name that was displayed by the `list bundles` command or the absolute path of a `.bnd` file. If the name consists of more than one word, you have to enclose it in quotes. Instead of specifying a bundle name, you can also specify the corresponding index number that has been output with the `list bundles` command. For bundles supporting multiple bundle instances, you can optionally specify a user-defined instance name using `--instance` *`instancename`*. This is useful for indicating the logical name or purpose of the service or server being connected to. For some bundles, the provided instance name may be used in EPL code. Once added, the instance cannot be renamed, though it is possible to delete and re-add it with a different name. If a user-defined instance is not specified, the default instance name is used. Duplicate instance names are not allowed for the project, even if the instances are of different bundles. You can only add bundles from the standard EPL and connectivity catalogs. Adding bundles from the Capital Markets Foundation, Capital Markets Adapters, or user-defined bundle catalogs is not supported.
	`bundlename bundlename ...`	Adds multiple bundles to the project (which was either created using the `apama_project` tool or Software AG Designer). The *`bundlename`* can be a display name that was displayed by the `list bundles` command or the absolute path of a `.bnd` file. If the name consists of more than one word, you have to enclose it in quotes.

Command	Option	Description
		Note: When adding multiple bundles, it is not permitted to specify index numbers or the `--instance` argument.
		You can only add bundles from the standard EPL and connectivity catalogs. Adding bundles from the Capital Markets Foundation, Capital Markets Adapters, or user-defined bundle catalogs is not supported.
`remove bundle`	*`bundlename`* | *`instancename`*	Removes the specified bundle from the project. Either specify an *`instancename`* to remove a specific instance or specify a *`bundlename`* to remove all instances associated with the specified bundle display name.
		CAUTION: All associated files will be deleted from disk. So check carefully and take a backup of your project if you are in doubt whether you really want to delete it.

Examples

The following examples show the different ways in which the `apama_project` tool can be started.

Note:
Keep in mind that you have to change to the project directory if you want to run the commands for listing, adding or removing bundles.

- Create an Apama project that is named "MyApamaProject" in the current directory:

```
apama_project create MyApamaProject
```

- List all connectivity and EPL bundles in `APAMA_HOME` that can be added to the current Apama project:

```
apama_project list bundles
```

- Add a bundle to the current Apama project, with a user-defined instance name:

```
apama_project add bundle "HTTP Client" --instance "MyHTTP"
```

- Add a bundle to the current Apama project using the index number:

```
apama_project add bundle 34
```

- Remove a bundle from the current Apama project, including all of its instances:

```
apama_project remove bundle "HTTP Client"
```

Deploying a correlator

The `engine_deploy` tool lets you perform the following actions with an Apama project that has been created with Software AG Designer or with the `apama_project` tool:

- generate an initialization file list,
- generate a correlator deployment directory,
- create a Zip file with the contents of the correlator deployment directory,
- generate a correlator deployment package (CDP),
- perform the initialization in a running correlator.

This tool can also be used with a directory of Apama files if you are not using Software AG Designer.

The executable for this tool is located in the `bin` directory of the Apama installation. Running the tool in the Apama Command Prompt or using the `apama_env` wrapper (see "Setting up the environment using the Apama Command Prompt" on page 15) ensures that the environment variables are set correctly.

You can also use Ant to generate a correlator deployment directory or a correlator deployment package. The `apama-macros.xml` file includes the `generate-correlator-deploy-dir` and `project-to-cdp` macros for this purpose. See also "About deploying Apama applications with an Ant script" on page 14.

Note:

If a license file cannot be found, the correlator cannot read user-generated CDPs. See "Running Apama without a license file" in *Introduction to Apama*.

Synopsis

To use this tool, run the following command:

```
engine_deploy action [options] path1 [path2 ...]
```

When you run this command with the `-h` option, the usage message for this command is shown.

Description

The *`action`* that you have to specify when you run this tool can be one of the following:

Action	Description
`--outputList` *`file`* `| stdout`	Specifies where the initialization file list is to be created. This can be written either to a file or standard output.

Action	Description

`-d dir | zipfile`
 	
or
 	
`--outputDeployDir dir | zipfile`
	Specifies the deployment directory into which the project artifacts are to be copied and in which the YAML configuration files and properties files are to be created. You must not specify a subdirectory of the project directory as the deployment directory.
	
	Instead of a directory, you can also specify the name of a Zip file that is to be created (for example, `OutputDir.zip`) and which contains the same contents as the deployment directory. If you specify the name of an existing Zip file, this Zip file will be overwritten.

`--outputCDP file`	Specifies the file name of the correlator deployment package (CDP) that is to be generated. The CDP is created using the correct injection order, and it contains any EPL files, JMon JAR files, event files and nested CDPs. **Note:** You can also create CDPs using Software AG Designer. See "Exporting correlator deployment packages" in *Using Apama with Software AG Designer*. Note that Apama's in-process API for Java (JMon) is deprecated and will be removed in a future release.
`--inject host port`	Specifies that each EPL file is to be injected into the correlator that is running on the specified host and port. **Note:** It is not advisable to inject CDP files containing event files with large numbers of events into a persistent correlator. In this case, it is recommended that you create a CDP which only contains event files. A CDP which only contains event files is not persisted.

Note:
Only one action can be specified at a time.

Deployment directory

When you use this tool to generate a deployment directory, it copies all required files from the project into a deployment directory. It also generates all required YAML configuration files and properties files using the information that is currently defined in the project's launch configuration (see "Launching Projects" in *Using Apama with Software AG Designer* for more information on how to set up a launch configuration).

The deployment directory includes all of the project artifacts, except for log files, along with following generated files:

File name	Contents
`initialization.yaml`	List of all files that are to be loaded by the correlator at startup. See also "Deploying Apama applications with a YAML configuration file" on page 118. This file is always generated.
`initialization.properties`	Substitution variables for locations outside the project directory and `APAMA_HOME` used by `initialization.yaml`. This file is not generated for projects that use only initialization files from `APAMA_HOME` and the project directory.
`connectivity.yaml`	Connectivity configuration. This file is only generated when the input directory is an Apama project and when this project includes connectivity bundles.
`arguments.yaml`	Configuration entries for the port, log file and log level. This file is only generated when a launch configuration is present.
`correlator.properties`	Substitution variables for customizing the settings in `arguments.yaml`, such as the port. Includes `extraArgs` for any extra command line arguments specified in the launch configuration. These cannot be used by the YAML configuration file for the correlator. Therefore, they must be manually passed on the command line by whatever tool is responsible for staring the correlator. This file contains properties `logsDir` and `dataDir` to allow easily changing the location where data (for example, runtime cache directory, pid file) and log files are written without needing to modify the properties for each file path individually. There is also a property `startTimeoutSecs`, which is not currently used by any Apama tools, but can be used to provide a hint to customer-developed deployment or testing tools about how long to wait for the component to start up.
`persistence.yaml`	Persistence configuration of the project. This file is only generated when a launch configuration is present.
`persistence.properties`	Values for the persistence options such as `storeLocation`, `clear` and `enabled`. See also "Configuring persistence in a YAML configuration file" on page 116.

After all output has been generated, you use the `correlator` executable with the `--config` option to start the correlator with all the YAML and properties files that have been generated. For example:

```
correlator --config C:/MyDeployDir
```

See "Starting the correlator" on page 78 for detailed information on the available options.

If you want to override one or more property values that are defined in the generated properties files, you have to send an additional properties file containing these overrides (or you can set the property directly by specifying it on the command line) to the correlator *before* sending the generated properties files. For example, when you specify the following, the property values defined in the file `myOverrides.properties` will take precedence over all of the properties defined in other files.

```
correlator --config myOverrides.properties --config deployDir/
```

Important:

If the project requires any JAR files from an EPL plug-in in Java or from a JMon application, you have to make available the JAR files before you start the `engine_deploy` tool.

Note that Apama's in-process API for Java (JMon) is deprecated and will be removed in a future release.

Options

The `engine_deploy` tool takes the following options:

Option	Description
`--include` *`pattern`*`[,`*`pattern`*`]`	Specifies the files from the project that are to be included in the output/injection. For example: `**/foo/Bar*.evt,**.mon`
`--exclude` *`pattern`*`[,`*`pattern`*`]`	Specifies the files from the project that are to be excluded from the output/injection. For example: `**/foo/Bar*.evt` **Note:** Log files are always excluded.

Operands

`path1` and other optional paths that you can specify when you run the `engine_deploy` tool can point to the following:

- A project directory. This is the directory which contains the `.project` file and, if defined, the `.dependencies` file.

  If exactly one deployment (`.deploy`) file exists in the project directory, it is automatically used. If the directory contains more than one deployment file, an exception is thrown. In this case, you have to select the deployment file to be used by specifying the path to it (instead of specifying the path to a project directory). You also have to specify the path to a deployment file explicitly if your launch configuration contains multiple correlators, followed by an exclamation mark (!) and the correlator name as described below.

Note:
You can only specify one project at a time. If a project references additional projects, then the generated injection order might not be accurate.

If you are not using Software AG Designer or the `apama_project` tool, you can specify any directory containing Apama files (for example, `.mon` files, etc.).

- A deployment (`.deploy`) file. This file is automatically generated by Software AG Designer for each launch configuration that has been defined for a given project. It is located under *`project_dir`*`/config/launch`. See also "Defining custom launch configurations" in *Using Apama with Software AG Designer*.

  If more than one correlator is defined in the deployment file, you have to add an exclamation mark (!) followed by the correlator name (otherwise, the tool will give an error message and fail). For example:

  ```
  MyDeployFile.deploy!myCorrelator
  ```

  If your launch configuration has multiple correlators, it is recommended that you generate a separate correlator deployment directory for each correlator.

  If a deployment file is used which contains environment variables, you have to explicitly specify these variables when you start the correlator with the `correlator` executable. These variables are not captured from the launch configuration. This is important when using EPL plug-ins and connectivity plug-ins that are written in C++.

- Zero or more `.properties` files which contain substitution variables that have been defined in the specified project. The properties files are used when you specify a project directory or deployment file. You can specify these paths in any order.
- A text (`.txt`) file. This is the initialization file list which lists the project artifacts to be included.
- One or more correlator deployment packages (`.cdp` files) to be injected into the correlator.

Examples

The following examples (for Windows) show the different ways in which the `engine_deploy` tool can be started.

- Create an initialization list by pointing to a project directory containing the EPL files:

  ```
  engine_deploy --outputList C:/initialization_list.txt MyProject
  ```

- Create a correlator deployment directory by pointing to a deployment file within the project, and using the substitution variables that have been defined in a properties file that is also available in the project:

  ```
  engine_deploy --outputDeployDir C:/MyDeployDir
     MyProject/config/launch/MyDeployFile.deploy!myCorrelator
     C:/MyProjects/environment.properties
  ```

  The name of the correlator that is to be used is given after the exclamation mark (!).

Files coming from external variables are also copied into `C:/MyDeployDir`. In addition, the file `initialization.properties` is generated which contains information on these variables.

- Create a correlator deployment package by pointing to a deployment file within the project:

```
engine_deploy --outputCDP C:/output.cdp
   C:/MyDeployDir/MyProject/config/launch/MyDeployFile.deploy!myCorrelator
```

The correlator deployment package can then be injected into the correlator using `engine_initalize` or `engine_inject`.

- Perform the initialization into a running correlator by pointing to a deployment file within the project and excluding specific files from the injection:

```
engine_deploy --inject C:/MyDeployDir
   --exclude MyProject/dashboards/**
   MyProject/config/launch/MyDeployFile.deploy!myCorrelator
```

The `--exclude` option specifies that the generated deployment directory does not contain the files from the `MyProject/dashboards` directory, and that these files are also not to be used during injection.

Deleting code from a correlator

The `engine_delete` tool removes EPL code and JMon 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 or using the `apama_env` wrapper (see "Setting up the environment using the Apama Command Prompt" on page 15) ensures that the environment variables are set correctly.

Note:
Apama's in-process API for Java (JMon) is deprecated and will be removed in a future release.

Synopsis

To remove applications from the correlator, run the following command:

```
engine_delete [ options ] [ name1 [ name2 ... ] ]
```

When you run this command with the `-h` option, the usage message for this command is shown.

Description

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

Time taken to delete code

Deleting code from a correlator can require scanning the state of the correlator to ensure that the types being deleted are no longer in use. Thus, the deletion will run at least as slowly as it takes the slowest context in the correlator to respond to external events, and will depend on how many objects there are live in the correlator.

If a type is found to be in use and you are not using the `-F` or `-a` option, then the deletion will fail with an error message, reporting what is still using the type that was requested to be deleted. If events of the type being deleted are sent to the correlator, they will fail to be parsed and the correlator will report errors.

Options

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

Option	Description
`-h | --help`	Displays usage information. Optional.
`-n` *`host`* `| --hostname` *`host`*	Name of the host on which the correlator is running. The default is `localhost`. Optional. Non-ASCII characters are not allowed in host names.
`-p` *`port`* `| --port` *`port`*	Port on which the correlator is listening. Optional. The default is `15903`.

Option	Description
`-f filename | --file 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 | --force`	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 | --kill`	Kills all instances of the named monitor regardless of whether an instance is in use. For example, you can specify this option to remove a monitor that is stuck in an infinite loop. Any `ondie()` and `onunload()` actions defined in killed monitors are not executed.
`-a | --all`	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 161.
`-y | --yes`	Removes the "are you sure?" prompt when using the `-a` option.
`-v | --verbose`	Requests verbose output.
`-u | --utf8`	Indicates that input files are in UTF-8 encoding. This specifies that the `engine_delete` tool should not convert the input to any other encoding.
`-V | --version`	Displays version information for the `engine_delete` tool.

Operands

The `engine_delete` tool takes the following operands:

Operand	Description
`[ name1 [ name2 ... ] ]`	The names of zero or more EPL or JMon applications, monitors and/or event types to delete from the 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 `engine_delete` tool returns the following exit values:

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

Packaging correlator input files

The `engine_package` tool assembles EPL files, JAR files and event files into a correlator deployment package (CDP). You can inject a CDP file into the correlator just as you inject an EPL file. 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. See also "Deploying a correlator" on page 126 for alternative ways to deploy the correlator or for creating CDPs from Software AG Designer projects.

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 or using the `apama_env` wrapper (see "Setting up the environment using the Apama Command Prompt" on page 15) ensures that the environment variables are set correctly.

Synopsis

To package files into a CDP file, run the following command:

```
engine_package [ options ] [ file1 [ file2 ... ] ]
```

When you run this command with the `-h` option, the usage message for this command is shown.

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.

You can also specify another CDP file to include in this package. The files from the original CDP are injected in the specified place in the order within this package.

Options

The `engine_package` tool takes the following options:

Option	Description
`-h | --help`	Displays usage information.
`-V | --version`	Displays version information for the `engine_package` tool.
`-o` *`filename`* `| --output` *`filename`*	Name of the CDP file to create. Required.
`-m` *`filename`* `| --manifest` *`filename`*	Name of the manifest file that lists the files you want to package.
`-u | --utf8`	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:

Operand	Description
`[` *`file1`* `[` *`file2`* `... ] ]`	The names of the EPL, JAR, event or other CDP files to be included in the package. 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:

Value	Description
`0`	Returned on success.
`1`	Returned 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\java-plugin.jar
   ```

2. To create the CDP file, call the `engine_package` tool 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` tool 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
   ```

Sending events to correlators

The `engine_send` tool sends Apama-format events into a 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 or using the `apama_env` wrapper (see "Setting up the environment using the Apama Command Prompt" on page 15) 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.

Note:

You can also send events using Software AG Designer. For more information, see "Sending an event from the Engine Information view" in *Using Apama with Software AG Designer*.

Synopsis

To send Apama-format events to a correlator or IAF adapter, run the following command:

```
engine_send [ options ] [ file1 [ file2 ... ] ]
```

When you run this command with the `-h` option, the usage message for this command is shown.

Description

The `engine_send` tool sends Apama-format events to a 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 a hyphen (-) or not specify a filename so that 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 206.

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 options:

Option	Description
`-h | --help`	Displays usage information. Optional.
`-n host | --hostname host`	Name of the host on which the correlator to which you want to send events is running. Optional. The default is `localhost`. Non-ASCII characters are not allowed in host names.
`-p port | --port port`	Port on which the correlator is listening. Optional. The default is `15903`.
`-c channel | --channel channel`	For events for which a channel is not specified, this option designates the delivery channel. If a channel is not specified for an event and you do not specify this option, the event is delivered to the default channel, which is the empty string. All public contexts receive events sent to the default channel. All queries receive events sent to the default channel.

Option	Description
	To send events to only running Apama queries, specify the `com.apama.queries` channel. See "Defining Queries" in *Developing Apama Applications*. **Note:** Apama queries are deprecated and will be removed in a future release.
`-l` *count* | `--loop` *count*	Number of times to cycle through and send the input events. Optional. Replace *count* 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`.
`-v` | `--verbose`	Requests verbose output during execution. Optional.
`-u` | `--utf8`	Indicates that input files are in UTF-8 encoding. This specifies that the `engine_send` tool should not convert the input to any other encoding.
`-V` | `--version`	Displays version information for the `engine_send` tool. Optional.

Operands

The `engine_send` tool takes the following operands:

Operand	Description
`[` *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 206. If you do not specify any filenames, the `engine_send` tool takes input from `stdin`.

Exit status

The `engine_send` tool returns the following exit values:

Value	Description
0	The events were sent successfully.
1	No connection to the 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.

Receiving events from correlators

The `engine_receive` tool lets you connect to a running correlator and receive events from it. 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 or using the `apama_env` wrapper (see "Setting up the environment using the Apama Command Prompt" on page 15) ensures that the environment variables are set correctly.

Synopsis

To receive Apama-format events from a correlator, run the following command:

```
engine_receive [ options ]
```

When you run this command with the `-h` option, the usage message for this command is shown.

Description

The `engine_receive` tool receives events from a 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 206.

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 more information, see "Subscribing to channels" in *Developing Apama Applications*.

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`.

Options

The `engine_receive` tool takes the following options:

Option	Description
`-h | --help`	Displays usage information. Optional.
`-n host | --hostname host`	Name of the host on which the correlator is running. Optional. The default is `localhost`. Non-ASCII characters are not allowed in host names.
`-p port | --port port`	Port on which the correlator is listening. Optional. The default is `15903`.
`-c channel | --channel 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 | --filename file`	Dumps all received events in the specified file. Optional. The default is to write the events to `stdout`.
`-s | --suppressBatch | --suppressbatch`	Omits `BATCH` timestamps from the output events. Optional. The default is to preserve `BATCH` timestamps in events.
`-z | --zeroAtFirstBatch | --zeroatfirstbatch`	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.
`-C | --logChannels`	Specifies that you want `engine_receive` output to include the channel that an event arrives on. If you then use the `engine_receive` output as input to `engine_send`, events are

Option	Description
	delivered back to the same-named channels. See "Event association with a channel" on page 211.
`-r | --reconnect`	Automatically (re)connect to the server when available.
`-x | --qdisconnect`	Disconnect from the correlator if the `engine_receive` tool cannot keep up with the events from the correlator.
`-v | --verbose`	Requests verbose output during `engine_receive` execution. Optional.
`-u | --utf8`	Indicates that received event files are in UTF-8 encoding. This specifies that the `engine_receive` tool should not convert the input to any other encoding.
`-V | --version`	Displays version information for the `engine_receive` tool. Optional.

Exit status

The `engine_receive` tool returns the following exit values:

Value	Description
`0`	All events were received successfully.
`1`	No connection to the 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.

Watching correlator runtime status

The `engine_watch` tool lets you monitor the runtime operational status of a running correlator. The executable for this tool is located in the `bin` directory of the Apama installation. Running the tool in the Apama Command Prompt or using the `apama_env` wrapper (see "Setting up the environment using the Apama Command Prompt" on page 15) ensures that the environment variables are set correctly.

Synopsis

To monitor the operation of a correlator, run the following command:

```
engine_watch [ options ]
```

When you run this command with the `-h` option, the usage message for this command is shown.

Description

The `engine_watch` tool periodically polls a correlator for status information, writing the standard status messages to `stdout` (see "List of correlator status statistics" on page 143 for more information on the standard status messages). When you also specify the `-a` option, any user-defined status values are appended to the standard status messages. For additional progress information, use the `-v` option.

Options

The `engine_watch` tool takes the following options:

Option	Description
`-h | --help`	Displays usage information. Optional.
`-n host | --hostname host`	Name of the host on which the correlator is running. The default is `localhost`. Non-ASCII characters are not allowed in host names.
`-p port | --port port`	Port on which the correlator is listening. Optional. The default is `15903`.
`-i ms | --interval ms`	Specifies the poll interval in milliseconds. Optional. The default is `1000`.
`-f filename | --filename filename`	Writes status output to the named file. Optional. The default is to send status information to `stdout`.
`-r | --raw`	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.
`-a | --all`	Outputs all user-defined status values after the standard status messages. Optional. The default is to output only the standard status messages.
`-t | --title`	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.

Option	Description
`-o | --once`	Outputs one set of status information and then quits. Optional. The default is to indefinitely return status information at the specified poll interval.
`-v | --verbose`	Displays process names and versions in addition to status information. Optional. The default is to display only status information.
`-V | --version`	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:

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

List of correlator status statistics

This topic gives a detailed list of the status values that can be monitored for a correlator. The descriptions below show where the status values are used. The status is available through the following mechanisms:

- REST API: The name of the key in the REST API. See also "Managing and Monitoring over REST" on page 71 and the descriptions of `/correlator/status` and `/info/stats` in the *API Reference for Component Management REST APIs*.
- Java API: The name of the method in the `EngineStatus` Java API (and also the equivalent, where available, in the C++, .NET and EPL APIs). See the `com.apama.engine.EngineStatus` interface in the *API Reference for Java (Javadoc)*.

  Note:

  In the C++ API, the status statistics names are defined as constants inside the `client_status_names.hpp` file. See also the *API Reference for C++ (Doxygen)*.

- Log field: The name of the status log field in the `Status:` log lines in the main correlator log file. See also "Descriptions of correlator status log fields" on page 93.
- Prometheus metric name: The name used to expose internal correlator statistics to the Prometheus monitoring system. See also "Monitoring with Prometheus" on page 75.

- Display name: The standard status message that the `engine_watch` tool writes to `stdout` (see "Watching correlator runtime status" on page 141). The same status message is also shown in the Engine Status view of Software AG Designer (see "Using the Engine Status view" in *Using Apama with Software AG Designer*).

The descriptions below also indicate the typical trend. This can be one of the following:

- Steady: After any start-up phase, this number would typically be steady. It may increase as bursts of events come in, or if there is a change in the size of the application (for example, the number of items the application is tracking). Typically, if these numbers are continually trending upwards when there is no more being asked of the application, that indicates an application leak of monitor instances, listeners or objects. This will eventually lead to an out of memory condition.
- Increasing: This may be increasing in normal usage. Depending on deployment, some statistics may not be increasing, though if they normally are and have stopped increasing, this may indicate that something is preventing events being delivered or processed correctly.
- Low: This number is typically 0 or near 0. If this number increases, this typically indicates that the correlator is not keeping up with processing events. For queues, it is normal that during bursts of activity, these may be non-zero for some time. Steadily increasing queue sizes can be a sign of back-pressure due to a slow receiver, or the system is not keeping up and may eventually block senders due to not processing the events at the rate they arrive.
- Varies: Will typically vary. 0 may indicate a problem with events being delivered.
- None: Typically, all contexts and receivers should be keeping up, so none are reported as slow (in which case, the empty string will be returned from the API).

The term "receiver" which is used in the descriptions below refers to any of the following:

- EPL, Java or C++ plug-ins using the `Correlator.subscribe` method.
- Connectivity plug-ins for "towards" transport events.
- JMS connections sending events out of the correlator.
- Client library connections, including other correlators that have been connected with the `engine_connect`, `iaf` or `engine_receive` tools.

Time since the correlator was started

The time in milliseconds since the correlator was started.

Typical trend: increasing.

- REST API: `uptime`
- Java API: `getUptime`
- Log field: *not applicable*
- Prometheus metric name: `sag_apama_correlator_uptime_seconds`

- Display name: `Uptime (ms)`

Number of contexts

The number of contexts in the correlator, including the main context.

Typical trend: steady.

- REST API: `numContexts`
- Java API: `getNumContexts`
- Log field: `nctx=`*n*
- Prometheus metric name: `sag_apama_correlator_contexts_total`
- Display name: `Number of contexts`

Number of monitors

The number of EPL monitor definitions injected into the correlator. This number changes on injections, deletions or if the last instance of a monitor terminates.

Typical trend: steady.

- REST API: `numMonitors`
- Java API: `getNumMonitors`
- Log field: *not applicable*
- Prometheus metric name: `sag_apama_correlator_monitors_total`
- Display name: `Number of monitors`

Number of monitor instances

The number of monitor instances, also known as sub-monitors.

Typical trend: steady.

- REST API: `numProcesses`
- Java API: `getNumProcesses`
- Log field: `sm=`*n*
- Prometheus metric name: `sag_apama_correlator_monitor_instances_total`
- Display name: `Number of sub-monitors`

Number of Java applications and Java EPL plug-ins

The number of Java applications and Java EPL plug-ins loaded in the correlator. This number changes on injections and deletions.

Typical trend: steady.

- REST API: `numJavaApplications`
- Java API: `getNumJavaApplications`
- Log field: *not applicable*
- Prometheus metric name: `sag_apama_correlator_java_applications_total`
- Display name: `Number of Java applications`

Number of listeners

The number of listeners in all contexts. This includes on statements and active stream source templates.

Typical trend: steady.

- REST API: `numListeners`
- Java API: `getNumListeners`
- Log field: `ls=`*n*
- Prometheus metric name: `sag_apama_correlator_listeners_total`
- Display name: `Number of listeners`

Number of sub-listeners

The number of sub-event-listeners that are active across all contexts. Stream source templates will have one sub-event-listener. An `on` statement can have multiple sub-event-listeners. See also "Evaluating event listeners for all A-events followed by B-events" in *Developing Apama Applications*.

Typical trend: steady.

- REST API: `numSubListeners`
- Java API: `getNumSubListeners`
- Log field: *not applicable*
- Prometheus metric name: `sag_apama_correlator_sub_listeners_total`
- Display name: `Number of sub-listeners`

Number of event types

The number of event types defined within the correlator. This number changes on injections and deletions.

Typical trend: steady.

- REST API: `numEventTypes`
- Java API: `getNumEventTypes`
- Log field: *not applicable*
- Prometheus metric name: `sag_apama_correlator_event_types_total`
- Display name: `Number of event types`

Number of executors on input queues

The number of executors on the input queues of all contexts. As well as events, this can include clock ticks, spawns, injections and other operations. A context in an infinite loop will grow by 10 per second due to clock ticks. Every context has an input queue, which by default is a maximum of 20,000 entries.

Typical trend: low.

- REST API: `numQueuedInput`
- Java API: `getNumQueuedInput`
- Log field: `iq=`*n*
- Prometheus metric name: `sag_apama_correlator_queued_input_total`
- Display name: `Events on input queue`

Number of received events

The number of events that the correlator has received from external sources since the correlator started. This includes connectivity plug-ins, correlator-integrated JMS, engine_send, other correlators connected with engine_connect, dashboard servers, the IAF, and events that are not parsed correctly. This number excludes events sent within the correlator from EPL monitors or EPL plug-ins.

Typical trend: increasing.

- REST API: `numReceived`
- Java API: `getNumReceived`
- Log field: `rx=`*n*
- Prometheus metric name: `sag_apama_correlator_input_total`
- Display name: `Events received`

Number of processed events

The number of events processed by the correlator in all contexts. This includes external events and events routed to contexts by monitors. An event 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.

Typical trend: increasing.

- REST API: `numProcessed`
- Java API: `getNumProcessed`
- Log field: *`not applicable`*
- Prometheus metric name: `sag_apama_correlator_processed_total`
- Display name: `Events processed`

Sum of events on route queues

The sum of routed events on the route queues of all contexts.

Typical trend: low.

- REST API: `numQueuedFastTrack`
- Java API: `getNumQueuedFastTrack`
- Log field: `rq=`*`n`*
- Prometheus metric name: `sag_apama_correlator_queued_route_total`
- Display name: `Events on internal queue`

Number of routed events

The number of events that have been routed across all contexts since the correlator was started.

Typical trend: increasing.

- REST API: `numFastTracked`
- Java API: `getNumFastTracked`
- Log field: `rt=`*`n`*
- Prometheus metric name: `sag_apama_correlator_route_total`
- Display name: `Events routed internally`

Number of external consumers/receivers

The number of external consumers/receivers connected to receive emitted events. This includes connectivity plug-ins, correlator-integrated JMS, engine_receive, or correlators connected using engine_connect.

Typical trend: steady.

- REST API: `numConsumers`
- Java API: `getNumConsumers`
- Log field: `nc=`*`n`*
- Prometheus metric name: `sag_apama_correlator_consumers_total`
- Display name: `Number of consumers`

Number of events on output queues

The number of events waiting on output queues to be dispatched to any connected external consumers/receivers.

Typical trend: low.

- REST API: `numOutEventsQueued`
- Java API: `getNumOutEventsQueued`
- Log field: `oq=`*`n`*
- Prometheus metric name: `sag_apama_correlator_queued_output_total`
- Display name: `Events on output queue`

Number of events created for sending to external channels

The number of events that have been sent (see "The send ... to statement" in *Developing Apama Applications*) or emitted (see "The emit statement" in *Developing Apama Applications*) to channels which have at least one external consumer/receiver subscribed (see also "Number of external consumers/receivers" on page 149). This excludes events sent to channels with no external consumers/receivers. This counts each event once, even if delivered to multiple external consumers/receivers.

Typical trend: increasing.

- REST API: `numEmits`
- Java API: `getNumOutEventsCreated`
- Log field: *`not applicable`*
- Prometheus metric name: `sag_apama_correlator_created_output_total`

- Display name: `Output events created`

Number of events delivered to external consumers/receivers

The number of events that have been delivered to external consumers/receivers. This counts for each external consumer/receiver an event is sent to. It counts the number of deliveries of events.

Note:

This status indicator counts every event that was *delivered*, whereas the previous status indicator counts every event that was *sent*. For example, sending one event to a channel with two external consumers/receivers would be counted as one event sent (`numEmits`), but two events delivered (`numOutEventsSent`).

Typical trend: increasing.

- REST API: `numOutEventsSent`
- Java API: `getNumOutEventsSent`
- Log field: `tx=`*n*
- Prometheus metric name: `sag_apama_correlator_output_total`
- Display name: `Output events sent`

Number of events on input queues of all public contexts

The number of events on the input queues of all public contexts. See also "About context properties" in *Developing Apama Applications* for information on the receiveInput flag.

Typical trend: low.

- REST API: `numInputQueuedInput`
- Java API: `getNumInputQueuedInput`
- Log field: `icq=`*n*
- Prometheus metric name: `sag_apama_correlator_queued_input_public_total`
- Display name: `Events on input context queues`

Name of slowest context

The name of the slowest context. This may or may not be a public context.

Typical trend: none.

- REST API: `mostBackedUpInputContext`
- Java API: `getMostBackedUpInput`
- Log field: `lcn=`*name*

- Prometheus metric name: The name of the slowest context is given as a Prometheus label on the Prometheus metric `sag_apama_correlator_slowest_input_queue_size_total`
- Display name: `Slowest context name`

Number of events on queue for slowest context

The number of events on the slowest context's queue, as identified by the name of the slowest context.

Typical trend: low.

- REST API: `mostBackedUpICQueueSize`
- Java API: `getMostBackedUpQueueSize`
- Log field: `lcq=`*`n`*
- Prometheus metric name: `sag_apama_correlator_slowest_input_queue_size_total`
- Display name: `Slowest context queue size`

Time difference in seconds for slowest context

For the context identified by the slowest context name, this is the time difference in seconds between its current logical time and the most recent time tick added to its input queue.

Typical trend: low.

- REST API: `mostBackedUpICLatency`
- Java API: `getMostBackedUpICLatency`
- Log field: `lct=`*`seconds`*
- Prometheus metric name: `sag_apama_correlator_slowest_input_queue_latency_seconds`
- Display name: *not applicable*

Name of slowest consumer/receiver of events

The name of the consumer/receiver with the largest number of incoming events waiting to be processed. This is the slowest non-context consumer/receiver of events, which can be an external receiver or an EPL plug-in.

Typical trend: none.

- REST API: `slowestReceiver`
- Java API: `getSlowestReceiver`
- Log field: `srn=`*`name`*

- Prometheus metric name: The name of the slowest consumer/receiver of events is given as a Prometheus label on the Prometheus metric `sag_apama_correlator_slowest_output_queue_size_total`
- Display name: `Slowest receiver name`

Number of events on queue for slowest consumer/receiver

The number of events on the slowest consumer's/receiver's queue, as identified by the name of the slowest consumer/receiver.

Typical trend: low.

- REST API: `slowestReceiverQueueSize`
- Java API: `getSlowestReceiverQueueSize`
- Log field: `srq=`*n*
- Prometheus metric name: `sag_apama_correlator_slowest_output_queue_size_total`
- Display name: `Slowest receiver queue size`

Number of events per second

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.

Typical trend: varies.

- REST API: *not applicable*
- Java API: *not applicable*
- Log field: *not applicable*
- Prometheus metric name: *not applicable*
- Display name: `Event rate over last interval`

Number of enqueued events

The number of events queued from the enqueue statement (not the enqueue...to statement). The enqueue statement is deprecated.

Typical trend: low.

- REST API: `enqueueQueueSize`
- Java API: *not applicable*
- Log field: *not applicable*
- Prometheus metric name: *not applicable*

- Display name: *not applicable*

Virtual memory

Virtual memory. For the REST API, the value is in megabytes. For the log field, the value is in kilobytes. For Prometheus, the value is in bytes.

Typical trend: steady.

- REST API: `virtualMemoryMB`
- Java API: *not applicable*
- Log field: `vm=kB`
- Prometheus metric name: `sag_apama_correlator_virtual_memory_bytes`
- Display name: *not applicable*

Physical memory

Physical memory. For the REST API, the value is in megabytes. For the log field, the value is in kilobytes. For Prometheus, the value is in bytes.

Typical trend: steady.

- REST API: `physicalMemoryMB`
- Java API: *not applicable*
- Log field: `pm=kB`
- Prometheus metric name: `sag_apama_correlator_physical_memory_bytes`
- Display name: *not applicable*

Peak physical memory usage

The highest amount of physical memory used by the correlator at any measurement point since startup, given in units of megabytes. This is the highest measured amount of memory, measured when a status line is logged or status is requested from the correlator.

Typical trend: steady.

- REST API: `peakPhysicalMemoryMB`
- Java API: *not applicable*
- Log field: *not applicable*
- Prometheus metric name: *not applicable*
- Display name: *not applicable*

Number of contexts on run queue

The number of contexts on the run queue. These are the contexts that have work to do but are not currently running.

Typical trend: low.

- REST API: *not applicable*
- Java API: *not applicable*
- Log field: `runq=`*n*
- Prometheus metric name: *not applicable*
- Display name: *not applicable*

Number of pages read from swap space

The number of pages per second that are being read from swap space. If this is greater than zero, it may indicate that the machine is under-provisioned, which can lead to reduced performance, connection timeouts and other problems. Consider adding more memory, reducing the number of other processes running on the machine, or partitioning your Apama application across multiple machines.

Typical trend: low.

- REST API: `swapPagesRead`
- Java API: *not applicable*
- Log field: `si=`*n*
- Prometheus metric name: `sag_apama_correlator_swap_pages_read_hertz`
- Display name: *not applicable*

Number of pages written to swap space

The number of pages per second that are being written to swap space. If this is greater than zero, it may indicate that the machine is under-provisioned, which can lead to reduced performance, connection timeouts and other problems. Consider adding more memory, reducing the number of other processes running on the machine, or partitioning your Apama application across multiple machines.

Typical trend: low.

- REST API: `swapPagesWrite`
- Java API: *not applicable*
- Log field: `so=`*n*
- Prometheus metric name: `sag_apama_correlator_swap_pages_write_hertz`

- Display name: *not applicable*

Number of snapshots

The number of persistence snapshots taken since the correlator started. These statistics will only exist if the correlator is running in persistent mode.

Typical trend: increasing.

- REST API: `persistenceNumSnapshots`
- Java API: *not applicable*
- Log field: `numSnapshots=`*`n`*
- Prometheus metric name: `sag_apama_correlator_persistence_snapshots_total`
- Display name: *not applicable*

Timestamp of last snapshot

The UNIX timestamp of the last completed snapshot. These statistics will only exist if the correlator is running in persistent mode.

Typical trend: increasing.

- REST API: `persistenceLastSnapshotTime`
- Java API: *not applicable*
- Log field: `lastSnapshotTime=`*`timestamp`*
- Prometheus metric name: `sag_apama_correlator_persistence_last_snapshot_timestamp`
- Display name: *not applicable*

Time waiting for a snapshot (EWMA)

An exponentially weighted moving average (EWMA) of the time in milliseconds taken to wait for a snapshot. These statistics will only exist if the correlator is running in persistent mode.

Typical trend: varies.

- REST API: `persistenceSnapshotWaitTimeEwmaMillis`
- Java API: *not applicable*
- Log field: `snapshotWaitTimeEwmaMillis=`*`milliseconds`*
- Prometheus metric name: `sag_apama_correlator_persistence_snapshot_wait_ewma_seconds`
- Display name: *not applicable*

Time to commit to database (EWMA)

An exponentially weighted moving average (EWMA) of the time in milliseconds taken to commit to the database. These statistics will only exist if the correlator is running in persistent mode.

Typical trend: varies.

- REST API: `persistenceCommitTimeEwmaMillis`
- Java API: *not applicable*
- Log field: `commitTimeEwmaMillis=`*`milliseconds`*
- Prometheus metric name: `sag_apama_correlator_persistence_commit_time_ewma_seconds`
- Display name: *not applicable*

Number of changed rows (EWMA)

An exponentially weighted moving average (EWMA) of the number of rows changed per snapshot. These statistics will only exist if the correlator is running in persistent mode.

Typical trend: varies.

- REST API: `persistenceLastSnapshotRowsChangedEwma`
- Java API: *not applicable*
- Log field: `lastSnapshotRowsChangedEwma=`*`n`*
- Prometheus metric name: `sag_apama_correlator_persistence_snapshot_rows_changed_ewma_total`
- Display name: *not applicable*

Total heap memory used by the JVM

The total heap memory used by the Java virtual machine (JVM) which is embedded in the correlator. For the REST API, the value is in megabytes. For Prometheus, the value is in bytes. These statistics will only exist if the embedded JVM has been enabled. If the JVM is disabled, the REST API will return 0 (zero) as the value, and Prometheus will not have this metric.

Typical trend: steady.

- REST API: `jvmMemoryHeapUsedMB`
- Java API: *not applicable*
- Log field: *not applicable*
- Prometheus metric name: `sag_apama_correlator_jvm_heap_used_bytes`
- Display name: *not applicable*

Total free heap memory in the JVM

The total heap memory that is free in the Java virtual machine (JVM) which is embedded in the correlator. For the REST API, the value is in megabytes. For Prometheus, the value is in bytes. These statistics will only exist if the embedded JVM has been enabled. If the JVM is disabled, the REST API will return 0 (zero) as the value, and Prometheus will not have this metric.

Typical trend: steady.

- REST API: `jvmMemoryHeapFreeMB`
- Java API: *not applicable*
- Log field: *not applicable*
- Prometheus metric name: `sag_apama_correlator_jvm_heap_free_bytes`
- Display name: *not applicable*

Total non-heap memory used by the JVM

The total non-heap memory used by the Java virtual machine (JVM) which is embedded in the correlator. For the REST API, the value is in megabytes. For Prometheus, the value is in bytes. These statistics will only exist if the embedded JVM has been enabled. If the JVM is disabled, the REST API will return 0 (zero) as the value, and Prometheus will not have this metric.

Typical trend: steady.

- REST API: `jvmMemoryNonHeapUsedMB`
- Java API: *not applicable*
- Log field: *not applicable*
- Prometheus metric name: `sag_apama_correlator_jvm_non_heap_used_bytes`
- Display name: *not applicable*

Total memory used by all buffer pools in the JVM

The sum of memory used by all buffer pools in the Java virtual machine (JVM) which is embedded in the correlator. For the REST API, the value is in megabytes. For Prometheus, the value is in bytes. These statistics will only exist if the embedded JVM has been enabled. If the JVM is disabled, the REST API will return 0 (zero) as the value, and Prometheus will not have this metric.

Typical trend: steady.

- REST API: `jvmMemoryBufferPoolUsedMB`
- Java API: *not applicable*
- Log field: *not applicable*
- Prometheus metric name: `sag_apama_correlator_jvm_buffer_pool_used_bytes`

- Display name: *not applicable*

Total memory used by the JVM

The sum of all memory used by the Java virtual machine (JVM) which is embedded in the correlator (that is, the used heap memory, the used non-heap memory, and the used buffer pool memory). For the REST API and the log field, the value is in megabytes. For Prometheus, the value is in bytes. These statistics will only exist if the embedded JVM has been enabled. If the JVM is disabled, the REST API and the log field will return 0 (zero) as the value, and Prometheus will not have this metric.

Typical trend: steady.

- REST API: `jvmMemoryAllUsedMB`
- Java API: *not applicable*
- Log field: `jvm=`*`MB`*
- Prometheus metric name: `sag_apama_correlator_jvm_memory_all_bytes`
- Display name: *not applicable*

Number of threads in use by the JVM

The total number of active threads in the Java virtual machine (JVM). These statistics will only exist if the embedded JVM has been enabled. If the JVM is disabled, the REST API will return 0 (zero) as the value, and Prometheus will not have this metric.

Typical trend: steady.

- REST API: `jvmNumThreads`
- Java API: *not applicable*
- Log field: *not applicable*
- Prometheus metric name: `sag_apama_correlator_jvm_num_threads`
- Display name: *not applicable*

Inspecting correlator state

The `engine_inspect` tool lets you inspect the state of a running 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 or using the `apama_env` wrapper (see "Setting up the environment using the Apama Command Prompt" on page 15) ensures that the environment variables are set correctly.

Synopsis

To inspect applications on a running correlator, run the following command:

```
engine_inspect [ options ]
```

When you run this command with the `-h` option, the usage message for this command is shown.

Description

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

Note:
Apama's in-process API for Java (JMon) is deprecated and will be removed in a future release.

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

Options

The `engine_inspect` tool takes the following options:

Option	Description
`-m | --monitors`	Displays the names of all EPL monitors in the correlator and the number of monitor instances each monitor has spawned.
`-j | --java`	Displays the names of all JMon applications in the correlator and the number of event listeners each JMon application has created. **Note:** Apama's in-process API for Java (JMon) is deprecated and will be removed in a future release.
`-e | --events`	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*.
`-t | --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 a stream network.
`-x | --contexts`	Displays the names of any user-defined contexts, how many monitor instances are running in each context, what channels each context is subscribed to, and how many entries are on each context's input queue.

Option	Description
`-a | --aggregates`	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.
`-P | --pluginReceivers`	Displays the names of any plug-in receivers, the channels the plug-in is subscribed to, and the number of items on the plug-in's input queue. A plug-in receiver is an EPL plug-in that is subscribed to one or more channels.
`-R | --receivers`	Displays the names of any external receivers, each receiver's address, the channels each receiver is subscribed to, and the number of entries on each receiver's output queue.

`-r | --raw`	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 JMon 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 | --help`	Displays usage information.
`-n` *`host`* `| --hostname` *`host`*	Name of the host on which the correlator is running. The default is `localhost`. Non-ASCII characters are not allowed in host names.
`-p` *`port`* `| --port` *`port`*	Port on which the correlator is listening. The default is `15903`.
`-v | --verbose`	Displays process names and versions in addition to application information. Optional. The default is to display only application information.
`-V | --version`	Displays version information for the `engine_inspect` tool.

Exit status

The `engine_inspect` tool returns the following exit values:

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

Shutting down and managing components

All Apama components (correlator, IAF, dashboard data server, and dashboard display server) implement an interface with which they can be asked to shut themselves down, provide their process identifier, and respond to communication checks.

For historical reasons, there are several tools that all do the same thing. You can use any of these tools to manage any component:

- `engine_management`
- `component_management`
- `iaf_management`

When managing a correlator, the recommendation is to use the `engine_management` tool, which provides some additional correlator-specific options that are not available in the other tools. The only other differences in behavior among these tools are:

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

The executable for the `engine_management` tool is located in the `bin` directory of the Apama installation. Running the tool in the Apama Command Prompt or using the `apama_env` wrapper (see "Setting up the environment using the Apama Command Prompt" on page 15) ensures that the environment variables are set correctly.

Note:

Some of the functions of the `engine_management` tool can be performed from within EPL. For more information, see "Using the Management interface" in *Developing Apama Applications*.

Synopsis

To use the correlator's management tool, run the following command:

```
engine_management [ options ]
```

When you run this command with the `-h` option, the usage message for this command is shown.

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:

- Correlator
- Adapter

- Dashboard data server and dashboard display server (using the management port, and not the data port)

  If you want to use the dedicated `dashboard_management` tool, see "Managing and stopping the data server and display server" in *Building and Using Apama Dashboards*.

The `engine_management` tool sends output to `stdout`.

Options

The `engine_management` tool takes the following options. These options are also available for the `component_management` and `iaf_management` tools. Keep in mind that all of these tools use different ports (see above). To obtain all information for a particular component, specify the `-a` option. All options are optional.

Option	Description
`-V | --version`	Displays version information for the `engine_management` tool.
`-h | --help`	Display usage information.
`-v | --verbose`	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 are having trouble obtaining the information you want, specify the `-v` option to help determine where the problem is.
`-n` *`host`* `| --hostname` *`host`*	Name of the host on which the component is running. The default is `localhost`. Non-ASCII characters are not allowed in host names.
`-p` *`port`* `| --port` *`port`*	Port on which the component you want to connect to is listening. The default is `15903`.
`-w | --wait`	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`* `| --waitFor` *`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

Option	Description

	`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.

`-N | --getname`	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 | --gettype`	Displays the type of the component that the `engine_management` tool connects to. The returned value is one of the following: `correlator` or `iaf`. 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.
`-M | --getuptime`	Gets the uptime of the component in milliseconds. This can be useful if you wish to track when and for how long a particular component has been running for.
`-Vm | --getvmemory`	Gets the virtual memory usage of the component in megabytes. This can be useful if you wish to measure

Option	Description
	the virtual memory usage of a component, for example, to identify possible memory leaks. For the Java-based dashboard data server and display server, the virtual memory value returned is the total of the heap and non-heap "used" memory, as given by the `java.lang.management.MemoryMXBean` class.
`-Pm | --getpmemory`	Gets the physical memory usage of the component in megabytes. This can be useful if you wish to measure the physical memory usage of a component, for example, to identify possible memory leaks. For the Java-based dashboard data server and display server, the physical memory value returned is the total of the heap and non-heap "committed" memory, as given by the `java.lang.management.MemoryMXBean` class.
`-Y | --getphysical`	Displays the physical ID of the component. This can be useful if you are looking at log information that identifies components by their physical IDs.
`-L | --getlogical`	Displays the logical ID of the component. This can be useful if you are looking at log information that identifies components by their logical IDs.
`-O | --getloglevel`	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 | --getversion`	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 | --getproduct`	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 | --getbuild`	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 | --getplatform`	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.

Option	Description
`-P | --getpid`	Displays the process identifier of the correlator you are connecting to. This can be useful if you are looking at log information that identifies components by their process identifier.
`-H | --gethostname`	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 | --getusername`	Displays the user name of the component. On a multiuser machine, this is useful for determining who owns a component.
`-D | --getdirectory`	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 | --getport`	Displays the port of the component.
`-c | --getconnections`	This option is for use by technical support. It displays all the connections to the component.
`-a | --getall`	Displays all information for the component.
`-xs` *`id:id reason`* `| --disconnectsender` *`id:id reason`*	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. You can specify the component ID as *`physical_ID/logical_ID`*.
`-xr` *`id:id reason`* `| --disconnectreceiver` *`id:id reason`*	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. You can specify the component ID as *`physical_ID/logical_ID`*.
`-I` *`category`* `| --getinfo` *`category`*	This option is for use by technical support. It displays component-specific information for the specified category.
`-d | --deepping`	Ping the component. This confirms that the component process is running and acknowledging communications.

Option	Description
`-l level | --setloglevel level`	Sets the amount of information that the component logs in the component-specific log file. In order of decreasing verbosity, you can specify `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL`, `CRIT`, or `OFF`. **Note:** Setting the log level of the main correlator log file to anything other than `INFO` is discouraged. See the description of the `-v` (or `--loglevel`) option in "Starting the correlator" on page 78 for more details. If `--setloglevel` and `--setApplicationLogFile` both use the same log file, then the log file defined with `--setApplicationLogFile` is not changed.
`-r type arg* | --dorequest type arg*`	This option sends a component-specific request. For example: `engine_management -r cpuProfile frequency`. This returns the profiling frequency in Hertz. The following request types are available and apply to the correlator only: ■ `applicationEventLogging` — Sends detailed application information to the correlator log file. See "Viewing garbage collection and application events information" on page 170. ■ `codeCoverage` — Lets you check which lines in an EPL file have been executed. See "Recording code coverage information" on page 197. ■ `cpuProfile` — Lets you profile Apama EPL applications. See "Using the CPU profiler" on page 178. ■ `eplMemoryProfileOverview` — Returns information on all the monitors in the correlator. See "Using the EPL memory profiler" on page 172. ■ `eplMemoryProfileMonitorInstanceDetail` — Returns monitor instance details. See "Using the EPL memory profiler" on page 172. ■ `eplMemoryProfileMonitorDetail` — Returns aggregated monitor instance details. See "Using the EPL memory profiler" on page 172.

Option	Description
	■ `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. In addition, `flushAllQueues` also waits for any queued MemoryStore operations to complete, such as the preparation of a new store. ■ `flushChannelCache` — Notifies all `dynamicChainManagers` again (for example, Universal Messaging) about all channels which contexts are subscribed to or have sent to. This allows the manager to change its decision about whether it needs to subscribe to the channel (for example, when a channel has been created on Universal Messaging after the correlator was started). See also "Requirements of a transport chain manager plug-in class" in *Connecting Apama Applications to External Components*. ■ `setOOB` — Enables out of band notifications for a correlator. See Out of band connection notifications" in *Developing Apama Applications*. ■ `startInternalClock` — Starts the internal clocking of a correlator which was started with the `-Xclock` option (see "Determining whether to disable the correlator's internal clock" on page 100 for more information on the `-Xclock` option). `startInternalClock` first advances the time of all contexts to the current wall-clock time and then continues sending clock ticks at the configured

Option	Description
	frequency. It will do nothing if the internal clock is already running. ■ `toStringQueues` — Outputs the current contents of all input and output queues within the running correlator. This can be helpful for identifying slow senders/receivers and potential causes (such as very large events or excessive flow). ■ `verbosegc` — Enables logging of garbage collection events. See "Viewing garbage collection and application events information" on page 170. The following request type applies to the IAF only: ■ `getEventTypes` — Returns a string representation of the event types known to the running IAF. Certain other requests for the `-r` option are available for use by Apama technical support. See "Management requests" on page 168 for additional options.
`-s why | --shutdown 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 log file with a `CRIT` flag, and then shuts down.

Management requests

The options in the tables below replicate `-r` (or `--dorequest`) request types of the same name.

The following options are specific to the correlator:

Option	Description
`--rotateLogs`	Rotates all the log files. See "Rotating all correlator log files" on page 186.
`--setApplicationLogFile [node=]path`	Sets the log file for EPL log messages (global or per-package). For more information on how to set, get and unset the log file, see "Setting EPL log files and log levels dynamically" on page 182.
`--setApplicationLogLevel [node=]level`	Sets the log level for EPL log messages (global or per-package). For more information on how

Option	Description
	to set, get and unset the log level, see "Setting EPL log files and log levels dynamically" on page 182.
`--getApplicationLogFile` *node*	Displays the EPL log file for this node.
`--getApplicationLogLevel` *node*	Displays the EPL log level for this node.
`--getRootApplicationLogFile`	Displays the root EPL log file.
`--getRootApplicationLogLevel`	Displays the root EPL log level.
`--unsetApplicationLogFile` *node*	Unsets the EPL log file for this node.
`--unsetApplicationLogLevel` *node*	Unsets the EPL log level for this node.
`--unsetRootApplicationLogFile`	Unsets the root EPL log file.
`--unsetRootApplicationLogLevel`	Unsets the root EPL log level.

The following option is specific to the correlator and the IAF:

Option	Description
`--setLogFile` *path*	Instructs the component to close the component-specific log file it is using and to open a new log file with the name you specify. This has no effect on EPL logging which uses a separate log file. See "Rotating specified log files" on page 187 and "IAF log file rotation" in *Connecting Apama Applications to External Components*.

The following option is specific to the IAF:

Option	Description
`--reopenLog`	Reopens the log file of the component. See "IAF log file rotation" in *Connecting Apama Applications to External Components*.

Exit values

The `engine_management` tool returns the following exit values:

Value	Description
`0`	All status requests were processed successfully.

Value	Description
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.

Viewing garbage collection and application events information

The information in this topic applies to the correlator only.

You can enable logging of verbose garbage collection and application events in different ways and at different times, as described below.

Using the engine_management tool to enable garbage collection and application event logging for a running correlator

A handy way to view garbage collection (GC) information for a running correlator 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, where the garbage collection messages are logged at `DEBUG` level and are signposted by a prefix, `<apama.verboseGC.MonitorName>`, where *`MonitorName`* indicates the monitor where garbage collection occurred.

To disable logging of garbage collection information, execute the following command:

```
engine_management -r verbosegc off
```

The above commands provide an alternative to the following command, which provides a great deal of detailed output (such as context-state changes, event triggering, spawning, routing, etc.) in addition to garbage collection information:

```
engine_management -r applicationEventLogging on
```

Again, this output goes to the correlator log, with the messages being logged at `DEBUG` level. Any additional log messages to the garbage collection messages are signposted by the prefix `<apama.applicationEvents>`.

To turn this off, execute the following command:

```
engine_management -r applicationEventLogging off
```

See "Shutting down and managing components" on page 161 for more information on the `-r` (or `--dorequest`) option of the `engine_management` tool.

Enabling garbage collection and application event logging at correlator startup

You can also enable garbage collection logging at correlator startup. To do so, execute the following command:

```
correlator --loglevel apama.verboseGC=DEBUG
```

This enables verbose garbage collection logging for all monitors.

If you only wish to enable garbage collection logging for a particular monitor, append the name of the monitor (for example, "MyMonitor") to `apama.verboseGC` as follows:

```
correlator --loglevel apama.verboseGC.MyMonitor=DEBUG
```

This enables verbose garbage collection logging only for the monitor which has the name "MyMonitor".

Similarly, you can also enable application event logging at start time by starting a correlator with the following command:

```
correlator --loglevel apama.applicationEvents=DEBUG
```

Note:

If you enable application event logging at start time in this fashion, this will not enable verbose garbage collection logging (as is the case when enabling it using the `engine_management` request). Furthermore, it is not possible to enable application event logging on a per-monitor basis, as it is for garbage collection logging.

See "Starting the correlator" on page 78 for more information on the `--loglevel` (or `-v`) option of the `correlator` command.

Using a YAML configuration file to enable garbage collection and application event logging at correlator startup

Another way to enable garbage collection logging or application event logging at start time is through a YAML configuration file. For more information, see the descriptions of the following categories in "Setting correlator and plug-in log files and log levels in a YAML configuration file" on page 113:

- `apama.verboseGC`
- `apama.verboseGC.`*`MonitorName`*
- `apama.applicationEvents`

Using the EPL memory profiler

The information in this topic applies to the correlator only.

You use the EPL memory profiler to display information on monitors and monitor instances.

The EPL memory profiler is invoked using the `-r` (or `--dorequest`) option of the `engine_management` tool, followed by a request. Several requests are available for the EPL memory profiler, which are described below.

Important:
Do not use these requests on latency-sensitive applications. You should use them routinely only when developing or debugging.

When a request is issued, the correlator execution is momentarily paused to gather statistics.

The information that is returned for a request can be viewed directly (for example, in the Apama Command Prompt), or it can be written to a comma-separated values (CSV) file which can easily be viewed in tabular form using a tool such as Microsoft Excel.

Note:
All byte counts returned by a request are approximate values. The EPL memory profiler only shows memory usage that can be directly attributed to individual monitor instances. There are some parts of the correlator runtime that are not tracked, but these are typically small and fixed. Any memory used by Java or C++ plug-ins is not tracked. The profiler is useful in indicating the shape of the memory usage of an application - which monitors and event types are using more memory in proportion to the rest of the EPL runtime.

The values returned for the number of bytes and the number of EPL objects also include EPL objects that are no longer being used, and have not yet been garbage-collected. Therefore, the values will never be precise unless you are lucky enough to make this request just after garbage collection has run. See also "Garbage collection" in *Developing Apama Applications*.

The size of event expressions, including internal data structures associated with them, is excluded (and is typically small).

Each request returns the following string, in addition to the column headers described below:

`"Version:`*`version`*`, Snapshot time:`*`time`*`, Component ID:`*`id`*`, Host:`*`host-name`*`, Port:`*`port`*`, EPL memory:`*`bytes`*`"`

String element	Output
`Version:`*`version`*	Information on the correlator version.
`Snapshot time:`*`time`*	Time at which the EPL memory profiler has taken the snapshot. This is the date in milliseconds. The date “1446716459541”, for example, translates into “Thu Nov 05 2015 09:40:59” in UTC time.
`Component ID:`*`id`*	Correlator component ID.

String element	Output
`Host:host-name`	Name of the host on which the correlator is running.
`Port:port`	Port number on the above host.
`EPL memory:bytes`	Total memory used by the EPL types in the correlator.

Returning information on all monitors

The following command returns information on all the monitors in the correlator:

```
engine_management -r eplMemoryProfileOverview
```

This request does not take arguments. If arguments are passed, they are ignored.

The output shows the following information in the following order:

Column header	Information shown in this column
Monitor	The name of the monitor.
Monitor instances	The number of monitor instances.
EPL objects	The number of EPL objects created by the monitor instances (for example, dictionaries, events, sequences, and so on).
Listeners	The number of active listeners.
Bytes	The approximate number of bytes used by EPL objects created by the monitor instances.
Overhead bytes	The approximate number of bytes covering miscellaneous internals that the correlator maintains for book-keeping per monitor instance.

Returning monitor instance details

The following command returns information for all EPL types across the monitor instances of a specific monitor in the correlator:

```
engine_management -r eplMemoryProfileMonitorInstanceDetail monitor-name
```

where *monitor-name* is the name of the monitor.

You can also specify `all` to list the instance details of all monitors. The results are returned using the following sort order:

1. monitor name (ascending)
2. monitor instance ID (ascending)
3. EPL type (ascending)

The output shows the following information in the following order:

Column header	Information shown in this column
Monitor	The name of the monitor.
Persistent	`true` if the monitor is persistent. `false` it is not persistent.
EPL type	The type of the EPL object (see "Types" in the "EPL Reference", which is part of *Developing Apama Applications*) and also any active listeners. The output shows one entry for each listener. For example, if there is a monitor with one instance, and which has 2 listeners where each listener has 10 active instances, then the output will contain 2 rows. The number of EPL objects will then be 10 for each row.
Context name	The name of the context.
Context ID	The ID of the context.
Monitor instance ID	The ID of the monitor instance.
EPL objects	The number of EPL objects created by the monitor instances (for example, dictionaries, events, sequences, and so on).
Bytes	The approximate number of bytes used by EPL objects created by the monitor instances.

The output for the context is a combination of EPL type and monitor instance. For example, if there are 10 monitor instances where each instance has lots of objects of 3 different types, then the output will have 30 rows.

Unlike other EPL objects which belong to a single monitor instance, some strings are shared between several monitor instances. When a string is only used by a single monitor instance, it is shown like any other object in the output of the request, that is, with an EPL type of “string”. However, if the same string is shared between multiple monitor instances, then each monitor or monitor instance that is using it will show the EPL type as “string (shared)”. This is a performance optimization which avoids unnecessary copying. For example, a string may be shared in the following cases:

- When a monitor containing a string spawns to another monitor instance.
- When a monitor has a string that it sends inside an event to a monitor in another context.
- When an input event containing strings is received by multiple monitor instances which then store these strings.

One of the implications of sharing is double-counting, for both the number of EPL objects and the number of bytes. If multiple monitor instances refer to the same shared strings, the output of the request will include these numbers against each monitor instance separately. However, the duplication is eliminated when object sizes are summed up for the “EPL memory” value, so it may end up being notably lower than the sum of the “Bytes” in each row.

See “Handling of reference types” on page 175 for more information.

Returning aggregated monitor instance details

The following command is similar to `eplMemoryProfileMonitorInstanceDetail`, except that it aggregates the object count and size from each monitor instance, displaying data per monitor rather than per monitor instance.

```
engine_management -r eplMemoryProfileMonitorDetail monitor-name
```

where *monitor-name* is the name of the monitor. You can also specify `all` to list all monitors, sorted by the monitor name.

The output shows the following information in the following order:

Column header	Information shown in this column
Monitor	The name of the monitor.
Persistent	`true` if the monitor is persistent. `false` it is not persistent.
EPL type	The type of the EPL object (see "Types" in the "EPL Reference", which is part of *Developing Apama Applications*) and also any active listeners. The output shows one entry for each listener. For example, if there is a monitor with one instance, and which has 2 listeners where each listener has 10 active instances, then the output will contain 2 rows. The number of EPL objects will then be 10 for each row.
EPL objects	The number of EPL objects created by the monitor instances (for example, dictionaries, events, sequences, and so on).
Bytes	The approximate number of bytes used by EPL objects created by the monitor instances.

This request also takes account of shared strings. See the description of the `eplMemoryProfileMonitorInstanceDetail` request for details.

See "Handling of reference types" on page 175 for more information.

Handling of reference types

For reference types (such as sequence and dictionaries), the size is not reflected in the object referencing it. Instead, the size is associated with the actual object which is referenced. For example, if an event references/contains a sequence, the size of the sequence has no effect on the byte count of that event.

The `any` type does not show up in the EPL memory profiler output. Reference types held in an `any` value show up as if they were held in a value of their type. Boxed primitives show up as a separate type (with the first letter in capitals) in the EPL memory profiler output; for example:

```
monitorName,false,..Integer,main,1,1,2,240
monitorName,false,AnyContainer,main,1,1,2,240
```

In the above example, there are two `AnyContainer` objects, and there are two boxed primitives (the `..Integer` type). For both, there are two objects in memory and are consuming 240 bytes for each type.

Visualizing the EPL memory profiler information in Microsoft Excel

You can write the output from each of the above requests to a comma-separated values (CSV) file which can easily be viewed in tabular form using a tool such as Microsoft Excel.

The example below shows how to visualize the output of the `eplMemoryProfileMonitorInstanceDetail` request in Microsoft Excel using a pivot table.

1. Save the output of the request in a CSV file and open this file with Microsoft Excel.
2. Create a new **PivotTable** in Microsoft Excel, and in the resulting dialog select the range of data for the pivot table (ideally, the information for the entire range is already provided in the corresponding text box).
3. For this example, add the following fields to the report:

   Monitor

   EPL type

   Context name

   EPL objects

   Bytes

   If required, you can also add the following fields:

   Monitor instance ID

   Context ID

   The monitor instance ID is helpful if multiple monitor instances exist within the same context.

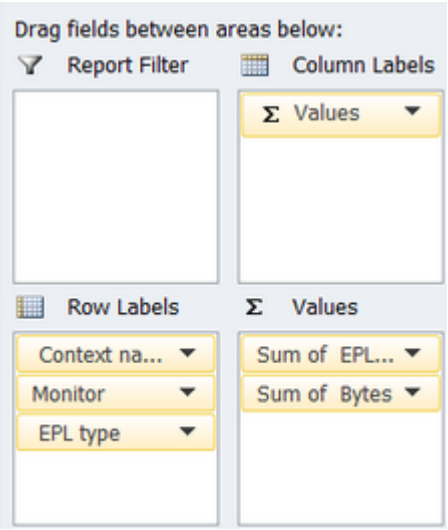
   After you have added the fields, you can see the following in the table:

   - row labels which include the monitor name, the context name and the EPL type, and
   - columns which sum up the number of EPL objects and the approximate number of bytes.

   For example:

Row Labels	Sum of EPL objec	Sum of Bytes
com.apama.primitive.PrimitiveMonitor	**38**	**25600**
main	**14**	**1952**
com.apama.primitive.Configure	1	120
com.apama.primitive.PrimitiveMonitor.Global Variables	1	136
context	4	384
Listener - PrimitiveMonitor.mon:29	1	192
Listener - PrimitiveMonitor.mon:31	4	800
Listener - PrimitiveMonitor.mon:40	1	200
sequence<float>	1	112
string	1	8
Primitive-Context1	**6**	**4840**
com.apama.primitive.Configure	1	120
com.apama.primitive.PrimitiveMonitor.Global Variables	1	120
Listener - PrimitiveMonitor.mon:48	1	192
Listener - PrimitiveMonitor.mon:52	1	192
sequence<float>	1	4208
string	1	8
Primitive-Context2	**6**	**17128**
com.apama.primitive.Configure	1	120
com.apama.primitive.PrimitiveMonitor.Global Variables	1	120
Listener - PrimitiveMonitor.mon:48	1	192
Listener - PrimitiveMonitor.mon:52	1	192
sequence<float>	1	16496
string	1	8
Primitive-Context3	**6**	**808**
com.apama.primitive.Configure	1	120
com.apama.primitive.PrimitiveMonitor.Global Variables	1	120
Listener - PrimitiveMonitor.mon:48	1	192
Listener - PrimitiveMonitor.mon:52	1	192
sequence<float>	1	176
string	1	8
Primitive-Context4	**6**	**872**
com.apama.primitive.Configure	1	120
com.apama.primitive.PrimitiveMonitor.Global Variables	1	120
Listener - PrimitiveMonitor.mon:48	1	192
Listener - PrimitiveMonitor.mon:52	1	192
sequence<float>	1	240
string	1	8
com.apama.reference.ReferenceTypeMonitor	**12139**	**1471080**
main	**15**	**2280**
com.apama.reference.Configure	1	120
com.apama.reference.ReferenceTypeMonitor.Global Variables	1	136

4. If you want to get an overview of the context level in the row labels, just drag the **Context name** label above the **Monitor** label as shown in the example below, and then check the changes in the report:



5. Similarly, if you want to see how the EPL objects are distributed over the different contexts and monitors, just move the **EPL type** to the very top of the row labels, followed by the **Monitor** and **Context name** labels.

6. Once the data is shown as wanted in the table, you can conditionally format the table to highlight individual columns, for example, to show high values or values that are above the threshold or above the average. Detailed information on how to do this can be found in the Excel help.

   A basic use case is to highlight the values for the object count and byte count that are above the average. To do so, select the **Sum of EPL objects** column and then choose the following command: **Conditional Formatting > Top/Bottom Rules > Above Average**. You can then select a formatting option from a dialog, for example, red text. As a result, all values in the cells of the **Sum of EPL objects** column that are above the average are shown in red. If you want, you can do the same for the **Sum of Bytes** column.

   You can also use additional conditional formatting (for example, color scales) to highlight the cells with values above the average.

Using the CPU profiler

The information in this topic applies to the correlator only.

Using the CPU profiler, you can profile applications written with EPL. 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. The CPU profiler that is described here allows you to obtain profile information without the overhead of Software AG Designer (see also "Profiling EPL Applications" in *Using Apama with Software AG Designer*).

The CPU profiler is invoked using the `-r` (or `--dorequest`) option of the `engine_management` tool:

```
engine_management -r cpuProfile argument
```

where *`argument`* can be one of the following:

Argument	Description
`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: CPU, Plug-in, Blocked, or Runnable.
- **Plug-in.** The correlator is executing a plug-in 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 `cpuProfile get` or `cpuProfile 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,Plug-in,Blocked,Total ticks:573
3,3,processor:processor.::listenerAction::,create-state.mon:
50,556,293,0,556,0,0,263,0
```

In the above 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 plug-in (263 samples). Each context spent similar amounts of time executing EPL and executing plug-ins but in different listeners (notice the different line numbers).

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

Column header	Information shown in this column
Context ID	ID of the context. A context ID is not present in data returned by `-r cpuProfile gettotal`.
Context name	Name of the context. A context name is not present in data returned by `-r cpuProfile 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 coassignment 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.

Column header	Information shown in this column
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.
CPU time	Number of samples in which the correlator is executing the location/action and is not in a plug-in (see Plug-in 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, Plug-in, 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.
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.
Plug-in	Number of samples in which the location is executing an EPL 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.

The `cpuProfile` request returns the following string, in addition to the column headers described above:

```
"Version:version, Snapshot time:time, Profile start time:time, Component ID:id,
Host:host-name, Port:port"
```

String element	Output
`Version:`*`version`*	Information on the correlator version.
`Snapshot time:`*`time`*	Time at which the CPU profiler has taken the snapshot. This is the date in milliseconds. The date “1446716459541”, for example, translates into “Thu Nov 05 2015 09:40:59” in UTC time.
`Profile start time:`*`time`*	Time at which the CPU profiler has been started. This is the date in milliseconds.
`Component ID:`*`id`*	Correlator component ID.
`Host:`*`host-name`*	Name of the host on which the correlator is running.
`Port:`*`port`*	Port number on the above host.

Setting EPL log files and log levels dynamically

This topic describes how to configure logging for individual EPL packages. It applies to the correlator only. For information about configuring the log level of the whole correlator and plug-ins running inside it, see “Setting correlator and plug-in log files and log levels in a YAML configuration file” on page 113.

You can configure per-package logging in several ways:

- Dynamically, using the following options of the `engine_management` tool as described in this topic:

  ```
  --setApplicationLogFile

  --setApplicationLogLevel
  ```

- Statically, in a YAML configuration file when starting the correlator. See “Setting EPL log files and log levels in a YAML configuration file” on page 111 for detailed information.
- Dynamically from within EPL. See "Using the Management interface" in *Developing Apama Applications* for detailed information.

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` tool 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 topics below provide information for managing individual log files and log levels.

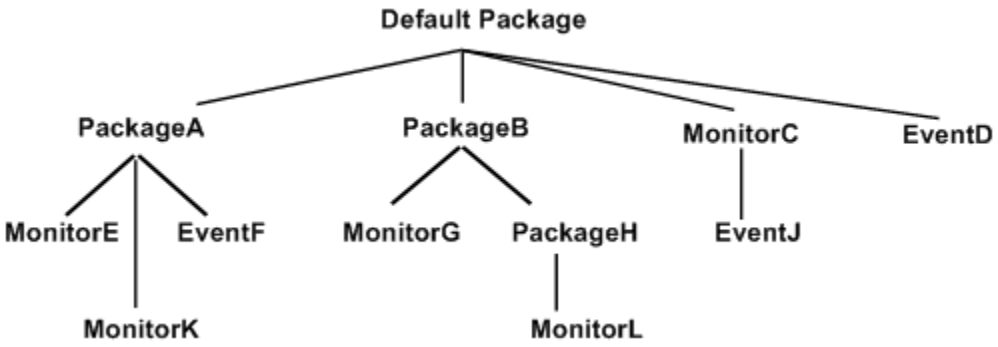
See also "Rotating correlator log files" on page 185.

Note:

Regularly rotating log files and storing the old ones in a secure location may be important as part of your personal data protection policy. For more information, see "Recommendations for logging by Apama application code" in *Developing Apama Applications*.

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.



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 EPL log levels" on page 183.

Managing EPL log levels

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

```
engine_management --setApplicationLogLevel [node=]logLevel
```

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

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

```
engine_management --getApplicationLogLevel [node]
```

If you do not specify a node, the tool 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 tool 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 --unsetApplicationLogLevel [node]
```

To manage the log level for an event that you define in a monitor, see "Managing event logging attributes" on page 185.

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 *Developing Apama Applications*.

Managing EPL log files

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

```
engine_management --setApplicationLogFile  [node=]logFile
```

Option	Description
node	Optionally, specify the name of a package, monitor or event. If you do not specify a node name, the tool associates the log file with the default package.
logFile	Specify the path of the log file. You specify the name of an EPL log file in the same way that you specify the name of a main correlator log file or input log file. See "Specifying log filenames" on page 90.

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

```
engine_management --getApplicationLogFile [node]
```

If you do not specify a node, the tool 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 tool 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 --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` tool and specify an unqualified event type name. Do not specify an enclosing scope, such as `com.apamax.myMonitor.NestedEventType`. For example:

```
engine_management --setApplicationLogFile NestedEventType=foo.log
engine_management --setApplicationLogLevel NestedEventType=DEBUG
```

Rotating correlator log files

Rotating a correlator log file refers to closing a log file being used by a running correlator and opening a new log file to be used instead from that point onwards. This lets you archive log files and avoid log files that are too large to easily view.

Each site should decide on and implement its own correlator log rotation policy. You should consider the following:

- How often to rotate log files.
- How large a correlator log file can be.

- What correlator log file naming conventions to use to organize log files.

There is a lot of useful header information in the main log file being used when the correlator starts. If you need to provide log files to Apama technical support, you should be able to provide the log file that was in use when the correlator started, as well as any other log files that were in use before and when a problem occurred.

Note:

Regularly rotating log files and storing the old ones in a secure location may be important as part of your personal data protection policy. For more information, see "Protecting and erasing data from Apama log files" in *Developing Apama Applications*.

To rotate the correlator log file and also rotate any other log file the correlator is using (input log file, EPL log files), see "Rotating all correlator log files" on page 186.

To rotate only the main correlator log file, see "Rotating specified log files" on page 187.

Note:

Some people use the term "log rolling" instead of "log rotation".

Rotating all correlator log files

The information in this topic applies to the correlator only.

To invoke rotation of all log files that the correlator is using, you can do the following:

- Invoke the `engine_management` tool and specify `--rotateLogs`.

  This rotates the main correlator log file, the correlator input log file if it is being generated, and any EPL log files that are being generated. When you invoke this management request then the correlator closes each log file it was using.

  If the log filename specification declared `${START_TIME}`, `${ROTATION_TIME}` and/or `${ID}`, then the correlator starts new log files with updated names according to the log filename specification; for example, if `${ID}` was specified, then the ID portion of a log filename would be incremented by 1.

- In EPL, create a monitor that uses the Management interface EPL plug-in to trigger log rotation on a schedule. See "Using the Management interface" in *Developing Apama Applications*.

- On UNIX only, you can write a `cron` job that periodically sends a `SIGHUP` signal to Apama processes.

  The standard UNIX `SIGHUP` mechanism causes Apama processes to re-open their log files. If the log file names were specified with `${ROTATION_TIME}` and/or `${ID}`, then the re-opened files have names that contain the rotation time and/or the incremented ID.

If you want a log filename to always be the same and so did not declare `${START_TIME}`, `${ROTATION_TIME}` or `${ID}` in the log filename specification, then the correlator starts new log files that have the same names as the log files it closed. On Windows, this would overwrite the closed

log files, so you must move the log files before invoking rotation. On UNIX, log files are appended to if the names are the same.

Rotating specified log files

Run one of the following utilities to rotate a particular log file. On Windows, set up scheduled tasks that run the utilities. On UNIX, write a `cron` job that periodically runs the utilities. The behavior is the same on both Windows and UNIX, except as noted.

- The following command (which is available for both the correlator and the IAF) instructs the component to close its component-specific log file and start using a new log file that has the name you specify. If the name of the file contains blanks, be sure enclose it in quotation marks.

  ```
  engine_management --setLogFile log-filename
  ```

- The following command (which is only available for the correlator) instructs the correlator to use the specified file as the log file for the specified node, which can be a package, monitor, or event. See also "Setting EPL log files and log levels dynamically" on page 182.

  ```
  engine_management --setApplicationLogFile [node=]log-filename
  ```

  If you use separate log files for particular packages, monitors, or events you might want to rotate those log files at the same time that you rotate the main correlator log file. This keeps your Apama log files in sync with each other. See "Rotating all correlator log files" on page 186.

On Windows, when you rotate a log file, you must ensure that the new log filename is different from the name of the log file that was in use. Apama takes care of this for you if you specify `${ROTATION_TIME}` and/or `${ID}` in the *log-filename* specification. If the name is not different, the old file is overwritten. If you want to use the same log filename, then you must move the file before you rotate it.

On UNIX, a log file is never overwritten. If you rotate a log file and specify the same name, then Apama appends messages to the content already there.

Apama does not support automatic log file rotation based on log file size.

The only way to rotate the correlator input log is to rotate all correlator log files. See "Rotating all correlator log files" on page 186.

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. The executable for this tool is located in the `bin` directory of the Apama installation.

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 Software AG Designer 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`

tool 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.

Synopsis

To debug applications on a running correlator, run the following command:

```
engine_debug [ [ global-options ] [command [options]] ... ]
```

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:

Abbreviation	Command	Description
`h [command]`	`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.
`ha`	`hashes`	Lists injected files and their hashes.
`si`	`stepinto`	Steps into an action.
`sot`	`stepout`	Steps out of an action.
`sov`	`stepover`	Steps over an instruction.
`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 195 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.

Abbreviation	Command	Description
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.
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

Abbreviation	Command	Description
		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 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

Abbreviation	Command	Description
		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*.

Exit status

The `engine_debug` tool returns the following exit values:

Value	Description
`0`	Success. All requests were processed successfully.
`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.

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, if you want to find out what the `status` command does, enter the following:

```
engine_debug help status
```

Or to find out which options you can specify with the `breakpoint add` command, enter the following:

```
engine_debug help breakpoint add
```

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 enable and disable 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
```

Working with breakpoints using the command-line debugger

You can use the command-line debugger to add, list and remove breakpoints.

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
```

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.

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 188.

Command	Abbreviation
`--action`	`-a`
`--breakonce`	`-bo`
`--breakpoint`	`-bp`
`--context`	`-c`
`--file`	`-f`
`--frame`	`-fm`

Command	Abbreviation
`--host`	`-n`
`--instance`	`-mt`
`--line`	`-l`
`--owner`	`-o`
`--port`	`-p`
`--raw`	`-R`
`--timeout`	`-to`
`--utf8`	`-u`
`--variable`	`-v`
`--verbose`	`-V`

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[setupctx(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::[setupctx(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
```

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
```

Generating code coverage information about EPL files

The correlator can generate "code coverage" information about EPL files indicating which lines have been executed. This is useful for measuring the quality of test cases, discovering lines of EPL code which are not being exercised by any tests, as well as for helping diagnose bugs or understand complex interactions in the EPL.

Recording code coverage information

The recording of code coverage information can be enabled and written (dumped) to disk using management requests, or using an environment variable that automatically writes out a coverage file when the correlator is shut down or when code is deleted from the correlator.

The `epl_coverage` tool can then be used to merge together the coverage files that have been produced by the correlator and produce summary statistics about how much of each source file is covered, as well as an HTML report where each source line is shown annotated with different colors to indicate which lines are not being covered. For detailed information, see "Creating code coverage reports" on page 199.

Enabling the code coverage feature will disable the compiled runtime, and it will also enable the debugger ("Using the command-line debugger " on page 187) and CPU profiler (see "Using the CPU profiler" on page 178).

Dumping code coverage information using management requests

One way to enable and dump code coverage information is via the `-r codeCoverage` option of the `engine_management` tool (see also "Shutting down and managing components" on page 161). You can send the following requests:

Request	Description
`codeCoverage on`	Enables the recording of code coverage information. This also disables optimizations for any subsequently injected files, disables use of the compiled runtime and enables the EPL debugger. Code coverage must be enabled before injecting EPL to record code coverage information. EPL injected before code coverage is enabled will be omitted from the coverage report (unless using the environment variable as described below). **Note:** This option is not suitable for production use.

Request	Description
`codeCoverage off`	Disables the recording of code coverage information. This also removes any in-memory coverage information stored so far, but does not reset any features changed by `codeCoverage on` such as optimizations and possibly the compiled runtime.
`codeCoverage dump [filename]`	Returns the code coverage information either for all EPL files in the correlator or just for the (optional) source EPL filename provided. The output format is suitable for input to the `epl_coverage` tool, and is encoded as a UTF-8 string.

Automatically writing code coverage information using an environment variable

It is also possible to start the correlator in a mode where it automatically writes code coverage information to disk when it is shut down or is given an `engine_delete --all` request (see also "Deleting code from a correlator" on page 131).

This mode is enabled by setting the `AP_EPL_COVERAGE_FILE` environment variable to the path of a file to which coverage information is to be written. If you do this, the correlator starts in code coverage collection mode with debugging enabled, the compiled runtime disabled and optimizations disabled. On shutdown, it writes the code coverage information to the path specified in the environment variable.

The environment variable can contain replacement tokens in the same format as the correlator log file (see "Specifying log filenames" on page 90). Given that the coverage file is not subject to log rotation, only the `${PID}` and `${START_TIME}` tags are appropriate.

Example (for Windows):

```
set AP_EPL_COVERAGE_FILE=c:\mypath\mycorrelator.${PID}.eplcoverage
start correlator
(run application, etc.)
engine_management --shutdown "Clean correlator shutdown from command line"
```

Of course, the correlator must be cleanly shut down for this to work, as no coverage information is written if the process is terminated without warning. If a dump is triggered by `engine_delete --all` and more EPL is then injected before the correlator is shut down, all coverage information written by `engine_delete` is overwritten by later coverage information and is thus lost. However, if `engine_delete` is immediately followed by a clean shutdown, there will be no new coverage information when the shutdown occurs. Therefore, the file will not be overwritten.

Code coverage and deletion of monitors

The code coverage information for transient monitors, monitors which have died, and monitors which were deleted by name is retained. This enables you to later call `codeCoverage dump` to get the coverage information.

An `engine_delete --all`, however, resets all the coverage information. When you set the `AP_EPL_COVERAGE_FILE` environment variable, the coverage information is automatically dumped during an `engine_delete --all`.

If you delete a monitor and then reinject the EPL file, then all coverage information for that EPL file is reset for the newly injected file.

Common usage patterns

- Enable code coverage, inject your application and send typical events into the correlator. Then dump a coverage report. This gives you a complete list of code covered by initialization and events being processed in the system.
- Set the `AP_EPL_COVERAGE_FILE` environment variable before running your test suite. Then collate all the coverage reports. This lets you check that your tests exercise all the code paths.
- Enable code coverage and inject your application. Then disable and enable code coverage (to clear the reporting data). Then send a single event through and dump a coverage report. This lets you see what code is run by a single event.

Creating code coverage reports

The `epl_coverage` tool takes one or more coverage files that have been output by the correlator's code coverage feature, merges them together to create a new combined `.eplcoverage` file (which can be used as input for the tool), and creates a CSV, XML and HTML report of the coverage of each source EPL file. The executable for this tool is located in the `bin` directory of the Apama installation.

Synopsis

To create code coverage reports, run the following command:

```
epl_coverage [ options ] file1.eplcoverage  [ file2.eplcoverage ... ]
```

Example (for Windows):

```
epl_coverage --output c:\mycoverage --source "%APAMA_WORK%\projects\myproject"
  --exclude "**/Apama/**/*.mon" *.eplcoverage
```

When you run this command with the `-h` option, the usage message for this command is shown.

Description

The `--output` argument specifies the directory into which the tool writes the output files. If not specified, the current directory is used.

The output includes the following files:

- **merged.eplcoverage**. A single file containing the combined EPL code coverage information from all the input files. This can be used as input to another invocation of the `epl_coverage` tool.
- **coverage_summary.csv**. A summary of the percentage of lines and instructions covered in each source file in the standard “comma-separated values” text format (in the operating system's local character encoding). This file may be useful for reviewing coverage information in a

spreadsheet, or as input for an automated tool that records coverage information as part of a continuous integration build/test system. The file starts with a header line beginning with the hash (#) character which identifies the columns used in the rest of the file. It is recommended that any tool that reads this file should use the header line to identify the contents of each column; this is helpful in case columns are added or reordered in a later release.

- **epl_coverage.xml**. An XML representation of the combined code coverage information for all the input files. This file is written in a widely-used coverage file format that can be read by many third-party tools (the file format that was popularized by Cobertura, which is a code coverage utility for Java; see also https://cobertura.github.io/cobertura/).
- **index.html** (and associated `.css` and `.html` files). An HTML summary of coverage information, including annotated copies of the source files showing which executable lines are covered. These files are always written in the UTF-8 encoding.

Where possible, you should always specify the directories of the `.mon` source files using the `--source` option. The coverage tool uses the source directories to identify the canonical location of each EPL file, which may not be the same as the path it was injected from, especially when initializing a correlator from a correlator deployment directory. This also makes it easier for third-party coverage tools to recognize the paths from the XML report and allows the HTML report to show the source code even when the `.mon` file was injected from a test output directory that no longer exists.

By default, EPL files are assumed to be in the UTF-8 (or ASCII) encoding. If your EPL files are in a different encoding, you can provide the `--source-encoding local` option, in which case the EPL files are read in the local encoding instead. Files with a UTF-8 byte-order marker are read in UTF-8 in either case. This affects both locating the correct source file in the provided `--source` directories, and correctly rendering the source code into the HTML reports.

You can apply filters that remove information about unwanted EPL files (such as test `.mon` files) from all of the output files, or to restrict which parts of the source directories are to be searched. The `--include` and `--exclude` options can each be specified multiple times. They specify file patterns to include or exclude (for example "**/foo/Bar*.mon"). These patterns use the following characters:

- forward slashes (/) to indicate directory separators (on all platforms),
- a single asterisk (*) to indicate any number of non-directory separator characters,
- two asterisks (**) to indicate any number of characters potentially including directory separators, and
- a question mark (?) to indicate a single character.

If no `--include` argument is provided, the default is to include all file paths, except those that are removed by `--exclude` arguments.

When the number of coverage input files is large, you can avoid an extremely long command line (which some operating systems do not support) by putting the coverage file list into a newline-delimited UTF-8 text file and providing the path to that file on the command line instead, prefixed with an `@` symbol. For example:

```
epl_coverage "@c:\mypath\coverage_file_list.txt"
```

Options

The `epl_coverage` tool takes the following options:

Option	Description
`-h | --help`	Displays usage information.
`-V | --version`	Displays version information for the `epl_coverage` tool.
`-o dir | --output dir`	Specifies the directory into which the tool writes the output files. If not specified, the current directory is used.
`-i pattern | --include pattern`	Filtering option for the coverage data and source directories which specifies the EPL files to include (defaults to `**`). This option can be specified multiple times. It matches against absolute paths, so filters should either start with an absolute path or with a `**/` wildcard.
`-x pattern | --exclude pattern`	Filtering option for the coverage data and source directories which specifies the EPL files to exclude (for example, `**/tests/` or `c:/foo/Bar.mon`). This is useful for avoiding unwanted coverage information for EPL files used in testing but not part of the application itself. This option can be specified multiple times. It matches against absolute paths, so filters should either start with an absolute path or with a `**/` wildcard.
`-s dir | --source dir`	Specifies one or more directories which will be recursively searched for `.mon` files and used to identify the canonical location of EPL files regardless of what directories they were injected from during testing.
`--title str`	HTML report option which specifies the title to write into the HTML file.
`--source-encoding [local|utf-8]`	The encoding to assume for EPL files. All files with a byte-order marker will use that encoding. All other files are assumed to be the given encoding (local or UTF-8). The default is `utf-8`.

Interpreting the code coverage reports

Many lines in an EPL file do not contain any executable instructions, for example, comments, event definitions (except where they contain actions) and event expressions used to declare listeners. These lines are not marked up by the `epl_coverage` tool.

Lines that do contain executable code may have one or more executable elements (instructions), and the `epl_coverage` tool reports whether all or only some of those instructions have been executed. It may therefore be useful to split complex EPL constructs (such as multi-part `if` statements) over

multiple lines as much as possible to make the output clearer as to what is covered. The exact details of how many instructions are on any given line is subject to change and therefore not documented, but information on partial coverage may sometimes be useful for identifying branching constructs where not all branches are covered.

Tip:

Every executable line that is not fully covered has the `(!)` string in the margin, which makes it possible to jump backwards and forwards between these lines using the Find functionality provided by most web browsers.

The purpose of the coverage information is to provide insight into areas of user EPL that are being missed by test cases. Although it is worth aiming for a high percentage of lines and instructions being covered, it is not always possible to write tests that cover every line. However, as long as someone looks at the lines that were missed, there is no need to worry about having less than 100 percent coverage.

Similarly, the information about partial line coverage can often be useful, particularly for control constructs where it might indicate a missed branch in an `if` statement, or a `while` loop condition that always returns `false`. But it will not always be possible for users to get 100 percent coverage of every line, or (since the internal instructions used by EPL are not documented and may be changed at any time between versions) even to understand the reason why a line was not fully covered in some cases. Software AG support cannot provide explanations for why a given line of EPL was only partly covered.

Examples

The following code snippets illustrate some common cases.

- The following line is partially but not fully covered if `a()` returns `true` every time this line is executed, since the instructions for the value of `b()` are never checked in this case.

  ```
  if a() or b() {
  ```

- The following line is only partially covered unless the test is run with `DEBUG` logging enabled, since expressions in log statements are only evaluated if the log level is specified.

  ```
  log "Hello world" at DEBUG;
  ```

- Another common example is a stream query that uses an aggregate where nothing drops out of the window while the test is executed. For example, if less than 100 seconds pass after the first `E()` event, the following line is only partially covered:

  ```
  from a in all E() within 100.0 select com.apama.aggregates.sum(a.val) as i
  ```

  If the test does not have anything drop out of the `within` window, then you will get amber coverage, as no code to remove a value from the set being aggregated over (by `sum`) is being executed. This may happen if no events go through this query, or if only less than 20 seconds pass since the first event.

- Any code in an `onunload` action will never be covered at all, since it is only executed with `engine_delete`, which also removes the coverage information.

Using EPL code coverage with PySys tests

The Apama installation includes the Python-based PySys test framework and Apama helper classes for PySys.

The Apama helper classes for PySys can enable code coverage recording, and automatically run a coverage report from the `.eplcoverage` files at the end of test execution, which will help users to create better test cases and to find code paths in their EPL applications that do not have adequate test coverage. To use this feature, start your tests with `-XcodeCoverage`, for example:

```
pysys run -XcodeCoverage
```

For an example, see the `README.txt` file in the `samples/pysys` directory of your Apama installation.

Replaying an input 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 an input log. An input 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 input log. Incoming messages include the following:

- Events
- EPL
- Java
- Correlator deployment packages (CDPs)
- Connection, deletion, and disconnection requests

If you are unable to diagnose the problem, you can provide the input log to Software AG Global Support. A support engineer can then feed your input log into a new correlator to try to diagnose the problem.

The information in the following topics describes how to generate and use an input log. See also "Examples for specifying log filenames" on page 92.

Regularly rotating log files and storing the old ones in a secure location may be important as part of your personal data protection policy. For more information, see "Handling personal data "at rest" in the correlator input log file" in *Developing Apama Applications*.

Creating an input log file

To create an input log, specify the following option when you start a correlator:

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

You specify the name of an input log file in the same way that you specify the name of a main correlator log file. See "Specifying log filenames" on page 90.

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

Rotating an input log file

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

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

In these situations, you can rotate the input log. Rotating the input log means that the correlator closes the current input log and starts sending messages to a new input log.

You should rotate the input log only when you have a specific need to do so. You do not want to have thousands of input logs in a directory since file systems do not handle this efficiently.

If you plan to rotate input logs, specify the `${ID}` tag when you specify the `--inputLog` option when you start the correlator. For examples, see "Examples for specifying log filenames" on page 92.

To rotate the input log, invoke the `engine_management` tool and specify the `--rotateLogs` option. The name of the new input log is the same as the name of the closed input log except that the correlator increments the ID portion of the input log filename by `1`. See "Rotating all correlator log files" on page 186.

Performance when generating an input log

When the file system that hosts the input log is fast, generating an input log should not have any noticeable effect on correlator performance in most cases. It is possible to use the input log with connectivity plug-ins (see "Using Connectivity Plug-ins" in *Connecting Apama Applications to External Components*), but the performance impact will be significant for chains using the `apama.eventMap` host plug-in and any chains that are using small batches of events. Consequently, the recommendation is to always run correlators that send information to input logs. Just make sure you have enough disk space for the input log. You need to monitor repeated use to determine how much space is required.

With the correlator generating an input log, you can implement your application so that it sends a minimum amount of information to the main correlator log file. You do not need to log application information because you can always recover application information from the input log. Implementing an application that sends large amounts of application information to the main correlator log file can negatively impact performance.

Reproducing correlator behavior from an input log

To use an input log to reproduce correlator behavior, you must do the following:

1. Run the `extract_replay_log` Python utility.

2. 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, you probably already have Python installed.

The format for running the `extract_replay_log` utility is as follows:

```
extract_replay_log.py [options] inputLogFile
```

Replace *`inputLogFile`* with the path for the input log you want to extract. If you specify the first input log in a series, the subsequent input logs must be in the same directory as the first input log.

The options you can specify are as follows:

Option	Description
`-o=`*`dir`* `|` `--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`* `|` `--lang=`*`lang`*	Specifies the language of the script that the `extract_replay_log` utility generates. Replace *`lang`* with one of the following: ■ `shell` to generate the `replay_execute.sh` UNIX shell script. ■ `batch` to generate the `replay_execute.bat` Windows batch file. This is the default.
`-c | --correlator`	Specifies that the script that `extract_replay_log` generates should include the command line for starting a correlator. When you run the generated script, the correlator will be started with all of the command-line options needed to replay the input log.
`--licence`	Specifies a path to a license file for starting a correlator.
`--port`	Specifies a port on which to start the correlator.
`-v | --verbose`	Indicates that you want verbose utility output.
`-h | --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 input log.
- Event files where each one is prefixed with “replay_”.

- EPL and possibly JAR and correlator deployment package (CDP) files where each one is prefixed with “replay_”.

Invoking the replay script

Before you run the replay script, you can optionally 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()`, `rand()` or `incrementCounter`
  - `send`, `emit`, `spawn...to`, 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 input 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 in the input log.

After 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 input log into the correlator running on the default port.

While the correlator exactly reproduces the activity captured in the input 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 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. For detailed information on the `engine_send` tool, see “Sending events to correlators” on page 136.

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 correlator that is executing the `engine_send` tool. For detailed information on the `engine_receive` tool, see "Receiving events from correlators" on page 139.

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.

The specific EPL types and how they map from the event representation are shown in an example in "Event types" on page 210, but there are certain basic types that can be included as shown in the following example:

```
// integer
MyEvent(-1,1)

// decimal and float
MyEvent(-2.0,2.0)

// decimal and float in exponential form (0.02,200)
MyEvent(-2.0e-2,2.0e2)

// string
MyEvent("three")

// boolean
MyEvent(true,false)
```

Both `decimal` and `float` types can be represented in scientific form if required, including when nested in `optional` or `any` types and inner events.

A `string` is a sequence of characters enclosed in double quotes ("). The backslash character (\) is used as an escape character to allow inclusion of special characters such as newlines and horizontal tabs.

To include this character in a string	use this notation
Double quote	`\"` This makes sure that the quote is not treated as the end of the string literal.
Newline	`\n`
Tab	`\t`
Backslash	`\\` Use two backslashes if you want to include a single backslash in the string. The compiler will remove any extra backslashes.

Examples:

```
MyEvent("Hello, World!")
MyEvent("\ta\tstring\twith\ttabs\tbetween\twords")
MyEvent("a string on\n two lines")
MyEvent("a string with \\ a backslash and a \" quote")
```

Localization, such as different formats for decimals or quotation marks, is not supported.

Each event is given on a separate line. Only single-line comments are allowed. Start each comment line with `//` or `#`. Any blank lines are ignored.

For example, following are three valid events:

```
// This is an event file that contains some sample events.
// Here are three stock price events:
my.test.StockPrice("XRX",11.1)
my.test.StockPrice("IBM",130.6)
my.test.StockPrice("MSFT",70.5)
```

For those events, the following event type definition must be injected into the package `test`:

```
package my.test;

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 several ways:

- Specifying that batches of events should be sent at specified time intervals.
- Specifying that all events on all queues should be processed before sending the next event.

Event timing

In `.evt` files, it is possible to specify the following:

- Time intervals for sending batches of events to the correlator.
- Waiting for all events on all queues at that point in time to be processed before sending the next event.

Adding `BATCH` tags to send events at intervals

You can specify time intervals for sending batches of events to the correlator. This is achieved by specifying the `BATCH` tag followed by a time offset in milliseconds. For example, the following specifies two batches of events to be sent 50 milliseconds apart.

```
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)
```

The addition of a "time" allows simulations of "bursts" of events, or more random distributions of event traffic. Times are measured as an offset 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 (that is, 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.

Using `&FLUSHING` mode for more predictable event processing order

Sending events in flushing mode can help provide a more predictable event processing order. However, flushing mode is slower than the default behavior.

By default, events are delivered in an optimal way, not waiting for previously sent events to be processed before the next event is delivered to contexts (or other consumers of channels). When flushing mode is enabled the behavior is as follows:

1. The correlator sends an event.
2. The correlator processes all events on all queues at that point in time, repeating this as many times as specified in the flushing specification.
3. The correlator sends the next event.

To enable flushing mode, insert the following line in a `.evt` file:

```
&FLUSHING(n)
```

Replace *n* with an integer that specifies how many times to flush queues in between each event. Set this to the maximum length of a chain of send-to operations between contexts that could occur in your application. If you specify a number that is bigger than required the correlator simply repeats the flush operation, which incurs a small overhead. To disable flushing mode, insert the following line in the `.evt` file:

```
&FLUSHING(0)
```

Enabling or disabling flushing mode affects only the events sent on that connection or from that event file.

When sending `&TIME` events in to a multi-context application, the time ticks are delivered directly to all contexts. This can be different than the way in which events in the `.evt` file are sent in to the correlator and then sent between contexts in an application. This difference can result in processing events at an incorrect simulated time. In these cases, sending `&FLUSHING(1)`, for example, before sending time ticks and events can result in more predictable and reliable behavior.

Event types

The following example illustrates how each 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;
  optional<Nested> opt;
  any anyValue;
}
```

the following is an expanded string 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
  location(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
  optional<Nested>(Nested(10)), // optional payload inside parentheses, may be
                                 // empty, in which case it is represented by the
                                 // string "optional()"
  any(integer,42)                // any names a type and the string form of that
                                 // type; other examples include the following:
                                 // any(sequence<Nested>,[Nested(1)])
                                 // may also be empty, in which case the string
                                 // form is: "any()"
);
```

Note:
This example is split over several lines for clarity. In practice, this definition must all be written on the same line, and without the comments. Otherwise the correlator will fail to parse the event.

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)}]
```

You can also get an event's string representation in EPL by using the `toString()` method.

Event association with a channel

The `engine_send` tool can send an event file that associates channels with events. Likewise, the `engine_receive` tool can output an event file that includes the channel on which an event was received. The event format is the same for both tools:

```
"channel_name",event_type_name(field_value1[, field_valuen]...)
```

For example, suppose you want to send `Tick` events, which contain a `string` followed by an `integer`, to the `PreProcessing` channel. The contents of the `.evt` file would look like this:

```
"PreProcessing",Tick("SOW", 35)
"PreProcessing",Tick("IBM", 135)
```

A channel name is optional. In a file being sent with the `engine_send` tool, you can mix event representations that specify channels with event representations that do not specify channels. Events for which a channel is specified go to only those contexts subscribed to that channel.

The default behavior is that events are sent on the default channel (the empty string) when a channel is not explicitly specified. Events sent on the default channel go to all public contexts. To change the default behavior for events sent by the `engine_send` tool, you can specify `engine_send -c channel`. If a channel is not explicitly specified for an event, then it is sent to the channel identified with the `-c` option . See "Sending events to correlators" on page 136.

Using the Data Player command-line interface

Note:
The Data Player is deprecated and will be removed in a future release.

Apama's Data Player in Software AG Designer 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. Software AG Designer 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 Software AG Designer.

To use the command-line interface to the Data Player, you must have already used the GUI interface in Software AG Designer. That is, you must have already defined queries and query configurations in Software AG Designer. When you use the command-line interface (that is, the `adbc_management` tool), you specify query names and query configurations that you created in Software AG Designer. The executable for this tool is located in the `bin` directory of the Apama installation.

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 with Software AG Designer*.

Synopsis

To use the Data Player from the command line, run the following command:

```
adbc_management --query queryName --configFile file [ options ]
```

When you run this command with the `-h` option, the usage message for this command is shown.

Options

The `adbc_management` tool takes the following options:

Option	Description
`-h | --help`	Displays usage information.
`-n host | --hostname host`	Name of the host on which the correlator is running. The default is `localhost`. Non-ASCII characters are not allowed in host names.
`-p port | --port port`	Port on which the correlator is listening. The default is `15903`.
`--query queryName`	Runs the specified query, which is defined in the query configuration file that you identify with the `--configFile` option. This is a query you created with Apama's Data Player in Software AG Designer. 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`	The query configuration file to use. This is the query configuration file associated with your project. In Software AG Designer, the query configuration file is always called `dataplayer_queries.xml` (in the project's `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 `Historical` followed by the name of the event in the container, for example, `HistoricalTick`. The container event will be in the default namespace. 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

Option	Description
	contained event, and reroute it such that application listeners can match on it.
`--backTest` *boolean*	This option is equivalent to the 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`.