Developing Clients

5.2.0

October 2014

Document ID: PAM-Developing_Clients-5.2.0-20141001@237655

Table of Contents

Preface

About this documentation

Apama applications that are to run within the event correlator can either be built natively in The Apama Event Processing Language or in Apama's in-process API for Java (JMon), or graphically through Apama's Event Modeler. (The Apama Event Processing Language is the new name for MonitorScript.)

Although Apama includes a suite of tools to allow EPL code to be submitted to the event correlator interactively, as well as submit events from text files, it is often necessary to go further and integrate the event correlator directly with other software. Often this is required in order to drive custom graphical user interfaces, or to deliver messages to and receive messages from the event correlator (such as market data and order management).

In environments that require the event correlator to be integrated with middleware infrastructure and data buses it is usually preferable to do this with Apama's Integration Adapter Framework (IAF). For information on developing adapters with the IAF, see The Integration Adapter Framework in *Developing Adapters*.

If your environment needs to interface programmatically with the event correlator, Apama provides a suite of Client Software Development Kits. These allow developers to write custom software adapters that interface applications, event sources and event clients to the event correlator. These adapters can be written in C, C++, or Java, or as .NET applications. In Java, adapters can be written in one of four interface layers: the Java Client API, the JavaBeans API, the EventService API, or the ScenarioService API.

This topic describes how to use these Client Software Development Kits.

Preface

Documentation roadmap

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

Apama provides documentation in three formats:

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

On Windows, to access the documentation, select Start > All Programs > Software AG > Apama 5.2 > Apama Documentation . On UNIX, display the `index.html` file, which is in the `doc` directory of your Apama installation directory.

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

Title	Contents
What's New in Apama	Describes new features and changes since the previous release.
Installing Apama	Instructions for installing the Developer, Server, or User Apama installation options.
Introduction to Apama	Introduction to developing Apama applications, discussions of Apama architecture and concepts, and pointers to sources of information outside the documentation set.
Using Apama Studio	Instructions for using Apama Studio to create and test Apama projects; write, profile, and debug EPL programs; write JMon programs; develop custom blocks; and store, retrieve and playback data.
Developing Apama Applications in Event Modeler	Instructions for using Apama Studio's Event Modeler editor to develop scenarios. Includes information about using standard functions, standard blocks, and blocks generated from scenarios.
Developing Apama Applications in EPL	Introduces Apama's Event Processing Language (EPL) and provides user guide type information for how to write EPL programs. EPL is the native interface to the correlator. This document also provides information for using the standard correlator plug-ins.
Apama EPL Reference	Reference information for EPL: lexical elements, syntax, types, variables, event definitions, expressions, statements.
Developing Apama Applications in Java	Introduces the Apama in-process API for Java, referred to as JMon, and provides user guide type information for how to write Java programs that run on the correlator. Reference information in Javadoc format is also available.
Building Dashboards	Describes how to create dashboards, which are the end-user interfaces to running scenario instances and data view items.
Dashboard Property Reference	Reference information on the properties of the visualization objects that you can include in your dashboards.
Dashboard Function Reference	Reference information on dashboard functions, which allow you to operate on correlator data before you attach it to visualization objects.

Title	Contents
Developing Adapters	Describes how to create adapters, which are components that translate events from non-Apama format to Apama format.
Developing Clients	Describes how to develop C, C++, Java, or .NET clients that can communicate with and interact with the correlator.
Writing Correlator Plug-ins	Describes how to develop formatted libraries of C, C++ or Java functions that can be called from EPL.
Deploying and Managing Apama Applications	Describes how to: • Use the Management & Monitoring console to configure, start, stop, and monitor the correlator and adapters across multiple hosts. • Deploy dashboards over wide area networks, including the internet, and provide dashboards with effective authorization and authentication. • Improve Apama application performance by using multiple correlators, and saving and reusing a snapshot of a correlator's state. • Use the Apama ADBC adapter to store and retrieve data in JDBC, ODBC, and Apama Sim databases. • Use the Apama Web Services Client adapter to invoke Web Services. • Use correlator-integrated messaging for JMS to reliably send and receive JMS messages in Apama applications. • Use Universal Messaging to connect correlators.
Using the Dashboard Viewer	Describes how to view and interact with dashboards that are receiving run-time data from the correlator.

Preface

Contacting customer support

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

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

Preface

Chapter 1: The Client Software Development Kits

Apama applications that are to run within the event correlator can either be built natively in the Apama Event Processing Language (EPL) or in JMon, or graphically through Apama's Event Modeler. (Event Processing Language is the new name for MonitorScript.)

Although Apama includes a suite of tools to allow EPL code to be submitted to the event correlator interactively, as well as submit events from text files, it is often necessary to go further and integrate the event correlator directly with other software. Often this is required in order to drive custom graphical user interfaces, or to deliver messages to and receive messages from the event correlator (like market data and order management).

In environments that require the event correlator to be integrated with middleware infrastructure and data buses it is usually preferable to do this with Apama's Integration Adapter Framework (IAF). For information on developing adapters with the IAF, see "The Integration Adapter Framework" in *Developing Adapters*.

If your environment needs to interface programmatically with the event correlator, Apama provides a suite of Client Software Development Kits. These allow developers to write custom software applications that interface existing enterprise applications, event sources and event clients to the event correlator. These custom applications can be written in C, C++, Java or .NET. In Java, applications can be written in one of four interface layers. The Java and .NET interfaces, in order of increasing abstraction are:

- Raw, low-level Java/.NET Client API — Base layer upon which the other Java/.NET API layers are built. This is equivalent to the C++ SDK.
- JavaBeans API/.NET Engine Client API — More powerful and provides extensive higher level functionality.
- EventService API — Use when attaching a listener to a named channel or when using events as a messaging transport for synchronous or asynchronous pseudo RPC mechanisms.
- ScenarioService API — Used to provide an interface to scenarios that have been built with Event Modeler.

This section of the documentation describes how to use these Client Software Development Kits.

Basic operations

Interfacing with the event correlator is straightforward. Conceptually there are six basic operations that are possible:

- Inject EPL code into an event correlator
- Send events into an event correlator
- Register as an event receiver with an event correlator in order to receive events from it

- Delete primary EPL entities; monitors and event definitions
- Interrogate an event correlator about the monitors and event definitions that have been defined within it (inspect)
- Transfer the entire run-time state of an event correlator to a file, or initialize an event correlator from a previous state transfer

A seventh operation is provided for convenience, and it is to request an operational status update from an event correlator.

For further information on these facilities, see "Tuning correlator performance" in *Deploying and Managing Apama Applications*.

The Client Software Development Kits

The client software development kits

Apama provides Client Software Development Kits (SDKs) for C, C++, .NET, Java and JavaBeans. These allow software written either in C, C++ or Java to interface with the event correlator. Apama also provides engine client libraries for .NET applications. The .NET client library documentation is located here: `install_dir\doc\dotNet\engine_client_dotnet5.2.chm`.

The SDKs for C, C++ and Java are located in the `lib` folder of the Apama installation. For C and C++ this consists of the libraries `libengine_client.so.5.2` (on Solaris or Linux), or `engine_client5.2.lib` (on Windows). The equivalent Java classes are provided within `engine_client5.2.jar` and are documented in `doc\javadoc`. The .NET engine client libraries, `engine_client5.2.dll` and `engine_client_dotnet5.2.dll` libraries are located in the Apama installation's `bin` directory.

In order to program against the C/C++ SDKs a developer must use the definitions from the `engine_client_c.h` header file for C or `engine_client_cpp.hpp` header file for C++ located in the `include` folder of the Apama installation.

C++ compilers vary extensively in their support for the ISO C++ standard and in how they support linking. For this reason Apama supports only specific C/C++ compilers and development environments. For a list of the supported C++ compilers, see Software AG's Knowledge Center in Empower at https://empower.softwareag.com.

On the other hand, C has been standardized for several years, and for this reason the C development kit should work with the majority of modern C/C++ compilers on all platforms. However, note that when using a C compiler and linker you still need to link against the standard C++ library since Apama's underlying libraries contain C++ code.

To configure the build for an Apama client:

- On Linux, copying and customizing an Apama makefile from a sample application is the easiest method.
- On Windows, you might find it easiest to copy an Apama sample project. If you prefer to use a project you already have, be sure to add `$(APAMA_HOME)\include` as an include directory. To do this in Visual Studio, select your project and then select Project Properties > C/C++ > General > Additional Include Directories.

  Also, link against `engine_client$(APAMA_LIBRARY_VERSION).lib`. To do this in Visual Studio, select your project and then select Project Properties > Linker > Input > Additional Dependencies and add:

```
engine_client$(APAMA_LIBRARY_VERSION).lib
```

Finally, select Project Properties > Linker > General > Additional Library Directories, and add `$(APAMA_HOME)\lib`.

Apama requires Java 6 (as a minimum version) and ships with Oracle JRE 7.0. The recommendation is that you use JRE or JDK 7.0 to develop, build, test, and deploy your applications. Use of any JRE other than the one that Apama ships with is discouraged.

The Client Software Development Kits

Chapter 2: The Client Software Development Kits for C++ and Java

This topic focuses on the SDKs for C++ and Java, and it starts by introducing the main classes and methods provided, and then provides the class and method definitions.

The following section "Using the SDKs – C++ and Java Examples" on page 38 then describes a simple example that illustrates how to use the library to create and build a client program that can drive and interact with the event correlator.

For higher level Java interfaces see the following topics:

- "The JavaBeans API" on page 54
- "The EventService API" on page 63
- "The ScenarioService API" on page 70

Note: In C++ Applications, any strings passed by the application to the event correlator need to be encoded as UTF-8 (or as pure 7-bit ASCII, which is a subset of UTF-8). If the application environment is something other than UTF-8, use the `convertToUTF8()` and `convertFromUTF8()` functions as described in "Data classes" on page 13.

The library classes

The development libraries contain several classes and some static methods/functions.

The Client Software Development Kits for C++ and Java

Main classes

The primary class contained in the C++ library is `com::apama::engine::EngineManagement`

In Java the equivalent is an interface called `com.apama.engine.EngineManagement`

In C++ a developer needs to get an `EngineManagement` object by calling the function `connectToEngine(const char* host, unsigned short port)`

This method takes as parameters a host name and a socket port number. Together these parameters indicate the network location of an event correlator. `connectToEngine` then returns an `EngineManagement` object. Similarly, in Java one must call the static method `connectToEngine(java.lang.String host, int port)` of the factory class `com.apama.engine.EngineManagementFactory`, which returns an object that implements `EngineManagement`.

In both C++ and Java this object then allows a developer to:

- Inject EPL code
- Delete EPL entities
- Send events into a correlator
- Get a correlator's current operational status
- Connect a receiver of events
- Verify that the correlator is still available
- Interrogate the correlator as to what monitor and event definitions it has
- Connect a correlator as a consumer of another correlator

In order to receive events from an event correlator, the client needs to create a class that, in C++, inherits from `com::apama::event::EventConsumer` or, in Java, implements the interface `com.apama.event.EventConsumer`

When this is connected to the correlator, a `com::apama::event::EventSupplier` object is created to act as the unique interface between the correlator and that particular `EventConsumer` instance. The Java equivalent is `com.apama.event.EventSupplier`. In C++ each `EventSupplier` object which is created for an `EventConsumer` must be individually disconnected either by calling `.disconnect()` on it or deleted via the `deleteEventSupplier` API method before deleting the corresponding `EventConsumer` and before disconnecting the `EngineManagement` object from which it was created

Events are emitted onto named *channels*. For an application to receive events from the correlator it must register itself as an event receiver (an `EventConsumer`) and *subscribe* to one or more channels. Then if events are emitted to those channels by the correlator's monitors they will be forwarded to it.

Channels effectively allow both *publish-subscribe* message delivery, through which one can also achieve *point-to-point*. Channels can be set up to represent topics. External applications can then subscribe to event messages of the relevant topics. Otherwise a channel can be set up purely to indicate a destination and have only one application connected to it.

Note: A consumer with multiple subscriptions to the *same* channel will receive only a single copy of each event emitted to the subscribed channel. A consumer that subscribes to a channel without specifying a name will receive events from all channels; this behavior is the same for a subscription with no channel specified.

The developer must inherit from (or implement) `EventConsumer` in order to provide an implementation for its `sendEvents` method. This method is then called by the `EventSupplier` representing the correlator whenever the latter emits events that match the consumer's channel filter.

In order to send events to the correlator, an application calls methods of the `com.apama.engine.EngineManagement` instance. The following methods are available:

- `sentEvents` — This method of the `com.apama.engine.EngineManagement` sends the events passed to it using the calling thread. It automatically rebatches the actual event sending for efficiency. Your client application should call the `flushEvents()` method of of the `com.apama.engine.EngineManagement` class before exiting to ensure all events have been sent.
- `sendEventsNoBatching()` — This method of the `com.apama.engine.EngineManagement` class sends events to the correlator without doing any batching.
- `flushEvents()` — This method of the `com.apama.engine.EngineManagement` class waits for all events sent by `sendEvents()` to be acknowledged by the correlator. Note that this can take some time even

if the correlator is responsive as acknowledgements are sent intermittently, so avoid calling this within any performance critical loops. `flushEvents()` is implicitly called before inject and delete requests are sent. This call is cheap if there are no outstanding events.

For C++ clients, if a client wishes to be notified when it is disconnected, it should supply a class that inherits/implements the `DisconnectableEventConsumer` class/interface. The `disconnect()` method will be called on the client's class when the connection to the correlator is lost, with a reason explaining why.

For C clients, if a client wishes to be notified when it is disconnected, it should supply a pointer to an `AP_DisconnectableEventConsumer` (which contains a pointer to an `AP_EventConsumer` and function pointer to a disconnect method) to the `connectDisconnectableEventConsumer` function. The function pointed to by the disconnect function pointer will be called with an explantation of why the consumer was disconnect if the client is disconnected from the correlator.

The library classes

Data classes

The following classes represent the types used to interact with the event correlator. EPL monitor and event type definitions (expressed as UTF-8 encoded strings of EPL code in C++) need to be created and manipulated through objects of the class `com::apama::engine::MonitorScript` in C++ and `com.apama.engine.MonitorScript` in Java. Both monitors and event types can be deleted explicitly from the correlator through the `deleteName` method of the `EngineManagement` object.

Event instances of a type that has already been defined with the correlator need to be created and encoded as `com::apama::event::Event` or `com.apama.event.Event` objects. EngineManagement itself implements an EventConsumer and defines a sendEvent method. This is used to inject Event instances into the correlator.

In C++ the developer is responsible for deleting both types of object (`MonitorScript` and `Event`) after they have been used.

If your application uses a locale that does not use UTF-8 encoding, any strings passed to the event correlator need to be converted to UTF-8. For conversion purposes use the following functions:

- `com::apama::engine::convertToUTF8()`
- `com::apama::engine::convertFromUTF8()`

These functions convert between UTF-8 and what they assume to be the local character set. If this assumption is incorrect, unpredictable results may occur.

The library classes

Event correlator interrogation and status

A developer can enquire as to what definitions are present within a correlator. This can be achieved by calling the `inspectEngine` method on `EngineManagement`. This returns a `com::apama::engine::EngineInfo` or `com.apama.engine.EngineInfo` object, from which one can obtain:

- The number of monitors

- The number of event types
- The number of container types
- Information about the monitors

  This provides the name of every monitor and the number of instances of each monitor.
- Information about the event types

  This returns the name of every defined event type and indicates how many event templates are in use for each type.
- Information about the container types

  This returns the name of every defined container type.

A developer can request the correlator's current operational status by calling the `getStatus` method on `EngineManagement`. This returns a `com::apama::engine::EngineStatus` or `com.apama.engine.EngineStatus` object, which contains several runtime operational parameters, including

- The time in milliseconds that the correlator has been running
- The number of monitors defined in the correlator
- The number of monitor processes or active monitor instances (if a monitor spawns it creates a new process)
- The number of active listeners
- The number of event types defined
- Across all contexts, the total number of routed events waiting on input queues
- Across all contexts and excluding routed events, the total number of events waiting on input queues
- Across all contexts, the total number of events received on input queues since the correlator started
- The number of events that have been routed since the correlator was started
- The number of event consumers connected to the correlator
- The number of events waiting on the output queue
- The number of events that have been discarded from the output queue since the correlator started

The library classes

Additional functionality

The C++ and Java development kits vary slightly in the way they support construction and destruction of objects. While the C++ SDK provides a set of static library methods that must be called to create and delete objects of the main classes, the SDK for Java either provides factory classes or else has no restrictions on directly constructing objects.

The following methods (parameters not specified here) are provided:

- `com::apama::engine::engineInit()` – C++

  This method must be called exactly once when the client program is started. It initializes the library's state. There is no such requirement in Java and therefore no equivalent.

- `com::apama::engine::connectToEngine()` – C++

  `connectToEngine()` in `com.apama.engine.EngieManagementFactory` – Java

  This method is called to establish a connection to a running correlator instance.

- `com::apama::engine::createMonitorScript()` – C++

  This method creates and returns a `MonitorScript` object.

- `com.apama.MonitorScript` class – Java

  In Java one can directly construct a `MonitorScript` object.

- `com::apama::engine::deleteMonitorScript()` – C++

  This method frees and deletes a `MonitorScript` object. Note that the developer is responsible for calling this method and freeing the memory used by a `MonitorScript` object.

- `com.apama.MonitorScript` class – Java

  In Java the `MonitorScript` object is garbage collected as normal when no longer referenced.

- `com::apama::event::deleteEventSupplier()` – C++

  `disconnect()` in `com.apama.event.EventSupplier` – Java

  This method frees and deletes the `EventSupplier` specified and breaks the connection between the consumer and the `EventSupplier`/correlator.

- `com::apama::engine::deleteStatus()` – C++

  This method frees and deletes an `EngineStatus` object. Note that the developer is responsible for calling this method and freeing the memory used by an `EngineStatus` object.

- `com.apama.engine.EngineStatus` class – Java

  In Java, an object that implements `EngineStatus` is automatically garbage collected.

- `com::apama::event::createEvent()` – C++

  This method creates and returns an `Event` object, which can then be injected into the correlator.

- `com.apama.event.Event` class – Java

  In Java, one can directly construct an `Event` object.

- `com::apama::event::deleteEvent()` – C++

  This method frees and deletes an `Event` object. Note that the developer is responsible for calling this method and freeing the memory used by an `Event` object.

- `com.apama.event.Event` class – Java

  In Java, an `Event` object is garbage collected as normal.

- `com::apama::engine::disconnectFromEngine()` – C++

This method disconnects the client program from the correlator and cleans up data structures and memory resources.

There is no Java equivalent as it is not required.

- `com::apama::engine::engineShutdown()` – C++

  This method cleans up and shuts down the client library and must be called exactly once, after the program has disconnected from the correlator.

  There is no Java equivalent requirement.

The library classes

Exceptions

Several of the class methods and the static library methods can throw exceptions if they fail or encounter exceptional circumstances. All of these are of the type `com::apama::EngineException` in C++, or `com.apama.EngineException` in Java. Both contain a text message indicating the nature of the problem encountered.

The library classes

Logging in C++

The C++ (and C) SDK can output extensive logging information. This information can be useful in diagnosing connectivity issues or problems that one may encounter when writing software that interfaces with the correlator.

The author of a C++ client need not bother with the standard logging unless they want to modify its operating parameters.

By default, the log level is set to `WARN`, where only significant warnings and errors are displayed in the log. The whole list of log levels is `OFF` (i.e. no logging at all), `CRIT`, `FATAL`, `ERROR`, `WARN`, `INFO`, `DEBUG` and `TRACE`. These levels are listed in order of decreasing importance, and conversely in the order of least likely occurrence. A very large volume of information is output at `DEBUG` level.

To change the logging, three functions are provided in the C++ SDK;

- `com::apama::setLogLevel()`

  Sets the level at which the client library will log information.

- `com::apama::setLogFile()`

  Sets the file to which the client library should log information.

- `com::apama::setLogFD()`

  Sets the file descriptor to which the client library should log information.

For information on the complete signatures for these functions, see "The C++ header file" on page 18.

The library classes

Logging in Java

The logging facilities in Java are significantly more powerful than in C++. Beginning with Apama version 2.1, the underlying Client SDK for Java makes use of Log4j, a publicly available logging library for Java. Previously it made use of Apama's `SimpleLogger` class. For convenience, Apama provides a wrapper class that abstracts the logging capabilities provided by either, and it is this interface that is used by the Client SDK for Java.

Note: Full documentation for Log4j and the Apache Logging Service project can be found at `http://logging.apache.org`.

These logging facilities are provided in `com.apama.util.Logger`, for which Javadoc is provided.

The author of a Java client need not bother with the standard logging unless they want to modify its operating parameters. By default the SDK classes will log at `WARN` level. The log level can be changed as described in the Javadoc for the `Logger` class.

The Javadoc also provides instructions on how to get a reference to the Logger object in your own code so that you can produce your own logging output.

The library classes

Thread-safety

The C++ SDK is thread-safe, in the sense that you can call API methods from any thread. In particular you can:

- Call `connectToEngine()` and `disconnectFromEngine()` from different threads
- Attach and detach event consumers on different threads

Background threads are created when you call `engineInit()` and destroyed when `engineShutdown()` is called. Events received from a correlator will be handled in one of these background threads, so you cannot assume that events will be delivered to you on any particular thread. In the C++ SDK, you must call `engineShutdown()` before calling `exit()` or returning from the main function; not doing so may result in a crash.

Note that the API functions are not synchronized, so if you have data that you need to protect from concurrent access, you will need to implement synchronization yourself.

The same applies for the SDK for Java.

The library classes

The complete definitions

This section presents the C++ header file in its entirety, as it serves as a complete reference to classes and methods available in the C++ SDK.

For the Java equivalent, see the comprehensive and accessible HTML reference available in the `doc\javadoc` folder of the Apama installation.

The Client Software Development Kits for C++ and Java

The C++ header file

Here is a listing of the integration library's header file. The file is called `engine_client_cpp.hpp` and is available in the `include` folder of the Apama installation.

```
/**
 * engine_client_cpp.hpp
 *
 * This is the header file for the Apama Event Manager C++ SDK
 *
 * $Copyright(c) 2013 Software AG, Darmstadt, Germany and/or its licensors$
 *
 * $Id$
 */

#ifndef ENGINE_CLIENT_CPP_HPP
#define ENGINE_CLIENT_CPP_HPP

#include <AP_Types.h>
#include <AP_Platform.h>
#include <iostream>
#include <sstream>
#include <exception>

#undef ERROR

namespace com {
namespace apama {

 /**
  * An EngineException is thrown by methods in this library if any
  * problems are  encountered.
  */
 class AP_ENGINE_CLIENT_API EngineException : public std::exception {

 public:
  // Constructors
  EngineException(const char* message);
  EngineException(const EngineException& ce);
  EngineException& operator= (const EngineException& ce);

  // Destructor
  virtual ~EngineException() throw();

  /**
   * Retrieve the message enclosed within the exception.
   *
   * @return C style string with details of the error that
   * caused the operation to fail.
   */
  virtual const char* what() const throw();

  /**
   * Retrieve the set of warnings associated with the exception
   *
   * @return A pointer to a NULL terminated list of warnings,
```

```
   * possibly NULL if there are none. This list is owned by
   * the exception object. It should not be deleted.
   */
 virtual const char* const* getWarnings() const;

protected:
 const char* const* m_warnings;

private:
 char* m_message;
    };

/** Available logging levels */
typedef enum {
 OFF_LEVEL,
 FORCE_LEVEL,
 CRIT_LEVEL,
 FATAL_LEVEL,
 ERROR_LEVEL,
 WARN_LEVEL,
 INFO_LEVEL,
 DEBUG_LEVEL,
       TRACE_LEVEL
} LogLevel;

/**
 * Sets the level at which the client library will log information.
 *
 * @param level The level to log at
 */
AP_ENGINE_CLIENT_API void setLogLevel(LogLevel level);

/**
 * Sets the file to which the client library should log information.
 *
 * @param filename The filename to which the library should log. Will log
 * to stderr if the filename cannot be opened.
 * @param truncate If non-zero the log file will be truncated when it is
 * opened. If zero then it will simply be appended to.
 * @param utf8 Interpret filename as UTF-8, rather than the local character
 * set.
 */
AP_ENGINE_CLIENT_API void setLogFile(const AP_char8* filename, bool truncate,
     bool utf8 = false);

/**
 * Sets the file descriptor to which the client library should log information
 *
 * @param fd The file descriptor to log to
 */
AP_ENGINE_CLIENT_API void setLogFD(int fd);

/**
 * Sets the mutex which the client library logger should use. For internal use only.
 *
 */
AP_ENGINE_CLIENT_API void setLogMutex(void *mutex);

/**
 * Re-opens the log file. Also called if SIGHUP is received.
 *
 */
AP_ENGINE_CLIENT_API void reOpenLog();

/** Convert a string in local encoding to UTF-8, as required by most of the Apama API */
AP_ENGINE_CLIENT_API AP_char8* convertToUTF8(const AP_char8* s);

/** Convert a string in UTF-8 to the local encoding, since most of the
    Apama API returns UTF-8 */
AP_ENGINE_CLIENT_API AP_char8* convertFromUTF8(const AP_char8* s);
```

```
namespace event {

 /**
  * An Event object represents an event instance.
  */
 class AP_ENGINE_CLIENT_API Event {

  friend AP_ENGINE_CLIENT_API Event* createEvent(const char* eventString);
  friend AP_ENGINE_CLIENT_API void deleteEvent(Event* ev);

 public:
  /**
   * Retrieve the event's type and its contents as a string.
   *
   * @return C style string with the event's textual representation.
   * Note that these strings should be encoded in UTF-8.
   */
  virtual const char* getText() const = 0;

  /**
   * Retrieve the event's channel.  This currently only has meaning for
   * events which have been sourced from the correlator, in which case
   * it is set to the name of the channel the event was emitted from, or
   * "" (the empty string) for the wildcard channel.
   *
   * @return The channel of the event, encoded in UTF-8.
   */
  virtual const char* getChannel() const = 0;

  /**
   * Retrieve the event's timestamp.  This timestamp currently only has
   * meaning for events which have been sourced from the correlator, in
   * which case it is set to the correlator time at which it was created,
   * or the value it was set to explicitly by code running within the
   * correlator.
   *
   * @return the timestamp of the event, in floating point seconds since
   * the Unix epoch.
   */
   virtual double getTime() const = 0;

  // Stream output operators
  inline friend std::ostream& operator << (std::ostream& stream, const Event& obj) {
   stream << obj.getText();
   return stream;
  }
  inline friend std::ostream& operator << (std::ostream& stream, const Event* obj) {
   stream << obj->getText();
   return stream;
  }

 private:
  Event(const Event&);
  Event& operator= (const Event&);

 protected:
  Event();
  virtual ~Event();
 };

 /**
  * An EventSupplier represents the resources created by the Engine
  * to service a connection to an external sink of events. It
  * filters the event output of the Engine by delivering only the
  * events emitted on a particular set of channels. An
  * EventSupplier passes events to an EventConsumer.
  * EventSupplier objects should be freed using the
  * com::apama::event::deleteEventSupplier function.
  */
```

```
class AP_ENGINE_CLIENT_API EventSupplier {

 friend AP_ENGINE_CLIENT_API void deleteEventSupplier(EventSupplier* evsup);

public:
 /**
  * Disconnect the EventSupplier from its consumer and
  * release its resources.
  *
  * @exception EngineException
  */
 virtual void disconnect() = 0;

protected:
 EventSupplier();
 virtual ~EventSupplier();

private:
 EventSupplier(const EventSupplier&);
 EventSupplier& operator= (const EventSupplier&);
};

/**
 * An EventConsumer can connect to the Engine through an
 * EventSupplier and register to receive events. In order to
 * receive events from the Engine, a developer must inherit from
 * this class and define its sendEvents method. This method is
 * called by the EventSupplier when events are emitted from the
 * Engine.
 */
class AP_ENGINE_CLIENT_API EventConsumer {

public:
 /**
  * This method must be defined in inherited classes to
  * enable receiving of events. This method is called by an
  * EventSupplier.
  *
  * Note that EventSuppliers are not reentrant so
  * calling the disconnect method on the calling
  * EventSupplier is not permitted from within the
  * sendEvents implementation.
  *
  * @param events An array of pointers to Event objects.
  */
 virtual void sendEvents(const Event* const* events) = 0;

protected:
 EventConsumer();
 virtual ~EventConsumer();

private:
 EventConsumer(const EventConsumer&);
 EventConsumer& operator= (const EventConsumer&);
};

class  AP_ENGINE_CLIENT_API DisconnectableEventConsumer : public EventConsumer {

public:
 /**
  * Used to inform the consumer that it is not going
  * to be sent any more events.
  *
  * @param A string giving the reason why the consumer
  * is being disconnected. May be NULL.
  */
 virtual void disconnect(const char* reason) = 0;
protected:
        DisconnectableEventConsumer();
 virtual ~DisconnectableEventConsumer();
```

```
 private:
  DisconnectableEventConsumer(const DisconnectableEventConsumer&);
  DisconnectableEventConsumer& operator= (const DisconnectableEventConsumer&);
 };

 /**
  * This function allows creation of an Event object.
  *
  * @param eventString C style string representing the event
  * instance in MonitorScript. Note that this string should
  * be encoded in UTF-8.
  * @return A reference to an Event object.
  */
 AP_ENGINE_CLIENT_API Event* createEvent(const char* eventString);

 /**
  * This function allows deletion of an Event object.
  *
  * @param ev A reference to an Event object.
  */
 AP_ENGINE_CLIENT_API void deleteEvent(Event* ev);

 /**
  * This function allows deletion of an EventSupplier object.
  * It also stops the associated EventConsumer from receiving
  * events.
  *
  * @param evsup A reference to an EventSupplier object.
  */
 AP_ENGINE_CLIENT_API void deleteEventSupplier(EventSupplier* evsup);

} // namespace event

namespace engine {

 /**
  * A MonitorScript object encapsulates a MonitorScript code
  * fragment, containing package, event and monitor definitions to
  * be injected into an Engine.
  */
 class AP_ENGINE_CLIENT_API MonitorScript {

  friend AP_ENGINE_CLIENT_API MonitorScript*
      createMonitorScript(const char* monitorString);
  friend AP_ENGINE_CLIENT_API void deleteMonitorScript(MonitorScript* mon);

 public:
  /**
   * Retrieve the text of a MonitorScript fragment as a string.
   *
   * @return C style string containing the text of the
   * MonitorScript fragment. Note that this string should
   * be encoded in UTF-8.
   */
  virtual const char* getText() const = 0;

  // Stream output operators
  inline friend std::ostream& operator << (std::ostream& stream,
           const MonitorScript& obj) {
   stream << obj.getText();
   return stream;
  }
  inline friend std::ostream& operator << (std::ostream& stream,
           const MonitorScript* obj) {
   stream << obj->getText();
   return stream;
  }

 private:
  MonitorScript(const MonitorScript&);
```

```
 MonitorScript& operator= (const MonitorScript&);

protected:
 MonitorScript();
 virtual ~MonitorScript();
};

/**
 * A class used for the iterating trough all status items
 */
class AP_ENGINE_CLIENT_API StatusIterator {

public:
 /**
  * Checks if there are more status items available
   * @return True if there are more status items
  * available, false otherwise.
  */
 virtual bool hasNext() const = 0;

 /**
  * Sets arguments with the next status item name and value
  * @return True if the name and the value were set False
  * if there were no more status items, in which case the
  * name and the value arguments are not set
  */
 virtual bool getNext(const char *&name, const char *&value) const = 0;

 virtual ~StatusIterator();
private:
 StatusIterator(const StatusIterator&);
 StatusIterator& operator= (const StatusIterator&);

protected:
 StatusIterator();
};

/**
 * EngineStatus represents the operational status of the Engine.
 */
class AP_ENGINE_CLIENT_API EngineStatus {
 friend AP_ENGINE_CLIENT_API void deleteStatus(EngineStatus* status);

public:
 /**
  * Get the time in ms that the Engine has been running
  * for.
  */
 virtual AP_uint64 getUptime() const = 0;

 /**
  * Get the number of monitors defined in the Engine.
  */
 virtual AP_uint32 getNumMonitors() const = 0;

 /**
  * Get the number of monitor processes or active
  * sub-monitors. If a monitor spawns it creates a new
  * process.
  */
 virtual AP_uint32 getNumProcesses() const = 0;

 /**
  * Get the number of java applications defined in the Engine.
  */
 virtual AP_uint32 getNumJavaApplications() const = 0;

 /**
  * Get the number of active listeners.
  */
```

```
virtual AP_uint32 getNumListeners() const = 0;

/**
 * Get the number of active listeners.
 */
virtual AP_uint32 getNumSubListeners() const = 0;

/**
 * Get the number of event types defined.
 */
virtual AP_uint32 getNumEventTypes() const = 0;

/**
 * Get the number of events waiting on the internal input
 * queue.
 */
virtual AP_uint32 getNumQueuedFastTrack() const = 0;

/**
 * Get the number of events waiting on the input queue.
 */
virtual AP_uint32 getNumQueuedInput() const = 0;

/**
 * Get the number of events received since the Engine
 * started (including those discarded because they
 * were invalid)..
 */
virtual AP_uint64 getNumReceived() const = 0;

/**
 * Get the number of events taken off the input queue
 * and processed since the Engine started.
 */
virtual AP_uint64 getNumProcessed() const = 0;

/**
 * Get the number of events received on the internal input
 * queue since the Engine started.
 */
virtual AP_uint64 getNumFastTracked() const = 0;

/**
 * Get the number of event consumers connected to the
 * engine.
 */
virtual AP_uint32 getNumConsumers() const = 0;

/**
 * Get the number of events waiting on the output queue.
 */
virtual AP_uint32 getNumOutEventsQueued() const = 0;

/**
 * Gets the number of output events which have been
 * put onto the output queue. This corresponds to the
 * number of MonitorScript emit commands executed.
 */
virtual AP_uint64 getNumOutEventsCreated() const = 0;

/**
 * This is the number of output events sent out by the
 * correlator process. This differs from getNumOutEventsCreated
 * since events can be of interest to a varying number of
 * consumers.
 */
virtual AP_uint64 getNumOutEventsSent() const = 0;

/**
 * Get the number of active contexts.
```

```
 */
virtual AP_uint32 getNumContexts() const = 0;

/**
 * Returns an instance of the StatusIterator which
 * allows to iterate over all status items
 * @return pointer to the StatusIterator. The caller
 * is responsible for the deletion of the returned
 * instance
 */
virtual StatusIterator *getAllValues() const = 0;

/** Stream output operator */
inline friend std::ostream& operator << (std::ostream& stream, const EngineStatus& obj) {
 std::ostringstream ost;
 ost
 << "Uptime(ms):                  " << obj.getUptime() << std::endl
 << "Number of contexts:          " << obj.getNumContexts() << std::endl
 << "Number of monitors:          " << obj.getNumMonitors() << std::endl
 << "Number of sub-monitors:      " << obj.getNumProcesses() << std::endl
 << "Number of java applications: " << obj.getNumJavaApplications() << std::endl
 << "Number of listeners:         " << obj.getNumListeners() << std::endl
 << "Number of sub-listeners:     " << obj.getNumSubListeners() << std::endl
 << "Number of event types:       " << obj.getNumEventTypes() << std::endl
 << "Events on input queue:       " << obj.getNumQueuedInput() << std::endl
 << "Events received:             " << obj.getNumReceived() << std::endl
 << "Events processed:            " << obj.getNumProcessed() << std::endl
 << "Events on internal queue:    " << obj.getNumQueuedFastTrack() << std::endl
 << "Events routed internally:    " << obj.getNumFastTracked() << std::endl
 << "Number of consumers:         " << obj.getNumConsumers() << std::endl
 << "Events on output queue:      " << obj.getNumOutEventsQueued() << std::endl
 << "Output events created:       " << obj.getNumOutEventsCreated() << std::endl
 << "Output events sent:          " << obj.getNumOutEventsSent() << std::endl;
 stream << ost.str();
 return stream;
}

/** Stream output operator */
inline friend std::ostream& operator << (std::ostream& stream, const EngineStatus* obj) {
 std::ostringstream ost;
 ost
 << "Uptime(ms):                  " << obj->getUptime() << std::endl
 << "Number of contexts:          " << obj->getNumContexts() << std::endl
 << "Number of monitors:          " << obj->getNumMonitors() << std::endl
 << "Number of sub-monitors:      " << obj->getNumProcesses() << std::endl
 << "Number of java applications: " << obj->getNumJavaApplications() << std::endl
 << "Number of listeners:         " << obj->getNumListeners() << std::endl
 << "Number of sub-listeners:     " << obj->getNumSubListeners() << std::endl
 << "Number of event types:       " << obj->getNumEventTypes() << std::endl
 << "Events on input queue:       " << obj->getNumQueuedInput() << std::endl
 << "Events received:             " << obj->getNumReceived() << std::endl
 << "Events processed:            " << obj->getNumProcessed() << std::endl
 << "Events on internal queue:    " << obj->getNumQueuedFastTrack() << std::endl
 << "Events routed internally:    " << obj->getNumFastTracked() << std::endl
 << "Number of consumers:         " << obj->getNumConsumers() << std::endl
 << "Events on output queue:      " << obj->getNumOutEventsQueued() << std::endl
 << "Output events created:       " << obj->getNumOutEventsCreated() << std::endl
 << "Output events sent:          " << obj->getNumOutEventsSent() << std::endl;
 stream << ost.str();
 return stream;
}

protected:
 EngineStatus();
 virtual ~EngineStatus();

private:
 EngineStatus(const EngineStatus& old);
 EngineStatus& operator= (const EngineStatus& other);
};
```

```
/**
 * Base class for a named object (i.e. event type, container type
 * or monitor) returned by an engine inspection. Returned by methods
 * of the EngineInfo class. Strings returned by methods of this class
 * are valid until the EngineInfo class is deleted.
 */
class AP_ENGINE_CLIENT_API NameInfo {

public:
 /**
  * Name, excluding package, for example "MyEvent".
  */
 virtual const char* getName() const = 0;

 /**
  * Package name, for example "com.apamax", or an
  * empty string if in the  default package.
  */
 virtual const char* getPackage() const = 0;

 /**
  * Fully qualified name, for example "com.apamax.MyEvent"
  */
 virtual const char* getFullyQualifiedName() const = 0;

private:
 NameInfo(const NameInfo&);
 NameInfo& operator= (const NameInfo&);

protected:
 NameInfo();
 virtual ~NameInfo();
};

/**
 * Information about a monitor returned by an engine inspection.
 * Returned by the getMonitors method of the EngineInfo class.
 */
class AP_ENGINE_CLIENT_API NamedMonitorInfo : public NameInfo {

public:
 /**
  * Gets the number of sub-monitors belonging to this monitor.
  */
 virtual unsigned int getNumSubMonitors() const = 0;

private:
 NamedMonitorInfo(const NamedMonitorInfo&);
 NamedMonitorInfo& operator= (const NamedMonitorInfo&);

protected:
 NamedMonitorInfo();
 virtual ~NamedMonitorInfo();
};

/**
 * Information about a java application returned by an engine inspection.
 * Returned by the getJavaApplications method of the EngineInfo class.
 */
class AP_ENGINE_CLIENT_API NamedJavaApplicationInfo : public NameInfo {

public:
 /**
  * Gets the number of listeners belonging to this application.
  */
 virtual unsigned int getNumListeners() const = 0;

private:
 NamedJavaApplicationInfo(const NamedJavaApplicationInfo&);
```

```
 NamedJavaApplicationInfo& operator= (const NamedJavaApplicationInfo&);

protected:
 NamedJavaApplicationInfo();
 virtual ~NamedJavaApplicationInfo();
};

/**
 * Information about a context returned by an engine inspection.
 * Returned by the getContexts method of the EngineInfo class.
 */
class AP_ENGINE_CLIENT_API NamedContextInfo : public NameInfo {

public:
 /**
  * Gets the number of sub-monitors belonging to this context.
  */
 virtual unsigned int getNumSubMonitors() const = 0;

 /**
  * Gets the queue size of this context.
  */
 virtual unsigned int getQueueSize() const = 0;

private:
 NamedContextInfo(const NamedContextInfo&);
 NamedContextInfo& operator= (const NamedContextInfo&);

protected:
 NamedContextInfo();
 virtual ~NamedContextInfo();
};

/**
 * Information about a event type returned by an engine inspection.
 * Returned by the getEventTypes method of the EngineInfo class.
 */
class AP_ENGINE_CLIENT_API NamedEventTypeInfo : public NameInfo {

public:
 /**
  * Gets the number of event templates for this event type.
  */
 virtual unsigned int getNumEventTemplates() const = 0;

private:
 NamedEventTypeInfo(const NamedEventTypeInfo&);
 NamedEventTypeInfo& operator= (const NamedEventTypeInfo&);

protected:
 NamedEventTypeInfo();
 virtual ~NamedEventTypeInfo();
};

/**
 * Information about a timer type returned by an engine inspection.
 * Returned by the getTimers method of the EngineInfo class.
 */
class AP_ENGINE_CLIENT_API NamedTimerInfo : public NameInfo {

public:
 /**
  * Gets the number of timers for this timer type.
  */
 virtual unsigned int getNumTimers() const = 0;

private:
 NamedTimerInfo(const NamedTimerInfo&);
 NamedTimerInfo& operator= (const NamedTimerInfo&);
```

```
protected:
 NamedTimerInfo();
 virtual ~NamedTimerInfo();
};

/**
 * Information about an aggregate function returned by an engine
 * inspection.
 * Returned by the getAggregates method of the EngineInfo class.
 */
class AP_ENGINE_CLIENT_API NamedAggregateInfo : public NameInfo {

private:
 NamedAggregateInfo(const NamedAggregateInfo&);
 NamedAggregateInfo& operator= (const NamedAggregateInfo&);

protected:
 NamedAggregateInfo();
 virtual ~NamedAggregateInfo();
};

/**
 * Information about the monitors and types currently in an engine.
 * Instances of this class are returned by the inspectEngine
 * method of the EngineManagement interface and can be deleted with the
 * deleteInfo function.
 *
 * When an EngineInfo class is deleted, everything returned by the
 * getMonitors, getEventTypes, etc methods is deleted.
 * This includes the arrays themselves, the classes pointed to by
 * the arrays and any strings returned by those classes. This means
 * that deleteInfo is the only cleanup method that needs to be called
 * after an engine inspection.
 */
class AP_ENGINE_CLIENT_API EngineInfo {

 friend AP_ENGINE_CLIENT_API void deleteInfo(EngineInfo* info);

public:
 /**
  * Gets the number of monitors in the engine.
  */
 virtual unsigned int getNumMonitors() const = 0;

 /**
  * Gets the number of Java applications in the engine.
  */
 virtual unsigned int getNumJavaApplications() const = 0;

 /**
  * Gets the number of event types in the engine.
  */
 virtual unsigned int getNumEventTypes() const = 0;

 /**
  * Gets the number of timers in the engine.
  */
 virtual unsigned int getNumTimers() const = 0;

 /**
  * Gets the number of aggregate functions in the engine.
  */
 virtual unsigned int getNumAggregates() const = 0;

 /**
  * Returns information about the monitors in the engine,
  * in the form of a NULL terminated array of pointers to
  * MonitorInfo objects. The size of the array can be found
  * by calling getNumMonitors (or looking for the
  * NULL terminator).
```

```
     */
    virtual NamedMonitorInfo** getMonitors() const = 0;

    /**
     * Returns information about the Java applications in the
     * engine, in the form of a NULL terminated array of pointers
     * to JavaApplicationInfo objects. The size of the array can
     * be found by calling getNumJavaApplications (or looking for
     * the NULL terminator).
     */
    virtual NamedJavaApplicationInfo** getJavaApplications() const = 0;

    /**
     * Returns information about the event types in the engine,
     * in the form of a NULL terminated array of pointers to
     * NamedEventTypeInfo objects. The size of the array can be
     * found by calling getNumEventTypes (or looking for the
     * NULL terminator).
     */
    virtual NamedEventTypeInfo** getEventTypes() const = 0;

    /**
     * Returns information about the timers in the engine,
     * in the form of a NULL terminated array of pointers to
     * NamedTimerInfo objects. The size of the array can be
     * found by calling getNumTimers (or looking for the
     * NULL terminator).
     */
    virtual NamedTimerInfo** getTimers() const = 0;

    /**
     * Returns information about the aggregate functions in the
     * engine, in the form of a NULL terminated array of pointers
     * to NamedAggregateInfo objects. The size of the array can be
     * found by calling getNumAggregates (or looking for the
     * NULL terminator).
     */
    virtual NamedAggregateInfo** getAggregates() const = 0;

    /**
     * Gets the number of contexts in the engine.
     */
    virtual unsigned int getNumContexts() const = 0;

    /**
     * Returns information about the contexts in the
     * engine, in the form of a NULL terminated array of pointers
     * to ContextInfo objects. The size of the array can
     * be found by calling getNumContexts (or looking for
     * the NULL terminator).
     */
    virtual NamedContextInfo** getContexts() const = 0;

  private:
    EngineInfo(const EngineInfo&);
    EngineInfo& operator= (const EngineInfo&);

  protected:
    EngineInfo();
    virtual ~EngineInfo();
  };

  /**
   * The Engine Management class acts as the interface to the Engine,
   * and allows operations to be carried out on it. It is not
   * possible to construct an object of this class. The static
   * methods supplied below in the engine namespace must be used
   * instead.
   *
   * Because an EngineManagement object is also an EventConsumer,
```

```
   * it can have events sent to it (and to the Engine) through its
   * sendEvents() method.
   */
  class AP_ENGINE_CLIENT_API EngineManagement : public com::apama::event::EventConsumer {

public:
    enum ConnectMode {
       CONNECT_LEGACY,
       CONNECT_PARALLEL
    };

  public:
   /**
    * Inject MonitorScript text into the Engine.
    *
    * @param script MonitorScript text to be injected.
    * @exception EngineException
    */
   virtual void injectMonitorScript(MonitorScript& script) = 0;

   /**
    * Inject MonitorScript text into the Engine,
    * returning any warnings produced by the
    * MonitorScript compiler
    *
    * @param script MonitorScript text to be injected.
    * @return A pointer to a NULL terminated array of
    * warnings which must be deleted with the
    * deleteWarnings() function, or NULL if there are
    * none
    * @exception EngineException, which may contain
    * warnings as well as an error message
    */
   virtual const char* const* injectMonitorScriptWithWarnings(MonitorScript& script) = 0;

   /**
    * Inject MonitorScript text into the Engine,
    * supplying the filename it was injected from and
    * returning any warnings produced by the
    * MonitorScript compiler
    *
    * @param script MonitorScript text to be injected.
    * @param filename The filename the MonitorScript text
    * was injected from
    * @return A pointer to a NULL terminated array of
    * warnings which must be deleted with the
    * deleteWarnings() function, or NULL if there are
    * none
    * @exception EngineException, which may contain
    * warnings as well as an error message
    */
   virtual const char* const* injectMonitorScriptWithWarningsFilename(MonitorScript& script,
        const char *filename) = 0;

   /**
    * Delete a named object from the Engine.
    *
    * @param name The name of the object to be deleted.
    * @exception EngineException
    */
   virtual void deleteName(const char * name) = 0;

   /**
    * Force deletion of a named object from the Engine.
    *
    * @param name The name of the object to be deleted.
    * @exception EngineException
    */
   virtual void forceDeleteName(const char * name) = 0;
```

```
/**
 * Kill a named object in the Engine.
 *
 * @param name The name of the object to be killed.
 * @exception EngineException
 */
virtual void killName(const char * name) = 0;

/**
 * Deletes everything from the engine.
 */
virtual void deleteAll() = 0;

/**
 * Injects a Java application (a jar) into the engine.
 *
 * @param jarbytes A pointer to the array of bytes
 * containing the jar
 * @param size The size of the jar
 * @exception EngineException
 */
virtual void injectJava(const AP_uint8* jarbytes, AP_uint32 size) = 0;

/**
 * Injects a Java application (a jar) into the engine.
 *
 * @param jarbytes A pointer to the array of bytes
 * containing the jar
 * @param size The size of the jar
 * @return A pointer to a NULL terminated array of
 * warnings which must be deleted with the
 * deleteWarnings() function, or NULL if there are
 * none
 * @exception EngineException, which may contain
 * warnings as well as an error message
 */
virtual const char* const* injectJavaWithWarnings(const AP_uint8* jarbytes,
    AP_uint32 size) = 0;

/**
 * Injects a Java application (a jar) into the engine.
 *
 * @param jarbytes A pointer to the array of bytes
 * containing the jar
 * @param size The size of the jar
 * @param filename The name of the jar file
 * @return A pointer to a NULL terminated array of
 * warnings which must be deleted with the
 * deleteWarnings() function, or NULL if there are
 * none
 * @exception EngineException, which may contain
 * warnings as well as an error message
 */
virtual const char* const* injectJavaWithWarningsFilename(const AP_uint8* jarbytes,
    AP_uint32 size, const char *filename) = 0;

/**
 * Injects a CDP (Correlator Deployment Package file) into the engine.
 *
 * @param cdpbytes A pointer to the array of bytes
 * containing the CDP file
 * @param size The size of the CDP
 * @exception EngineException
 */
virtual void injectCDP(const AP_uint8* cdpbytes, AP_uint32 size,
    const char *filename=NULL) = 0;
/**
 * Injects a CDP (Correlator Deployment Package file) into the engine.
 *
 * @param cdpbytes A pointer to the array of bytes
```

```
     * containing the CDP file
     * @param size The size of the CDP file
     * @return A pointer to a NULL terminated array of
     * warnings which must be deleted with the
     * deleteWarnings() function, or NULL if there are
     * none
     * @exception EngineException, which may contain
     * warnings as well as an error message
     */
    virtual const char* const* injectCDPWithWarnings(const AP_uint8* cdpbytes,
        AP_uint32 size) = 0;
    /**
     * Injects a CDP (Correlator Deployment Package file) into the engine.
     *
     * @param cdpbytes A pointer to the array of bytes
     * containing the CDP file
     * @param size The size of the CDP file
     * @param filename The name of the CDP file
     * @return A pointer to a NULL terminated array of
     * warnings which must be deleted with the
     * deleteWarnings() function, or NULL if there are
     * none
     * @exception EngineException, which may contain
     * warnings as well as an error message
     */
    virtual const char* const* injectCDPWithWarningsFilename(const AP_uint8* cdpbytes,
        AP_uint32 size, const char *filename) = 0;
    /**
     * Get the Engine's current operational status. Use
     * deleteStatus to delete the returned object when
     * it is no longer needed.
     *
     * @exception EngineException
     */
    virtual EngineStatus* getStatus() = 0;

    /**
     * Connect an event receiver to the Engine.
     *
     * @param consumer The EventConsumer to connect to the
     * Engine.
     * @param channels An array of names representing the
     * channels to subscribe to. This is a null-terminated array
     * of pointers to zero-terminated char arrays. If it is null or
     * empty subscribe to all channels. Note that these channel
     * names should be encoded in UTF-8.
     * @disconnectSlow tell the correlator it can disconnect
     * this receiver if it is slow. Only the first consumer's
     * disconnectSlow value is used; subsequent consumers added
     * to this EngineManagement object share the connection and
     * thus the disconnect behaviour. Defaults to false.
     * @return A reference to an EventSupplier resource, which the caller
     * is responsible for freeing, using com::apama::event::deleteEventSupplier().
     * Each EventSupplier resource must be explicitly disconnected or deleted
     * before disconnecting this EngineManagement object.
     * @exception EngineException
     */
    virtual com::apama::event::EventSupplier* connectEventConsumer(
        com::apama::event::EventConsumer* consumer,
        const char* const* channels, bool disconnectSlow = false) = 0;

    /**
     * Connect this Engine as an event receiver to another Engine.
     *
     * @param target The Engine to connect to
     * @param channels An array of names representing the
     * channels to subscribe to. This is a null-terminated array
     * of pointers to zero-terminated char arrays. If this is null or empty,
     * subscribe to all channels. Note that these channel names
     * should be encoded in UTF-8.
```

```
     * @param disconnectSlow disconnect if slow. Only the first consumer's
     * disconnectSlow value is used; subsequent consumers added to this
     * EngineManagement object share the connection and thus the disconnect
     * behavior.
     * @param mode the connection mode to use,
     * defaults to legacy (single connection, all
     * events delivered to the default
     * channel). Set to CONNECT_PARALLEL for
     * connection per channel and channel values passed through.
     * @return true if successful
     * @exception EngineException
     */
    virtual bool attachAsEventConsumerTo(
        EngineManagement* target, const char* const* channels,
        bool disconnectSlow = false, ConnectMode mode = CONNECT_LEGACY) = 0;

    /**
     * Connect this Engine as an event receiver to another Engine.
     *
     * @param host The hostname of the Engine to connect to
     * @param port The port of the Engine to connect to
     * @param channels An array of names representing the
     * channels to subscribe to. This is a null-terminated array
     * of pointers to zero-terminated char arrays. If this is null or empty,
     * subscribe to all channels. Note that these channel names
     * should be encoded in UTF-8.
     * @param disconnectSlow disconnect if slow. Only the first consumer's
     * disconnectSlow value is used; subsequent consumers added to this
     * EngineManagement object share the connection and thus the disconnect
     * behaviour.
     * @param mode the connection mode to use,
     * defaults to legacy (single connection, all
     * events delivered to the default
     * channel). Set to CONNECT_PARALLEL for
     * connection per channel and channel values passed through.
     * @return true if succesful
     * @exception EngineException
     */
    virtual bool attachAsEventConsumerTo(
        const char* host, int port, const char* const* channels,
        bool disconnectSlow = false, ConnectMode mode = CONNECT_LEGACY) = 0;

    /**
     * Unsubscribe as an event receiver from another engine.
     *
     * @param target The Engine to unsubscribe from.
     * @param channels An array of names representing the
     * channels to unsubscribe from. This is a null-terminated array
     * of pointers to zero-terminated char arrays. If this is null or empty
     * unsubscribe from all channels.
     * @param mode the connection mode to use,
     * defaults to legacy (single connection, all
     * events delivered to the default
     * channel). Set to CONNECT_PARALLEL for
     * connection per channel and channel values passed through.
     * @exception EngineException
     */
    virtual void detachAsEventConsumerFrom(
        EngineManagement* target, const char* const* channels,
        ConnectMode mode = CONNECT_LEGACY) = 0;

    /**
     * Unsubscribe as an event receiver from another engine.
     *
     * @param host The host of the Engine to unsubscribe from.
     * @param port The port of the Engine to unsubscribe from.
     * @param channels An array of names representing the
     * channels to unsubscribe from. This is a null-terminated array
     * of pointers to zero-terminated char arrays. If this is null or empty
     * unsubscribe from all channels.
```

```
     * @param mode the connection mode to use,
     * defaults to legacy (single connection, all
     * events delivered to the default
     * channel). Set to CONNECT_PARALLEL for
     * connection per channel and channel values passed through.
     * @exception EngineException
     */
    virtual void detachAsEventConsumerFrom(
        const char* host, int port, const char* const* channels,
        ConnectMode mode = CONNECT_LEGACY) = 0;

    /**
     * Inject events into the Engine (inherited from
     * EventConsumer), automatically batching messages
     * on a separate thread. May return before events are
     * sent - call flushEvents before exiting.
     *
     * @param events An array of pointers to Event objects
     * containing the events to inject into the Engine.
     * @exception EngineException
     */
    virtual void sendEvents(const com::apama::event::Event* const* events) = 0;

    /**
     * Inject events into the Engine
     *
     * @param events An array of pointers to Event objects
     * containing the events to inject into the Engine.
     * @exception EngineException
     */
    virtual void sendEventsNoBatching(const com::apama::event::Event* const* events) = 0;

    /**
     * Send any outstanding events sent via sendEvents.
     */
    virtual void flushEvents() = 0;

    /**
     * Returns information about the monitors, event types and
     * container types which exist in the engine. Use deleteInfo
     * to delete the returned object when it is no longer needed.
     *
     * @return Information about the engine.
     * @exception EngineException
     */
    virtual EngineInfo* inspectEngine() = 0;

    /**
     * This method is used to check that the Engine is
     * still alive, potentially reconnecting if needed.
     * If the Engine is alive it returns
     * normally. If there is a problem then an
     * EngineException is thrown.
     *
     * @exception EngineException
     */
    virtual void ping() = 0;

    /**
     * This method is used to check that this object is
     * still connected to the Engine. It will never
     * reconnect. Returns true if connected.
     */
    virtual bool isConnected() = 0;

  protected:
    EngineManagement();
    virtual ~EngineManagement();

  private:
```

```
 EngineManagement(const EngineManagement&);
 EngineManagement& operator= (const EngineManagement&);
};

/**
 * This function must be called once per process first before
 * any other Engine operations are carried out.
 *
  * @exception EngineException
 */
AP_ENGINE_CLIENT_API void engineInit(const char* processName = "C++ Client");

/**
 * This function (or engineInit) must be called once per process first before
 * any other Engine operations are carried out.
 *
  * @exception EngineException
 */
AP_ENGINE_CLIENT_API void engineInitMessaging(const char *processName,
     bool initMessaging=true);

/**
 * This function must be called once per process before the
 * application closes down. It ensures that communications are
 * shutdown properly and state cleaned up.
 */
AP_ENGINE_CLIENT_API void engineShutdown();

/**
 * This function attempts to establish a connection to an
 * Engine. It returns an EngineManagement object that can be used
 * to access the Engine.
 *
 * @param host The machine name where an Engine can be located.
 * @param port The port that the Engine is listening on for
 * transport negotiations with the client.
 * @return An EngineManagement object if an Engine is located and
 * a connection established.
  * @exception EngineException
 */
AP_ENGINE_CLIENT_API EngineManagement* connectToEngine(const char* host,
     unsigned short port);

/**
 * Attempt to establish a receive-only connection to an Engine listening
 * on the named host and port.  If successful, the returned
 * EngineManagement can be used for the connectEventConsumer(),
 * getStatus(), inspectEngine() and ping() operations.
 * All other operations will fail.
 *
 * @param host The hostname of the machine the Engine is running on.
 * @param port The port that the Engine is listening on.
 * @return An EngineManagement object operating in receive-only mode.
 * @exception EngineException If the connection cannot be established.
 */
AP_ENGINE_CLIENT_API EngineManagement* connectToEngineReceiveOnly(const char* host,
     unsigned short port);

/**
 * Attempt to establish a monitor-only connection to an Engine listening
 * on the named host and port.  If successful, the returned
 * EngineManagement can be used for the getStatus(),
 * inspectEngine() and ping() operations.  All other operations will fail.
 *
 * @param host The hostname of the machine the Engine is running on.
 * @param port The port that the Engine is listening on.
 * @return An EngineManagement object operating in monitor-only mode.
 * @exception EngineException If the connection cannot be established.
 */
AP_ENGINE_CLIENT_API EngineManagement* connectToEngineMonitorOnly(const char* host,
```

```
      unsigned short port);

  /**
   * This function allows disconnection from an Engine.
   *
   * @param corr The Engine to disconnect from.
   */
  AP_ENGINE_CLIENT_API void disconnectFromEngine(EngineManagement* corr);

  /**
   * This function allows creation of MonitorScript objects.
   *
   * @param monitorString MonitorScript monitor/event definitions
   * @return A MonitorScript object.
   */
  AP_ENGINE_CLIENT_API MonitorScript* createMonitorScript(const char* monitorString);

  /**
   * This function allows deletion of MonitorScript objects.
   *
   * @param mon The MonitorScript object to delete.
   */
  AP_ENGINE_CLIENT_API void deleteMonitorScript(MonitorScript* mon);

  /**
   * This function allows deletion of an EngineStatus object.
   *
   * @param status The EngineStatus object to delete.
   */
  AP_ENGINE_CLIENT_API void deleteStatus(EngineStatus* status);

  /**
   * This function allows deletion of an EngineInfo object.
   * All objects returned by any calls to the methods of the
   * EngineInfo object are also deleted, so after calling
   * inspectEngine, deleteInfo is the only method which
   * needs to be called to clean up.
   *
   * @param info The EngineInfo object to delete.
   */
  AP_ENGINE_CLIENT_API void deleteInfo(EngineInfo* info);

  /**
   * This function allows deletion of the lists of warnings
   * returned by injectMonitorScriptWithWarnings(), injectCDPWithWarnings(),
   * and injectJavaWithWarnings()
   *
   * @param warnings A list of warnings to be deleted
   */
  AP_ENGINE_CLIENT_API void deleteWarnings(const char* const* warnings);

  /**
   * Set custom parameters for this instance of the client library.
   * Must be called before engineInit. The parameters are passed
   * as a set of name, value pairs formatted as
   * <name1>=<value1>;<name2=value2>;
   * (i.e. equals sign between name and value, and semicolon separating
   * the name value pairs).
   *
   * The following parameters are defined so far.
   *
   * ConfigFile - A filename from which to read a configuration file
   * LogicalID - The logical id with which this component should be
   * identified (a 64-bit integer)
   */
  AP_ENGINE_CLIENT_API void setEngineParams(const char* params);

 } // namespace engine
} // namespace apama
```

```
} // namespace com

#endif // ENGINE_CLIENT_CPP_HPP
```

The complete definitions

Chapter 3: Using the SDKs – C++ and Java Examples

This topic presents a simple example that illustrates the correct usage of the classes and methods of the development kits.

The example makes the event correlator act as an 'echo' server. Events are sent to it by some client code, and the event correlator sends the same events back. In setting this up, the code illustrates how to carry out the basic operations outlined in the earlier chapters, and demonstrates how to use the library's classes and methods.

The C++ and Java examples are basically identical.

A simple echo server in C++

Step 1 is to initialize the client-side library. This must be done only once during the lifetime of the client-side code.

```
com::apama::engine::engineInit();
```

Next, one must connect to a remote correlator. If the method fails an `EngineException` would be thrown.

```
EngineManagement* engine;
engine = com::apama::engine::connectToEngine(argv[1], atoi(argv[2]))
```

If the client code is to receive any events back from the event correlator it must register itself as an event receiver with the event correlator. In order to do this, the developer must create an object that inherits from `EventConsumer`, and implement its `sendEvents` method,

```
void receive_consumer::sendEvents(const Event* const * events) {
  for (const Event* const * event=events; *event; event++) {
    cout << *event << endl;
  }
}
```

In this example, the developer-defined `receive_consumer` constructor is establishing the connection with the event correlator. The `EventSupplier` returned is a handle to private resources allocated by the event correlator to service this consumer. These resources handle event filtering for this connection. These must be freed by the developer, and in this example they are being freed by calling `deleteEventSupplier()` in the destructor for this class.

```
receive_consumer::receive_consumer(EngineManagement* engine)
  : EventConsumer(), engine(engine), supplier(NULL)
{
  supplier = engine->connectEventConsumer(this, NULL);
}
```

Once initialization is complete the example starts interacting with the event correlator. First it creates a `MonitorScript` object and defines an EPL monitor and associated event type within it,

```
MonitorScript* script = com::apama::engine::createMonitorScript(
```

```
  "event TestEvent {" +
  "string text;" +
  "}" +
  "monitor Echo {" +
  "" +
  "TestEvent test;" +
  "" +
  "action onload {" +
  "on all TestEvent(*):test {" +
  "emit TestEvent(test.text);" +
  "}" +
  "}" +
  "}");
```

For a description of monitors and EPL syntax, please see "Getting Started with Apama EPL" in *Writing Apama EPL Applications*.

Then the EPL code is injected into the event correlator,

```
engine->injectMonitorScript(*script);
```

Finally, the now utilized `MonitorScript` object must be deleted. It is important to point out that the developer is responsible for calling this operation in order to free resources.

```
com::apama::engine::deleteMonitorScript(script);
```

The monitor is now active within the event correlator. To verify this, the example requests operational status from the event correlator. It then prints this out; after which it de-allocates the resources used by the `EngineStatus` object by calling `deleteStatus`.

```
EngineStatus* status = engine->getStatus();
cout << status << endl;
com::apama::engine::deleteStatus(status);
```

The next step is to send some events to the event correlator. The events here have been chosen to trigger the now active listener of the monitor injected earlier. Although events can only be represented as strings, nevertheless, the string format must still be equivalent to the `TestEvent` event type definition as contained within the `Echo` monitor injected into the event correlator earlier. Otherwise, the event instances will be rejected by the event correlator. In this example, the event type `TestEvent` only contains a single parameter of type `string`.

```
Event* events[3];
events[0] = com::apama::event::createEvent(
    "TestEvent(\"Hello, World\")");
events[1] = com::apama::event::createEvent(
    "TestEvent(\"Welcome to the Event Correlator\")");
events[2] = NULL;
engine->sendEvents(events);
com::apama::event::deleteEvent(events[0]);
com::apama::event::deleteEvent(events[1]);
```

Note that since `sendEvents` takes an array of `Event` references, the last element needs to be `NULL`. As before, the developer is responsible for deleting the `Event` objects after they have been passed to the event correlator.

If one wanted to pass in more complex events the procedure is identical. Consider the following event type definition in EPL,

```
event SharePrice {
    integer price;
    string companyName;}
```

This time the `Event` instance as passed to `createEvent` needs to look as in the following example,

```
Event* anEvent = com::apama::event::createEvent(
    "SharePrice(25,\"ACME\")");
```

The `Echo` monitor's active listener will be triggered by each of the `TestEvent` events passed into the event correlator. On every match it will execute the action specified; which is to create a new `TestEvent` containing a `string` parameter equivalent to the one matched upon. Clearly this is not very useful in practice, but here it serves as a good example.

The `sendEvent` method of the developer-defined `receive_consumer` should now have been invoked twice.

Finally, some cleanup is necessary before terminating.

First, disconnect and destroy the event consumer:

```
delete consumer;
```

then disconnect from the event correlator:

```
com::apama::engine::disconnectFromEngine(engine);
```

and finally clean up the client library:

```
com::apama::engine::engineShutdown();
```

Using the SDKs – C++ and Java Examples

The full example in C++

The complete example, with all the appropriate exception handling and error processing, is presented below. The source file for this example is available as `engine_client.cpp` and is available in the `samples\engine_client\cpp` directory of the Apama installation.

Building and running the example is easy. Full instructions are available in the `README.txt` contained in the same folder.

```
/*
 * engine_client.cpp
 *
 * This simple example illustrates how to use the C++ SDK to
 * interface with Apama. The example connects to a remote Event Correlator
 * (also called a Correlation Engine or just the Engine), creates
 * an event consumer and connects it to the Engine's supplier interface,
 * creates some MonitorScript code and injects it, injects some sample events,
 * and receives some back from the Engine when the monitor triggers. It then
 * disconnects from the Engine and exits.
 *
 * This sample uses the C++ SDK. It has been tested against only
 * CC 5.5 on Solaris or GCC 3.0.4 on Linux.  Other compilers may
 * generate code that will not link with the Apama runtime library.
 *
 * Copyright(c) 2002, 2004-2005 Software AG.  All rights
 * reserved.  Use, reproduction, transfer, publication or disclosure is
 * prohibited except as specifically provided for in your License Agreement
 * with PSC.
 *
 * $RCSfile: engine_client.cpp,v $ $Revision: 1.5.6.1 $ $Date: 2006/04/03 12:31:20 $
 */

#include <engine_client_cpp.hpp>
#include <iostream>
#include <stddef.h>
#include <stdlib.h>
#ifdef __unix__
#include <unistd.h>
#endif
#ifdef __WIN32__
#include <windows.h>
```

```
#endif

using namespace std;

using com::apama::engine::EngineManagement;
using com::apama::engine::EngineStatus;
using com::apama::engine::MonitorScript;
using com::apama::event::EventConsumer;
using com::apama::event::EventSupplier;
using com::apama::event::Event;
using com::apama::EngineException;

/**
 * Event receiver implementation.
 */
class receive_consumer : public EventConsumer {

  public:
    /**
     * Constructor creates the connection to the Engine.
     *
     * @param engine The Engine to connect to.
     *
     * @exception EngineException
     */
    receive_consumer(EngineManagement* engine);

    /**
     * Destructor disconnects from Engine.
     */
    virtual ~receive_consumer();

    /**
     * Receive events from the Engine and log them to stdout.
     *
     * @param events The received events.
     *
     * @exception EngineException
     */
    virtual void sendEvents(const Event* const * events);

  private:
    /** The Engine we're connected to */
    EngineManagement* engine;

    /** Per-connection resource handle returned by the Engine */
    EventSupplier* supplier;
};

/**
 * Create connection to engine.
 */
receive_consumer::receive_consumer(EngineManagement* eng) : EventConsumer(),
      engine(eng), supplier(NULL) {
  // Make the connection.  The returned EventSupplier is a handle to
  // private resources allocated by the Engine to deal with this
  // connection.  These will be freed by calling
  // deleteEventSupplier() in the destructor for this class.
  supplier = engine->connectEventConsumer(this, NULL);
}

/**
 * Disconnect from Engine
 */
receive_consumer::~receive_consumer() {
  // Disconnect from the Engine and free per-connection resources
  com::apama::event::deleteEventSupplier(supplier);
}

/**
```

```
 * Receive a batch of events, log to stdout.
 */
void receive_consumer::sendEvents(const Event* const * events) {
  for (const Event* const * event = events; *event; event++) {
    cout << **event << endl;
  }
}

/**
 * Main program.
 *
 * Return codes:
 *   0 = Everything OK
 *   1 = Couldn't connect to Engine
 *   2 = Something else went wrong
 */
int main(int argc, const char** argv) {
  // Return code
  int rc = 2;

  // Error message to display if anything goes wrong.  Update this
  // appropriately before each operation that might break.
  const char* emsg;

  if (argc == 3 && atoi(argv[2]) > 0) {

    // Set to true once the Engine library has been initialised
    bool initDone = false;

    // The engine
    EngineManagement* engine = NULL;

    try {
      try {
        // Initialise Engine client-side library
        rc = 1;
        emsg = "Failed to initialise engine library";
        com::apama::engine::engineInit();
        initDone = true;

        // Attempt to connect to a remote Engine
        emsg = "Failed to connect to engine";
        engine = com::apama::engine::connectToEngine(argv[1], atoi(argv[2]));

        // Create an event consumer
        emsg = "Event sink connection failed";
        receive_consumer* consumer = new receive_consumer(engine);

        // Inject some MonitorScript (don't forget to delete it when done)
        emsg = "MonitorScript injection failed";
        MonitorScript* script = com::apama::engine::createMonitorScript(
          "event TestEvent {"
            "string text;"
          "}"
          ""
          "monitor Echo {"
          ""
            "TestEvent test;"
          ""
            "action onload {"
              "on all TestEvent(*):test {"
                "emit TestEvent(test.text);"
              "}"
            "}"
          "}");
        engine->injectMonitorScript(*script);
        com::apama::engine::deleteMonitorScript(script);

        // Wait a few seconds to be sure the injection event has been processed
#ifdef __unix__
```

```
        sleep(3);
#endif
#ifdef __WIN32__
        Sleep(3000);
#endif
        // Get & display status (have to delete it when done)
        emsg = "Status gathering failed";
        EngineStatus* status = engine->getStatus();
        cout << *status << endl;
        com::apama::engine::deleteStatus(status);

        // Send some events (again, remember to delete Event objects when done)
        emsg = "Event sending failed";
        Event* events[3];
        events[0] = com::apama::event::createEvent("TestEvent(\"Hello, World\")");
        events[1] = com::apama::event::createEvent("TestEvent(\"Welcome to Apama\")");
        events[2] = NULL;
        engine->sendEvents(events);
        com::apama::event::deleteEvent(events[0]);
        com::apama::event::deleteEvent(events[1]);

        // Delete the event type and monitor we added
        emsg = "Name deletion failed";
        engine->deleteName("Echo");
        engine->deleteName("TestEvent");

        // Wait a few seconds for the output event to be received
        // and the deletions processed
#ifdef __unix__
        sleep(3);
#endif
#ifdef __WIN32__
        Sleep(3000);
#endif

        // Display status again
        emsg = "Status gathering failed";
        cout << endl;
        status = engine->getStatus();
        cout << *status << endl;
        com::apama::engine::deleteStatus(status);

        // Disconnect and destroy the event consumer
        emsg = "Event sink disconnection failed";
        delete consumer;

        // If we got this far, everything succeeded!
        rc = 0;
      }
      catch (EngineException& ex) {
        // Rethrow so exception printer can deal with it
        throw ex;
      }
      catch (...) {
        throw EngineException("Caught non-engine exception in main()");
      }
    }
    catch (EngineException& ex) {
      cerr << emsg << ": " << ex.what() << endl;
    }

    try {
      try {
        // Shutdown cleanly.
        if (engine) {
          // Disconnect from the Engine
          emsg = "Failed to disconnect from engine";
          com::apama::engine::disconnectFromEngine(engine);
        }
        if (initDone) {
```

```
          // Shutdown the Engine library
          emsg = "Failed to shutdown engine library";
          com::apama::engine::engineShutdown();
        }
      }
      catch (EngineException& ex) {
        // Rethrow so exception printer can deal with it
        throw ex;
      }
      catch (...) {
        throw EngineException("Caught non-engine exception in main()");
      }
    }
    catch (EngineException& ex) {
      cerr << emsg << ": " << ex.what() << endl;
    }
  }
  else {
    // Bad command line given
    cerr << "Usage: " << argv[0] << " <host> <port>" << endl;
  }

  // Done!
  return rc;
}
```

Using the SDKs – C++ and Java Examples

The Java example

The Java example is effectively identical to the C++ example described in the previous topic. The source file for this example is available as `JavaExample.java` and is available in the `samples\engine_client\java` folder of the Apama installation.

Building and running the example is easy. Full instructions are available in the `README.txt` contained in the same folder.

```
//*
/*
 * JavaExample.java
 *
 * This simple example illustrates how to use the SDK for Java to
 * interface with Apama. The example connects to a remote Event Correlator
 * (also known as a Correlation Engine or just the Engine), creates
 * an event consumer and connects it to the Engine's supplier interface,
 * creates some MonitorScript code and injects it, injects some sample events,
 * and receives some back from the Engine when the monitor triggers. It then
 * disconnects from the Engine and exits.
 *
 * $Copyright(c) 2013 Software AG, Darmstadt, Germany and/or its licensors$
 *
 * $RCSfile$ $Revision$ $Date$
 */
import com.apama.engine.EngineManagementFactory;
import com.apama.engine.EngineManagement;
import com.apama.engine.EngineStatus;
import com.apama.engine.MonitorScript;
import com.apama.event.EventConsumer;
import com.apama.event.EventSupplier;
import com.apama.event.Event;
import com.apama.EngineException;
/**
 * Event receiver implementation.
 */
class ReceiveConsumer implements EventConsumer {
  /** Per-connection resource handle returned by the Engine */
```

```
  private EventSupplier supplier;
  /**
   * Constructor creates the connection to the Engine.
   *
   * @param engine The Event Correlator to connect to.
   *
   * @exception EngineException
   */
  public ReceiveConsumer(EngineManagement engine) throws EngineException {
    // Make the connection.  The returned EventSupplier is a handle to
    // private resources allocated by the engine to deal with this
    // connection.
    supplier = engine.connectEventConsumer(this, new String[] {});
  }
  /**
   * Receive events from the Engine and log them to stdout.
   *
   * @param events The received events.
   *
   * @exception EngineException
   */
  public void sendEvents(Event[] events) throws EngineException {
    if (events!=null) {
      for (int i=0; i<events.length; i++) {
        System.out.println(events[i]);
      }
    }
  }
  /** No destructors in Java, so we need to explicitly deregister the consumer. */
  public void deregister() throws EngineException {
    if (supplier!=null)
        supplier.disconnect();
  }
} // non-public class ReceiveConsumer
/**
 * Main program.
 *
 * Return codes:
 *   0 = Everything OK
 *   1 = Couldn't connect to Engine
 *   2 = Something else went wrong
 */
public class JavaExample {
  public static void main(String[] argv) {
    // Return code
    int rc = 2;
    // Error message to display if anything goes wrong.  Update this
    // appropriately before each operation that might break.
    String emsg = null;
    if ((argv.length == 2) && (Integer.parseInt(argv[1]) > 0)) {
      // The engine
      EngineManagement engine = null;
      try {
        try {
          rc = 1;
          // Attempt to connect to a remote Engine
          emsg = "Failed to connect to engine";
          engine = EngineManagementFactory.connectToEngine(argv[0],
              Integer.parseInt(argv[1]));
          // Create an event consumer
          emsg = "Event sink connection failed";
          ReceiveConsumer consumer = new ReceiveConsumer(engine);
          // Inject some MonitorScript (don't forget to delete it when done)
          emsg = "MonitorScript injection failed";
          MonitorScript script = new MonitorScript(
            "event TestEvent {" +"\n" +
              "string text;" +"\n" +
            "}" +"\n" +
            "" +"\n" +
            "monitor Echo {" +"\n" +
```

```
            "" +"\n" +
              "TestEvent test;" +"\n" +
            "" +"\n" +
              "action onload {" +"\n" +
                "on all TestEvent(*):test {" +"\n" +
                  "emit TestEvent(test.text);" +"\n" +
                "}" +"\n" +
              "}" +"\n" +
            "}");
          engine.injectMonitorScript(script);
          // Wait a few seconds to be sure the injection event has been processed
          Thread.sleep(3000);
          // Get & display status (have to delete it when done)
          emsg = "Status gathering failed";
          EngineStatus status = engine.getStatus();
          System.out.println(status);
          // Send some events (In the Java API we do NOT need to null
          // terminate the array)
          emsg = "Event sending failed";
          Event[] events = new Event[2];
          events[0] = new Event("TestEvent(\"Hello, World\")");
          events[1] = new Event("TestEvent(\"Welcome to Apama\")");
          engine.sendEvents(events);
          // Delete the event type and monitor we added
          emsg = "Name deletion failed";
          engine.deleteName("Echo");
          engine.deleteName("TestEvent");
          // Wait a few seconds for the output event to be received and
          // the deletions processed
          Thread.sleep(3000);
          // Display status again
          emsg = "Status gathering failed";
          System.out.println();
          status = engine.getStatus();
          System.out.println(status);
          // Disconnect and destroy the event consumer
          emsg = "Event sink disconnection failed";
          consumer.deregister();
          // If we got this far, everything succeeded!
          rc = 0;
        }
        catch (EngineException ex) {
          // Rethrow so exception printer can deal with it
          throw ex;
        }
        catch (Throwable t) {
          throw new EngineException("Caught non-engine exception in main()");
        }
      }
      catch (EngineException ex) {
        System.err.println(emsg + ": " + ex);
      }
    }
    else {
      // Bad command line given
      System.out.println("Usage: java JavaExample <host> <port>");
    }
    // Done!
    System.exit(rc);
  } // main()
} // class JavaExample
```

Using the SDKs – C++ and Java Examples

Chapter 4: The C Client Software Development Kit

This topic explores the C version of the Client Software Development Kit. There may be several reasons for wishing to integrate with the event correlator using C instead of C++ or Java, not least if greater flexibility is required with regards to which compilers one needs to work with. The C standard has been mature for considerably longer than C++ so it should be possible to develop, compile and link against the C development kit with a large variety of C compilers and libraries.

Note: Due to the fact that the underlying Apama libraries are built with and include C++ code, you still need to link against the standard C++ library. Failure to do this might result in your clients failing immediately upon startup.

Whereas the SDKs for C++ and for Java are packaged through a set of classes that model a set of entities, the C SDK consists of a set of functions organized within structures. The function names are similar to the method names used within the classes. The structures are largely organized to resemble the C++ classes.

Using the C SDK

Note that the C SDK comes in two 'flavors' – one for pure C-only compilers, and one for C++ compilers. If you need to use the C SDK because your C++ compiler is not ISO compatible or is not supported by Apama, then you might find the C++ guise of the C SDK easier to work with. Both are available in the header file `engine_client_c.h`. This header file determines what kind of compiler is in use and defines its contents accordingly. `engine_client_c.h` can be located in the `include` folder.

While the reader is referred directly to the header file `engine_client_c.h` as a reference, this topic introduces the C SDK primarily through inspection of one of the included examples. There are two examples that use the C SDK in the `samples\engine_client` folder. They are `c\engine_client.c` and `cpp \engine_client_c.cpp`. Both use the C SDK, but while `engine_client.c` is intended as a pure C program written with a C only compiler, `engine_client_c.cpp` can be compiled with a C++ compiler.

This section concentrates on `c\engine_client.c`.

The first step is to initialize the Apama client-side library;

```
AP_EngineInit();
if (AP_CheckException()) {
  exThrown = 1;
  break;
}
```

Note the explicit 'exception' handling. As in C there is no notion of exceptions, the C SDK provides an analogous means of verifying that a method has succeeded. It is important that the developer use this mechanism with the majority of functions provided in its API.

Note: For clarity, the rest of these example code snippets do not include exception handling.

The next step is to connect to a remote correlator (or engine). As with the other development kits this method takes the host and port of the correlator to connect to.

```
engine = AP_ConnectToEngine(argv[1], atoi(argv[2]));
```

At this stage the example creates an 'event consumer' entity to connect to the correlator. It needs to register a function that will be invoked whenever the correlator emits event alerts.

To achieve this;

```
consumer = (AP_EventConsumer*)malloc(sizeof(AP_EventConsumer));
consumer->functions = &receiveConsumer_Functions;
supplier = engine->functions->connectEventConsumer(engine, consumer,
NULL);
```

The first line above creates a 'consumer' structure. This contains a pointer to a table of callback functions, which in this release of the SDK can in fact contain only one function, the function to be called when event alerts are generated. In fact if we go backwards to the beginning of the example, the following definitions were made before `main()`;

```
/**
 * Receive a batch of events, log to stdout.
 */
static void AP_ENGINE_CLIENT_CALL
  receiveConsumer_sendEvents(AP_EventConsumer* consumer,
  AP_Event** events) {
  AP_Event** event;
  for (event = events; *event; event++) {
    printf("%s\n", (*event)->functions->getText(*event));
  }
}
/**
 * Function table for our event consumer.
 */
static struct AP_EventConsumer_Functions
  receiveConsumer_Functions = {
  receiveConsumer_sendEvents
};
```

The second definition above, the structure `receiveConsumer_Functions`, is largely boilerplate code and a function similar to it needs to be defined for every consumer. It defines the table of callback functions (which in fact can only contain one function at present). The callback function itself is the first function above, here called `receiveConsumer_sendEvents`, which just prints out the alerts received.

The next part of the example defines and injects some EPL controlling code into the correlator;

```
/* Inject some MonitorScript (don't forget to delete it when done) */
script = AP_CreateMonitorScript(
  "event TestEvent {"
  "  string text;"
  "}"
  ""
  "monitor Echo {"
  ""
  "  TestEvent test;"
  ""
  "  action onload {"
  "    on all TestEvent(*):test {"
  "      emit TestEvent(test.text);"
  "    }"
  "  }"
  "}");
  engine->functions->injectMonitorScript(engine, script);
  AP_DeleteMonitorScript(script);
```

Note how first you define a string of EPL code, then inject it into the correlator, and finally delete it.

The example then proceeds to query the correlator as to its present runtime status, and then injects some events;

```
events[0] = AP_CreateEvent(
  "TestEvent(\"Hello, World\")");
events[1] = AP_CreateEvent(
  "TestEvent(\"Welcome to Apama\")");
events[2] = NULL;
engine->s_EventConsumer->functions->
  sendEvents(engine->s_EventConsumer, events);
AP_DeleteEvent(events[0]);
AP_DeleteEvent(events[1]);
```

In this instance two events are going to be injected. The injection method always takes in a batch of events in a structure. Injecting events in batches results in greater event throughput as it lessens the overhead of making the underlying process-to-process call. Typically a batch size of one hundred produces optimal performance, although this varies according to the size of the individual events.

Note how the events themselves are deleted at the end of the above code.

These events will match with the monitor injected earlier, so the correlator will produce an alert for each and call the callback function registered earlier.

The next steps in the main body of the example are to delete the monitor and event type definitions made earlier from the correlator and then recheck its status to verify that the types are no longer defined.

Finally it is cleanup time. The steps here are to disconnect the consumer structure from the correlator, and then destroy it.

```
supplier->functions->disconnect(supplier);
free((void*)consumer);
```

The next and final steps are to disconnect the SDK library from the correlator and delete (or shutdown) the library itself.

```
AP_DisconnectFromEngine(engine);
AP_EngineShutdown();
```

This concludes this very simple example. The whole example, complete with exception and error handling is given below.

As described in "Logging in C++" on page 16 for the C++ SDK, the C SDK can output extensive logging information. The author of a C client need not bother with the standard logging unless they want to modify its operating parameters.

By default, the log level is set to `WARN`, where only significant warnings and errors are displayed in the log. The whole list of log levels is `OFF` (i.e. no logging at all), `CRIT`, `FATAL`, `ERROR`, `WARN`, `INFO`, `DEBUG` and `TRACE`. These levels are listed in order of decreasing importance, and conversely in the order of least likely occurrence. A very large volume of information is output at `DEBUG` level.

To change the logging, three functions are provided in the C SDK; `AP_SetLogLevel()`, `AP_SetLogFile()`, and `AP_SetLogFD()`.

As described in "Thread-safety" on page 17 for the C++ SDK, the C SDK is thread-safe.

The C Client Software Development Kit

The complete C example

This is the C example, `samples\engine_client\c\engine_client.c`, in its entirety, complete with all exception and error handling. The reader is invited to peruse the alternative example, `samples\engine_client\cpp\engine_client_c.cpp`, for how to write an identically functional sample in the context of a C++ compiler, yet still using the C SDK.

```
/*
 * engine_client.c
 *
 * This simple example illustrates how to use the C SDK to
 * interface with Apama. The example connects to a remote Event Correlator
 * (also known as a Correlation Engine), creates an event consumer and connects
 * it to Apama's supplier interface, creates some MonitorScript code and injects it,
 * injects some sample events, and receives some back from the Engine when the
 * monitor triggers. It then disconnects from the Engine and exits.
 *
 * Copyright(c) 2002, 2004-2005 Software AG.  All rights
 * reserved.  Use, reproduction, transfer, publication or disclosure is
 * prohibited except as specifically provided for in your License Agreement
 * with PSC.
 *
 * $RCSfile: engine_client.c,v $ $Revision: 1.5.6.1 $ $Date: 2006/04/03 12:31:20 $
 */
#include <engine_client_c.h>
#include <stddef.h>
#include <stdlib.h>
#ifdef __unix__
#include <unistd.h>
#endif
#ifdef __WIN32__
#include <windows.h>
#endif
#include <stdio.h>
#define STATUS_BUFFER_SIZE 8192
/**
 * Receive a batch of events, log to stdout.
 */
static void AP_ENGINE_CLIENT_CALL receiveConsumer_sendEvents(
       AP_EventConsumer* consumer, AP_Event** events) {
  AP_Event** event;
  for (event = events; *event; event++) {
    printf("%s\n", (*event)->functions->getText(*event));
  }
}
/**
 * Function table for our event consumer.
 */
static struct AP_EventConsumer_Functions receiveConsumer_Functions = {
  receiveConsumer_sendEvents
};
/**
 * Main program.
 *
 * Return codes:
 *   0 = Everything OK
 *   1 = Couldn't connect to engine
 *   2 = Something else went wrong
 */
int main(int argc, const char** argv) {
  /* Return code */
  int rc = 2;
  /* Error message to display if anything goes wrong.  Update this */
  /* appropriately before each operation that might break. */
  const char* emsg;
  /* Buffer for stringified status reports */
  AP_char8 statusBuf[STATUS_BUFFER_SIZE];
  if (argc == 3 && atoi(argv[2]) > 0) {
    /* Set to true once the engine library has been initialised */
    AP_bool initDone = 0;
    /* Set to true to drop out of the loop with a pending exception */
```

```
    AP_bool exThrown = 0;
    /* The engine */
    AP_EngineManagement* engine = NULL;
    do {
      /* The event consumer */
      AP_EventConsumer* consumer = NULL;
      /* The (remote) event supplier reference */
      AP_EventSupplier* supplier = NULL;
      /* Monitorscript to be injected into engine */
      AP_MonitorScript* script = NULL;
      /* Engine status report */
      AP_EngineStatus* status = NULL;
      /* Events to be sent */
      AP_Event* events[3];
      /* Initialise Apama SDK client-side library */
      rc = 1;
      emsg = "Failed to initialise Apama SDK library";
      AP_EngineInit();
      if (AP_CheckException()) {
        exThrown = 1;
        break;
      }
      initDone = 1;
      /* Attempt to connect to a remote Engine */
      emsg = "Failed to connect to engine";
      engine = AP_ConnectToEngine(argv[1], (AP_uint16) atoi(argv[2]));
      if (AP_CheckException() || !engine) {
        exThrown = 1;
        break;
      }
      /* Create an event consumer */
      emsg = "Event sink connection failed";
      consumer = (AP_EventConsumer*)malloc(sizeof(AP_EventConsumer));
      consumer->functions = &receiveConsumer_Functions;
      supplier = engine->functions->connectEventConsumer(engine, consumer, NULL);
      if (AP_CheckException() || !supplier) {
        exThrown = 1;
        break;
      }
      /* Inject some MonitorScript (don't forget to delete it when done) */
      emsg = "MonitorScript injection failed";
      script = AP_CreateMonitorScript(
          "event TestEvent {"
            "string text;"
          "}"
          ""
          "monitor Echo {"
          ""
            "TestEvent test;"
          ""
            "action onload {"
              "on all TestEvent(*):test {"
                "emit TestEvent(test.text);"
              "}"
            "}"
          "}");
      engine->functions->injectMonitorScript(engine, script);
      if (AP_CheckException()) {
        exThrown = 1;
        break;
      }
      AP_DeleteMonitorScript(script);
      if (AP_CheckException()) {
        exThrown = 1;
        break;
      }
      /* Wait a few seconds to be sure the injection event has been processed */
#ifdef __unix__
      sleep(3);
#endif
```

```
#ifdef __WIN32__
      Sleep(3000);
#endif
      /* Get & display status (have to delete it when done) */
      emsg = "Status gathering failed";
      printf("\n");
      status = engine->functions->getStatus(engine);
      if (AP_CheckException() || !status) {
        exThrown = 1;
        break;
      }
      if (status->functions->print(status, statusBuf, STATUS_BUFFER_SIZE)) {
        printf("%s", statusBuf);
      }
      else {
        printf("Status buffer too small!");
      }
      printf("\n");
      AP_DeleteEngineStatus(status);
      /* Send some events (again, remember to delete Event objects when done) */
      emsg = "Event sending failed";
      events[0] = AP_CreateEvent("TestEvent(\"Hello, World\")");
      events[1] = AP_CreateEvent("TestEvent(\"Welcome to Apama\")");
      events[2] = NULL;
      engine->s_EventConsumer->functions->sendEvents(engine->s_EventConsumer, events);
      AP_DeleteEvent(events[0]);
      AP_DeleteEvent(events[1]);
      if (AP_CheckException()) {
        exThrown = 1;
        break;
      }
      /* Delete the event type and monitor we added */
      emsg = "Name deletion failed";
      engine->functions->deleteName(engine, "Echo");
      if (AP_CheckException()) {
        exThrown = 1;
        break;
      }
      engine->functions->deleteName(engine, "TestEvent");
      if (AP_CheckException()) {
        exThrown = 1;
        break;
      }
      /* Wait a few seconds for the output event to be received and
         the deletions processed */
#ifdef __unix__
      sleep(3);
#endif
#ifdef __WIN32__
      Sleep(3000);
#endif
      /* Display status again */
      emsg = "Status gathering failed";
      printf("\n");
      status = engine->functions->getStatus(engine);
      if (AP_CheckException() || !status) {
        exThrown = 1;
        break;
      }
      if (status->functions->print(status, statusBuf, STATUS_BUFFER_SIZE)) {
        printf("%s", statusBuf);
      }
      else {
        printf("Status buffer too small!");
      }
      printf("\n");
      AP_DeleteEngineStatus(status);
      /* Disconnect and destroy the event consumer */
      emsg = "Event sink disconnection failed";
      supplier->functions->disconnect(supplier);
```

```
      if (AP_CheckException()) {
        exThrown = 1;
        break;
      }
      free((void*)consumer);
      /* If we got this far, everything succeeded! */
      rc = 0;
    } while (0);
    if (exThrown) {
      if (AP_CheckException()) {
        printf("%s: %s\n", emsg, AP_GetExceptionMessage());
      }
      else {
        printf("%s\n", emsg);
      }
      exThrown = 0;
      AP_ClearException();
    }
    do {
      /* Shutdown cleanly */
      if (engine) {
        /* Disconnect from the engine */
        emsg = "Failed to disconnect from Engine";
        AP_DisconnectFromEngine(engine);
        if (AP_CheckException()) {
          exThrown = 1;
        }
      }
      if (initDone) {
        /* Shutdown the engine library */
        emsg = "Failed to shutdown Apama SDK library";
        AP_EngineShutdown();
        if (AP_CheckException()) {
          exThrown = 1;
        }
      }
    } while (0);
    if (exThrown) {
      if (AP_CheckException()) {
        printf("%s: %s\n", emsg, AP_GetExceptionMessage());
      }
      else {
        printf("%s\n", emsg);
      }
      exThrown = 0;
      AP_ClearException();
    }
  }
  else {
    /* Bad command line given */
    fprintf(stderr, "Usage: %s <host> <port>\n", argv[0]);
  }
  /* Done! */
  return rc;
}
```

The C Client Software Development Kit

Chapter 5: The JavaBeans API

We recommend that where possible Java developers should use the JavaBeans API. This API offers the most powerful combination of flexibility and simplicity provided by Apama for programmatic interfacing to the event correlator.

The full reference to the JavaBeans API is its Javadoc documentation, which you can peruse at `doc \javadoc\index.html`. This topic introduces the different elements included in the API, and walks through an example.

The key elements

The key elements included in the JavaBeans API are the `EngineClientBean` provided in the package `com.apama.engine.beans` and the `GenericComponentManagementBean` included in `com.apama.net.beans`.

The JavaBeans API

Overview of the EngineClientBean

The following functionality is provided in the `EngineClientBean`:

- Delete registered names from a correlator.
- Inject new EPL code into a correlator.
- Gather information from a correlator about the set of Monitors and Event Types that it is currently working with.
- Register to receive events from a correlator.
- Send events to a correlator.
- Provide access to the status information from a correlator.

The JavaBeans API

Functionality of the EngineClientBean

The EngineClientBean implements the interface `com.apama.net.beans.interfaces.PingClientInterface`.

The bean inherits from the abstract class `com.apama.engine.beans.AbstractEngineClientBean`.

The EngineClientBean provides a number of common methods and properties, like "`host`", "`port`", "`verbose`", "`connectionPollingInterval`" and "`beanConnected`". The full names are `PROPERTY_HOST`, `PROPERTY_PORT`, `PROPERTY_VERBOSE`, `PROPERTY_CONNECTION_POLLING_INTERVAL` and `PROPERTY_BEAN_CONNECTED`.

Each bean is capable of making its own connection to an event correlator and maintains that connection. That is, it can monitor that the connection is live throughout, and flag if the connection is dropped. This can occur if the network connection is lost or the remote correlator is terminated.

Changes to the value of the "`host`" or "`port`" properties will cause the bean to attempt to re-connect to an event correlator running on a new host/port. This re-connection will happen immediately if the bean was connected at the time of the property change, but will happen later in a "lazy" fashion if there was no existing connection at the time of the property change.

Beans also maintain a Boolean bound property called "`beanConnected`". The value will be set to `true` following a successful connection to an Engine, and to `false` at disconnection. Once connected, a background thread will periodically ping the remote Engine. This background thread will also maintain the value of the "`beanConnected`"property each time it tests the connection.

Warning: The bound properties supported by this bean will notify all registered listeners from within a synchronized block. Listeners should not invoke methods that would wait for another thread to complete a call into this bean. For example, calls to `System.exit(int)` would cause a deadlock situation when the shutdown handler thread attempts to call the `disconnect()` method. Listeners on the "`events`" property or EventListeners added to Consumers may not re-enter the bean, even directly. When registering a consumer, if the `async` parameter is supplied and set to `true`, then the consumer's listeners are called asynchronously – events are queued and delivered on a separate thread, and the EventListeners are permitted to call the EngineClientBean.

(See `addConsumers` in the Javadoc). Note that asynchronous consumers may be called after having been disconnected.

The EngineClientBean inherits the following methods:

- `void setHost(java.lang.String newHostValue)` - Setter for the "`host`" property. This is the name of the host on which the correlator to be monitored is running. Changing this property will cause any existing connection to be lost. If there was an existing connection, then a new connection will be created.
- `void setPort(int newPortValue)` - Setter for the "`port`" property. This is the port number on which the correlator to be monitored is listening. Changing this property will cause any existing connection to be lost. If there was an existing connection, then a new connection will be created.
- `void connectNow()` - Manually request that the bean connects to the remote event correlator (or 'server'). Repeated calls are permitted, and attempting to connect a bean that is already connected is identical to calling the `pingServer()` method. Note that other methods can implicitly cause a connection to be created.

- `void disconnect()` - This method must always be called before quitting the program that is using the bean. If the method is not called, then an event correlator (or 'engine') could be left in a state where it is attempting to send events to a non-existent client (or 'consumer').
- `java.lang.String getHost()` - Get the name of the host to be connected to. This is the name of the host on which the correlator to be monitored is running.
- `int getPort()` - Get the port number to be connected to. This is the port number on which the correlator to be monitored is listening.
- `boolean getBeanConnected()` - Get the "`beanConnected`"property's value. This is the status of the bean - connected, or not connected. It indicates if the Bean has a valid instance of the underlying RPC interface.
- `boolean isBeanConnected()` - Another name for the `getBeanConnected()` method.
- `public void setVerbose(boolean newVerboseValue)` - Setter for the `verbose` property. When `verbose` is set to true, some methods (in subclasses) will print progress messages on stdout.
- `public boolean getVerbose()` - Getter for the `verbose` property. When `verbose` is set to `true`, some methods (in subclasses) will print progress messages on `stdout`.
- `void setConnectionPollingInterval(int milliseconds)` - Set the polling interval (in milliseconds) for the internal connection test thread. If the parameter is negative, then the default value will be used instead.
- `int getConnectionPollingInterval()` - Get the polling interval (in milliseconds) for the internal connection test thread.
- `void pingServer()` - Manually test if the remote engine process is alive and responding to client requests. This method makes a no-arg, `void` return, method call on the client interface of the engine. If a connection is not yet established, this method will request a connection. In the event that a connection cannot be established, or an error during the ping, an `EngineException` will be raised.
- `void addPropertyChangeListener(java.beans.PropertyChangeListener listener)` - Add a property change listener for a property of the bean.
- `void removePropertyChangeListener(java.beans.PropertyChangeListener listener)` - Remove a property change listener.
- `void addPropertyChangeListener(java.lang.String propertyName, java.beans.PropertyChangeListener listener)` - Add a property change listener for a specific named property.
- `void removePropertyChangeListener(java.lang.String propertyName, java.beans.PropertyChangeListener listener)` - Remove a property change listener for a specific named property.

Note: Starting with Release 3.0 the following EngineClientBean methods have been moved to the GenericComponentManagementBean; for more information see "GenericComponentManagementBean" on page 62.

- `boolean deepPing()`
- `long getPID()`
- `void shutdown(java.lang.String why)`

Overview of the EngineClientBean

Recommended usage

The recommended way of using the EngineClient Bean is as follows

1. Call the default constructor of the bean (the one with no parameters)
2. Call `setHost()` to set the host on which the remote correlator is running
3. Call `setPort()` to set the port on which to contact the remote correlator
4. Call `connectNow()` or any other method which creates a connection to the remote correlator. All the specialized operations listed in the following sections will create a connection if one does not already exist when they are called.

Overview of the EngineClientBean

Logging

As described in "The Client Software Development Kits for C++ and Java" on page 11 with regards to the Client SDK for Java, the underlying client libraries, and the beans themselves, log information pertaining to their operation.

Authors of Java clients need not bother with the standard logging unless they want to modify its operating parameters. By default the SDK classes will log at `WARN` level. The log level can be changed as described in the Javadoc for the `com.apama.util.Logger` class.

The Javadoc also provides instructions on how to get a reference to the Logger object in your own code so that you can produce your own logging output.

Overview of the EngineClientBean

Delete operations

The delete operations are defined in the interface `com.apama.engine.beans.interfaces.DeleteOperationsInterface`.

These are implemented in `com.apama.engine.beans.EngineClientBean`.

Note the distinction between *delete*, *forced delete*, and *kill*.

- Delete will only work if the EPL element being deleted (an event type or monitor) is not referenced by any other element. So, for example, you cannot *delete* an event type that is used by any monitors.
- *Forced delete* will delete the specified element *as well as all other elements that refer to it*. 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.
- *Kill* is a stronger form of *forced delete*. *Forced delete* might fail to remove a monitor that is stuck in an infinite loop, whereas *kill* would remove the monitor at the next loop iteration.

The delete operations are:

- `void deleteAll()` - Deletes everything from the remote correlator (or *engine*). This is equivalent to a *kill* being applied to everything.
- `void deleteName(java.lang.String name, boolean force)` - *Delete* an EPL event type or monitor (i.e., a *name*) from the remote correlator.
- `void deleteNames(java.util.List<java.lang.String> names, boolean force)` - *Delete* a number of EPL elements.
- `void deleteNamesFromFile(java.util.List<java.lang.String> filenames, boolean force)` - *Delete* a number of EPL elements listed in one or more filenames (or `stdin`).
- `void deleteNamesFromFile(java.util.List<java.lang.String> filenames, boolean force, boolean utf8)` - *Delete* a number of EPL elements listed in one or more filenames (or `stdin`).
- `void killName(java.lang.String name)` - Kill an EPL element from the remote correlator.
- `void killNames(java.util.List<java.lang.String> names)` - *Kill* a number of EPL elements.
- `void killNamesFromFile(java.util.List<java.lang.String> filenames)` - *Kill* a number of EPL elements listed in one or more filenames (or `stdin`).
- `void setCancelDeleteFileRead(boolean newCancelFileReadValue)` - Setter for the "`cancelFileRead`" property for the Delete / Kill operations. The purpose of the "`cancelFileRead`" property is to provide a mechanism to cleanly terminate the processing of deletion of elements from a file, when the `deleteNamesFromFile` method is in progress. When "`cancelFileRead`" is set to `true`, the deleting loop will terminate at the next iteration.

The JavaBeans API

Inject operations

The inject operations are defined in the interface `com.apama.engine.beans.interfaces.InjectOperationsInterface`.

These are implemented in `com.apama.engine.beans.EngineClientBean`.

The inject operations are:

- `void injectCDP(byte[] cdpBytes)` - Send the bytes of a CDP(Correlator Deployment Package) to the engine without blocking concurrent calls to operations on this bean instance.
- `void injectCDP(byte[] cdpBytes, String filename)` - Send the bytes of a CDP(Correlator Deployment Package) to the engine without blocking concurrent calls to operations on this bean instance.
- `void injectCDPsFromFile(java.util.List<java.lang.String> filenames)` - Inject one or more CDPs from one or more files or `stdin`.
- `void injectJavaApplication(byte[] jarBytes)` - Inject the bytes of a Java in-process API (JMON) application (`.jar` file) into the remote correlator.
- `void injectJavaApplicationsFromFile(java.util.List<java.lang.String> filenames)` - Inject a number of JMon application(s) from one or more files or `stdin`.

- `void injectMonitorScript(MonitorScript script)` - Inject a `MonitorScript` object into the remote correlator. A `MonitorScript` object encapsulates an EPL code fragment that contains package, event and monitor definitions to be injected into a correlator.
- `void injectMonitorScriptFromFile(java.util.List<java.lang.String> filenames)` - Inject a number of monitors from one or more files or `stdin`.
- `void setCancelInjectFileRead(boolean newCancelFileReadValue)` - Sets the "`cancelFileRead`" property for the inject operations. The purpose of the "`cancelFileRead`" property is to provide a mechanism to cleanly terminate the injection of EPL code from file(s), while the `injectMonitorScriptFromFile()` method is in progress. When "`cancelFileRead`" is set to `true`, the EPL injecting loop terminates at the next iteration

The JavaBeans API

Inspect operations

The inspect operations are defined in the interface `com.apama.engine.beans.interfaces.InspectOperationsInterface`.

These are implemented in `com.apama.engine.beans.EngineClientBean`.

This bean provides the following bound properties; "`engineInfo`" and "`inspectPollingInterval`", or `PROPERTY_ENGINE_INFO` and `PROPERTY_INSPECT_POLLING_INTERVAL`.

The inspect operations are:

- `EngineInfo getEngineInfo()` - Get the most recently recorded inspection information. Note that calling this method does not invoke a remote call to a correlator, but simply returns the last known information as collected by the internal worker thread, if that thread is running.
- `int getInspectPollingInterval()` - Get the `inspectPollingInterval` (in milliseconds) that the background thread should wait between calls for new information.
- `EngineInfo getRemoteEngineInfo()` - Request the remote correlator inspection info. This method will not store the inspection result, and is available as an alternative to the background polling service. If a connection is not yet established, this method will request a connection.
- `void setInspectPollingInterval(int newInspectPollingInterval)` - Set the `inspectPollingInterval` (in milliseconds) that the background thread should wait between calls for new information.
- `void startInspectPollingThread()` - Start the local inspect polling thread.
- `void stopInspectPollingThread()` - Stop the local inspect polling thread.

The JavaBeans API

Receive operations

The receive operations are defined in the interface `com.apama.engine.beans.interfaces.ReceiveConsumerOperationsInterface`. This interface specifies the standard operations that support receiving of events from a remote correlator using uniquely-named consumers.

For complete details about the interface's methods, overloadings, and parameters along with other information, see the Apama Javadoc available at `doc\javadoc\index.html`.

The methods provided by `com.apama.engine.beans.interfaces.ReceiveConsumerOperationsInterface` for handling uniquely-named consumers include the following:

- `addConsumer()`
- `getConsumer`
- `getAllConsumers`
- `isAllConsumersConnected`
- `removeConsumer`
- `removeAllConsumers`

Deprecated operations

As of Apama release 5.0, the receive operations defined in the interface `com.apama.engine.beans.interfaces.ReceiveOperationsInterface` have been deprecated. Applications should use methods defined in `ReceiveConsumerOperationsInterface` instead.

The deprecated receive operations are:

- `getChannels()`
- `isReceiveEnabled()`
- `isReceiverConnected()`
- `setChannels(java.lang.String[] newChannelsValue)`
- `setReceiveEnabled(boolean newReceiveEnabled)`

The JavaBeans API

Send operations

The send operations are defined in the interface `com.apama.engine.beans.interfaces.SendOperationsInterface`.

These are implemented in `com.apama.engine.beans.EngineClientBean`.

The send operations are:

- `void sendEvents(Event[] events)` - Send an array of `Event` objects to the remote correlator. An `Event` object represents an event instance. The events are automatically rebatched with this method.
- `void sendEvents(boolean autoBatch, Event... events)` - Send events into the Engine (inherited from `EventConsumer`), optionally performing auto-batching. If `autoBatch` is true, events will be automatically rebatched to improve throughput (even across separate `sendEvents` calls). The `events` parameter is the array of Events to be sent.
- `void sendEventsFromFile(java.util.List<java.lang.String> filenames, int loop)` - Send a number of events from a file or `stdin`.

- `void sendEventsFromFile(java.util.List<java.lang.String> filenames, int loop,boolean utf8)` - Send a number of events from a file or `stdin`. If a connection is not yet established, this method will request a connection.

  This method will not perform auto-batching. For higher performance, use `sendEventsFromFile(List, int, boolean, boolean)`.

- `void sendEventsFromFile(java.util.List<java.lang.String> filenames, int loop, boolean utf8, boolean autoBatch)` - Send a number of events from a file or `stdin`, specifying the file encoding detection mode and whether to auto-batch events or not. If a connection is not yet established, this method will request a connection.

- `void flushEvents()` - Wait for any outstanding events from previous `SendOperationsInterface#sendEvents(Event...)` calls into the Engine, and then return.

- `void setCancelSendFileRead(boolean newCancelFileReadValue)` - Setter for the "`cancelFileRead`" property for the Send operations. The purpose of the "`cancelFileRead`" property is to provide a mechanism to cleanly terminate the processing of events from file, when the `sendEventsFromFile` method is in progress. When "`cancelFileRead`" is set to true, the event sending loop will terminate at the next iteration.

The JavaBeans API

Watch operations

The watch operations are defined in the interface `com.apama.engine.beans.interfaces.WatchOperationsInterface`.

These are implemented in `com.apama.engine.beans.EngineClientBean`.

The following bound properties are available in this bean; "`status`" and "`statusPollingInterval`", the full names being `PROPERTY_STATUS` and `PROPERTY_STATUS_POLLING_INTERVAL`.

The watch operations are:

- `EngineStatus getRemoteStatus()` - Request the remote correlator (or engine) status. This method will not store the status result, and is available as an alternative to the background polling service. If a connection is not yet established, this method will request a connection.

- `EngineStatus getStatus()` - Get the most recently recorded status. Note that calling this method does not invoke a remote call to a correlator, but simply returns the last known status as collected by the internal worker thread.

- `int getStatusPollingInterval()` - Get the `statusPollingInterval` (in milliseconds) that the background thread should wait between calls for new status information.

- `void setStatusPollingInterval(int newStatusPollingInterval)` - Set the `statusPollingInterval` (in milliseconds) that the background thread should wait between calls for new status information.

- `void startStatusPollingThread()` - Start the local status polling thread.

- `void stopStatusPollingThread()` - Stop the local status polling thread.

The JavaBeans API

GenericComponentManagementBean

The GenericComponentManagementBean provides the following methods, which are described fully in the Javadoc.

- `boolean deepPing()` — Ask the remote server to perform a “deep ping” operation.
- `java.lang.String doRequest(java.lang.String request)` — Execute a component-specific command.
- `java.lang.String getBuildNumber()` — Get the component’s build number.
- `java.lang.String getBuildPlatform()` — Get the component's build platform.
- `java.lang.String getComponentVersion()` — Get the component’s version number.
- `int getRemotePort()` — Get the port number that the component is listening on.
- `java.lang.String getCurrentDirectory()` — Get the component’s current working directory path.
- `java.lang.String getHostname()` — Get the hostname that component is running on.
- `GenericComponentManagement.GenericComponentInfo getInfo(java.lang.String[] categories)` — Request component-specific status/configuration information.
- `java.lang.String[] getInterfaces()` — Get the interfaces exported by the component.
- `long getLogicalId()` — Get the unique logical ID of the component .
- `GenericComponentManagement.GenericComponentLogLevel getLogLevel()` — Get the component's current logging level.
- `java.lang.String getName()` — Get the name of the component, encoded as UTF-8.
- `long getPhysicalId()` — Get the globally unique physical ID of the component.
- `long getPID()` — Return the process identifier of the remote server.
- `java.lang.String getProductVersion()` — Get the version number of the product the component belongs to.
- `java.lang.String getType()` — Get the type of the component, encoded as UTF-8.
- `java.lang.String getUsername()` — Get the effective username the component is running as.
- `boolean isGenericComponentManagementAvailable()` — Return true if the remote server actually implements the GenericComponentManagement interface.
- `static void main(java.lang.String[] args)` — This bean can be invoked from the command prompt.
- `void setLogLevel(GenericComponentManagement.GenericComponentLogLevel logLevel)` — Set the component's logging level.
- `void shutdown(java.lang.String why)` — Tell the remote server to shut itself down.

The JavaBeans API

Chapter 6: The EventService API

The Apama EventService application programming interface (API) is layered on top of the JavaBeans API described in "The JavaBeans API" on page 54. The EventService API interface allows client applications to focus on events and channels.

The key elements

The EventService API consists of two interfaces, `IEventService` and `IEventServiceChannel`, and the `EventServiceFactory` class. These are included in the package `com.apama.services.event`.

The `IEventService` interface provides a simple abstraction over the underlying `EngineClientBean` layer to manage a set of named channels and to send events.

The `IEventServiceChannel` interface provides methods to add and remove listeners, to issue requests to the correlator, and to invoke callback methods when responses to requests are received from the correlator.

The `EventServiceFactory` class provides methods for creating instances of classes that implement the `IEventService` interface.

The complete reference guide for the EventService API is available in HTML format in the `doc\javadoc` directory of the Apama installation.

A selection of sample applications that use the EventService API is available in the Apama installation's `samples\engine_client\javaEventService` directory.

The EventService API

The IEventService interface

To get an EventService object, call the `createEventService()` method of the `com.apama.services.event.EventServiceFactory` class. The returned object implements the `IEventService` interface. With this object the following methods are available:

- `IEventServiceChannel addChannel()` — defined as:

  ```
  IEventServiceChannel addChannel(
      java.lang.String channelName,
      java.util.Map<String, Object> channelConfig)
  ```

  Create an `EventServiceChannel` specifically for listening to a given channel or channels.

- `void destroy()`

  Destroy this service (not the correlator).

- `IEventServiceChannel getChannel(java.lang.String channelName)`

  Get an `EventServiceChannel` for a given channel or channels.

- `EngineClientInterface getEngineClient()`

  Get a handle on the underlying `EngineClient`.

- `boolean isDestroyed()`

  Determine if this service (not the correlator) is destroyed.

- `void removeChannel(java.lang.String channelName)`

  Remove an `EventServiceChannel` for a given channel or channels.

- `void sendEvent(Event event)`

  Send an event to the correlator using the `EngineClient`.

The EventService API

The IEventServiceChannel interface

After you have an EventService object, you can create an EventServiceChannel object by calling the `addChannel()` method. With the EventServiceChannel object the following methods are available:

- `void addEventListener(IEventListener eventListener)`

  Add an `IEventListener` that will be notified of every event (of a pre-registered type) that is received by this EventServiceChannel instance.

- `void addEventListener()` — defined as:

```
void addEventListener(
    IEventListener eventListener,
    EventType eventType)
```

  Add an `IEventListener` that will be notified of every event, of the specified EventType, that is received by this EventServiceChannel instance.

- `void asyncRequestResponse()` — defined as:

```
void asyncRequestResponse(
    IResponseListener responseListener,
    Event requestEvent,
    EventType responseEventType)
```

  Emulate an asynchronous RPC call to the correlator by sending an event, and then invoking a callback when a matching response event is received (non-blocking call).

- `void asyncRequestResponse()` - defined as:

```
void asyncRequestResponse(
    IResponseListener responseListener,
    Event requestEvent,
    EventType responseEventType,
    long timeout)
```

Emulate an asynchronous RPC call to the correlator by sending an event, and then invoking a callback when a matching response event is received (non-blocking call, with timeout).

- `void clearProcessingQueue()`

  Immediately remove (discard) all events from the internal processing queue.

- `void destroy()`

  Destroy this EventServiceChannel instance.

- `java.util.Map<String, Object> getConfig()`

  Get the channel configuration that define the requested operating semantics of this instance.

- `java.lang.String getName()`

  Get the name of the channel(s) that this instance is consuming events from.

- `boolean isDestroyed()`

  Determine if this EventServiceChannel channel (not the correlator) is destroyed.

- `void registerEventType(EventType eventType)`

  Register an arbitrary EventType with this EventServiceChannel.

- `void removeEventListener(IEventListener eventListener)`

  Remove a previously registered IEventListener that was listening to all events received.

- `void removeEventListener()` - defined as:

  ```
  void removeEventListener(
      IEventListener eventListener,
      EventType eventType)
  ```

  Remove a previously registered IEventListener that was listening to all events received of the specified EventType.

- `IResponseWrapper requestResponse()` — defined as:

  ```
  IResponseWrapper requestResponse(
      Event requestEvent,
      EventType responseEventType)
  ```

  Emulate a synchronous RPC call to the correlator by sending an event and awaiting a matching response event (blocking call).

- `IResponseWrapper requestResponse()` — defined as:

  ```
  IResponseWrapper requestResponse(
      Event requestEvent,
      EventType responseEventType,
      long timeout)
  ```

  Emulate a synchronous RPC call to the correlator by sending an event and awaiting a matching response event (blocking call with timeout).

- `void setLateResponseListener(IEventListener eventListener)`

  Register the single IEventListener that will be notified of any “late” responses to either synchronous or asynchronous request-response calls.

- `void unregisterEventType(EventType eventType)`

Unregister an arbitrary EventType from this EventServiceChannel.

An `EventServiceChannel` object provides several fields that you can use to specify configuration options and default values for channel properties. These are described in the Javadoc documentation for the `IEventServiceChannel` interface.

The EventService API

The EventServiceFactory class

The `EventServiceFactory` class provides the following methods for creating instances of classes that implement the `IEventService` interface:

- `static IEventService createEventService()`

  Create a new instance of the `EventService` with default parameters.

- `static IEventService createEventService()` — defined as:

  ```
  static IEventService createEventService(
      EngineClientInterface engineClient)
  ```

  Create a new `EventService` instance using the supplied `EngineClient` to connect to a correlator.

- `static IEventService createEventService()` — defined as:

  ```
  static IEventService createEventService(
      java.lang.String socket_hostname,
      int socket_port)
  ```

  Create a new `EventService` instance with an `EngineClient` connected to a correlator on the given host and port.

The EventService API

Examples of use

This section contains sections of code that illustrate the EventService API. The complete sample applications are located in the Apama installation's `samples\engine_client\javaEventService` directory. The directory contains a README.txt file that describes how to build and run the sample applications including how to inject the necessary monitors.

The following piece of code creates an EventService object and then sends a simple event to the correlator (the correlator would need a corresponding monitor to listen for the `SimpleEvent` event type):

```
IEventService eventService = EventServiceFactory.createEventService();
public EventServiceSample() {
  try {
    // create the simpleEvent object
    Event simpleEvent = new Event("SimpleEvent(\"hi there\")");
    // send it
    eventService.sendEvent(simpleEvent);
    System.out.println("SimpleEvent(hi there) sent...");
    System.out.print("\t Please check Correlator console for the string ");
    System.out.print("\t *** SimpleEvent(\"hi there\") received ***");
  } catch (EngineException ee) {
    ee.printStackTrace(System.err);
  }
}
```

The next code sample illustrates how to create a named channel. Then it adds a listener to the channel. To illustrate how the listener works, the code sample sends an event to the correlator, which (assuming the corresponding monitor has been injected) sends a notification that the event was received. Finally, the sample shows the listener's callback method that is invoked when the notification is received.

```
public ESChannelSample() {
  // Add the channel to those monitored by the event service
  IEventServiceChannel ourChannel =
    eventService.addChannel("eventService.sample.channel", null);
  // Create an EventType to describe the event we are expecting to receive
  EventType channelEventType = new EventType(
    "ChannelledEvent", new Field[] {new Field("s", StringFieldType.TYPE)});
  // Add a listener that will be notified whenever an event of the
  // specified type is received on the channel
  ourChannel.addEventListener(new MyEventListener(), channelEventType);
  // Send an event to the correlator
  try {
    Event cEvent = new Event("ChannelledEvent(\"hi there\")");
    eventService.sendEvent(cEvent);
  } catch (EngineException ee) {
    ee.printStackTrace(System.err);
  }
}
// Inner class to implement our callback method that is notified when
// events are received.
private class MyEventListener extends EventListenerAdapter {
  public void handleEvent(Event event) {
    System.out.println("MyEventListener.handleEvent() called, event = " +
        event);
  }
}
```

The following code sample defines a request event type and a response event type and registers the request event type with the channel. An `IResponseWrapper` object is used to issue the request. Note that the `requestResponse()` method is a blocking call.

```
  // Define the RequestEvent and ResponseEvent event type
  // NOTE: in order to use the requestResponse() method, the Request
  //       and Response event must have a member  "messageId"
  //       (IEventServiceChannel.DEFAULT_MESSAGEID_FIELD_NAME) of
  //       MonitorScript type integer, Java type Long.
  EventType requestEventType = new EventType("RequestEvent",
      new Field[] {
        new Field(IEventServiceChannel.DEFAULT_MESSAGEID_FIELD_NAME,
        IntegerFieldType.TYPE),  // field required for using
                                 // requestResponse() call
        new Field("msg", StringFieldType.TYPE)});
  EventType responseEventType = new EventType("ResponseEvent",
      new Field[] {
        new Field(IEventServiceChannel.DEFAULT_MESSAGEID_FIELD_NAME,
        IntegerFieldType.TYPE),  // field required for using
                                 // requestResponse() call
        new Field("responseMsg", StringFieldType.TYPE)});
  try {
    // Register the RequestEventType to the channel
    ourChannel.registerEventType(requestEventType);
    // Create the requestEvent
    Event requestEvent = new Event("RequestEvent(0, \"Hi There\")");
    // Send two events in a for loop.  User can experiment
    // what will happen if responseWrapper.releaseLock() is NOT called
    // by commenting out the line :
    //     responseWrapper.releaseLock()
    // inside the finally block.
    for (int i = 0; i < 2; i++) {
      IResponseWrapper responseWrapper = null;
      try {
        // Now, make the requestReponse.
```

```
        // NOTE: requestResponse() is a blocking call and won't return until
        //       a response is received (or other failure condition ocurred)
        System.out.println("Sending RequestEvent() ...");
        responseWrapper =
           ourChannel.requestResponse(requestEvent, responseEventType);
        // print the response event
        System.out.println("Response event received: " +
           responseWrapper.getEvent());
      } finally {
        if (responseWrapper != null) {
          // NOTE: we must release the lock after a response is received
          responseWrapper.releaseLock();
        }
      }
    }
  } catch (Exception ee) {
    ee.printStackTrace(System.err);
  }
}
```

The next piece of sample code registers a ResponseListener with the channel. The code defines an asyncRequest event type and an asyncResponse event type and a callback method that is used to handle the response. Note, the `asynchRequestResponse()` method is non-blocking.

```
  // Define the AsyncRequestEvent and AsyncResponseEvent event type
  // NOTE: in order to use the asyncRequestResponse() method, the Request
  //       and Response event must have a member  "messageId"
  //       (IEventServiceChannel.DEFAULT_MESSAGEID_FIELD_NAME) of
  //       MonitorScript type integer, Java type Long
  EventType asyncRequestEventType = new EventType("AsyncRequestEvent",
      new Field[] {
        new Field(IEventServiceChannel.DEFAULT_MESSAGEID_FIELD_NAME,
        IntegerFieldType.TYPE),  // field required for using
                                 // asyncRequestResponse() call
        new Field("msg", StringFieldType.TYPE)});
  EventType asyncResponseEventType = new EventType("AsyncResponseEvent",
      new Field[] {
        new Field(IEventServiceChannel.DEFAULT_MESSAGEID_FIELD_NAME,
        IntegerFieldType.TYPE),  // field required for using
                                 // asyncRequestResponse() call
        new Field("responseMsg", StringFieldType.TYPE)});
  try {
    // Register the RequestEventType to the channel
    ourChannel.registerEventType(asyncRequestEventType);
    // Create the requestEvent
    Event requestEvent = new Event("AsyncRequestEvent(0, \"Hi There\")");
    // Now, make the asyncRequestReponse.  MyReponseListener's
    // handleResponse() is the callback method when a response is available
    ourChannel.asyncRequestResponse(new MyResponseListener(),
                    requestEvent,
                    asyncResponseEventType);
    System.out.println("AsyncRequestEvent() sent...");
  } catch (Exception ee) {
    ee.printStackTrace(System.err);
  }
}
/** Inner class to implement our callback method that is notified when events
    are recieved. */
private class MyResponseListener implements IResponseListener {
  /** Callback method that is called when a matching response Event is
   * received for an asynchronous request-response call made on an
   * EventServiceChannel.
   *
   * @param requestEvent the original request Event.
   * @param responseEvent the corresponding response Event.
   */
  public void handleResponse(Event requestEvent, Event responseEvent) {
    System.out.println("MyResponseListener.handleResponse() called - ");
    System.out.println("\t requestEvent = " + requestEvent);
    System.out.println("\t responseEvent = " + responseEvent);
```

```
  }
  /** Callback method that is called when an exception is thrown within an
   * asynchronous request-response call made on an EventServiceChannel.
   *
   * @param exception The exception that was thrown in the asynchronous call.
   */
  public void handleException(Exception exception) {
    exception.printStackTrace(System.err);
  }
}
```

The EventService API

Chapter 7: The ScenarioService API

The Apama ScenarioService application programming interface (API) is layered on top of the Apama EventService API described in "The EventService API" on page 63. The ScenarioService API provides an external interface to scenarios built with the Event Modeler that are running in a correlator. It provides an interface to instances of those scenarios as well.

The key elements

The ScenarioService API consists of three interfaces, `IScenarioService`, `IScenarioDefinition`, and `IScenarioInstance`; the `ScenarioServiceFactory` class; and the `ScenarioServiceConfig` class. These are included in the package `com.apama.services.scenario`.

The `IScenarioService` interface is used to establish communication with the event correlator and to provide access to the scenario definitions in the correlator.

The `IScenarioDefinition` interface is used to represent a scenario running in the correlator (not an instance). The interface provides methods to obtain meta-information, such as parameter names, types, and constraints, about the scenarios in the correlator. The interface has methods for adding and removing listeners. It also has methods for accessing all instances of a scenario as well as for creating new instances of it.

The `IScenarioInstance` interface is used to represent a single instance of a scenario. The interface has methods to access the values of the instance's parameters as well as to delete the instance. The interface also has methods to add and remove listeners.

The ScenarioServiceFactory class provides three methods for creating new instances of classes that implement the `IScenarioService` interface.

The `ScenarioServiceConfig` class is a helper class for building a properties map when using the `ScenarioServiceFactory` to create a new ScenarioService

The complete reference guide for the ScenarioService API is available in HTML format in the Apama installation's `doc\javadoc` directory.

A selection of sample applications that use the ScenarioService API is available in the Apama installation's `samples\engine_client\javaScenarioService` directory.

The ScenarioService API

The IScenarioService interface

To use the ScenarioService in its basic form, an application needs to create an object that implements the `IScenarioService` interface by using the `ScenarioServiceFactory` method `createScenarioService()`. The recommended usage is to pass a listener to the factory method so that the application is notified of all scenarios in the correlator as they are discovered. If a listener is not passed directly to the factory, the application must perform the following steps in this precise order to guarantee it sees all scenarios:

1. Obtain an instance of the service via one of the static methods of `ScenarioServiceFactory`.
2. Add one or more listeners so that the application will be notified of new scenarios as they are added.
3. The application should call either `getScenarios()`, `getScenarioIds()`, or `getScenarioNames()` to discover any scenarios that the service discovered before the application listener was added.

Application listeners receive `PropertyChangeEvent`s from this interface. Notifications are available when scenarios are added or removed from the correlator, when a ScenarioService support monitor is unloaded from the correlator, and when the status of the scenario discovery mechanism is changed. For more information on `PropertyChangeEvent`s, refer to the Javadoc documentation.

Classes that implement the `IScenarioService` interface have the following methods:

- `void addListener(java.beans.PropertyChangeListener listener)`

  Add a `PropertyChangeListener` that will be notified of changes to any bound property of this object.

- `void addListener()` — defined as:

  ```
  void addListener(
      java.lang.String propertyName,
      java.beans.PropertyChangeListener listener)
  ```

  Add a `PropertyChangeListener` that will be notified of changes to a specific named bound property of this object.

- `void destroy()`

  Destroy this service and clean up resources.

- `java.util.Map<String, Object> getConfig()`

  Get the configuration properties that define the requested operating semantics of this instance of the service.

- `DiscoveryStatusEnum getDiscoveryStatus()`

  Get the status of the internal scenario discovery process.

- `IEventService getEventService()`

  Get the underlying EventService that is being used by this ScenarioService.

- `IScenarioDefinition getScenarioById(java.lang.String scenarioId)`

  Get a `ScenarioDefinition` for a specific scenario, using the ScenarioId as the lookup key.

- `IScenarioDefinition getScenarioByName(java.lang.String scenarioDisplayName)`

  Get a `ScenarioDefinition` for a specific scenario, using the Display Name as the lookup key.

- `java.util.Set<Long> getScenarioIds()`

  Get the IDs of all known scenarios (not instances) in the correlator.

- `java.util.Set<String> getScenarioNames()`

  Get the Display Names of all known scenarios (not instances) in the correlator.

- `java.util.List<IScenarioDefinition> getScenarios()`

  Get the ScenarioDefinitions of all known scenarios in the correlator.

- `boolean isDestroyed()`

  Determine if this service is destroyed.

- `void removeListener(java.beans.PropertyChangeListener listener)`

  Remove a `PropertyChangeListener` that was to be notified of changes to any bound property of this object.

- `void removeListener()` — defined as:

  ```
  void removeListener(
  java.lang.String propertyName,
  java.beans.PropertyChangeListener listener)
  ```

  Remove a `PropertyChangeListener` that was to be notified of changes to a specific named bound property of this object.

The ScenarioService API

The ScenarioDefinition interface

A class that implements the `IScenarioDefinition` interface represents a scenario (created with the Event Modeler) that is running in the correlator. ScenarioDefinition objects are returned from calls to the following ScenarioService methods:

- `getScenarios()`
- `getScenarioByName()`
- `getScenarioByID()`

With a ScenarioDefinition object, you can add and remove listeners, create new instances, get specific instances, and get meta information about the scenario, such as the scenario's description, display name, and input and output parameter names and types.

The complete set of methods for classes that implement the `IScenarioDefinition` interface includes the following:

- `void addListener(java.beans.PropertyChangeListener listener)`

  Add a `PropertyChangeListener` that will be notified of changes to any bound property of this object.

- `void addListener()` — defined as:

  ```
  void addListener(
      java.lang.String propertyName,
      java.beans.PropertyChangeListener listener)
  ```

Add a `PropertyChangeListener` that will be notified of changes to a specific named bound property of this object.

- `IScenarioInstance createInstance(java.lang.String owner)`

  Create a new instance of this scenario, using default values for all input parameters.

- `IScenarioInstance createInstance()` — defined as:

```
IScenarioInstance createInstance(
    java.lang.String owner,
    java.util.Map<String, Object> inputParameters)
```

  Create a new instance of this scenario, with the supplied values for input parameters.

- `IScenarioInstance createInstance()` — defined as:

```
IScenarioInstance createInstance(
    java.lang.String owner,
    java.util.Map<String, Object> inputParameters,
    java.beans.PropertyChangeListener listener)
```

  Create a new instance of this scenario, with the supplied values for input parameters, and atomically add a listener specified by *listener*.

- `IScenarioInstance createInstance()` — defined as:

```
IScenarioInstance createInstance(
    java.lang.String owner,
    java.util.Map<String, Object> inputParameters,
    java.lang.String property,
    java.beans.PropertyChangeListener listener)
```

  Create a new instance of this scenario, with the supplied values for input parameters, and atomically add a listener. The property that the listener listens to is specified by the *property* argument.

- `java.lang.String getDescription()`

  Get the description of the scenario.

- `DiscoveryStatusEnum getDiscoveryStatus()`

  Get the status of the internal scenario instance discovery process.

- `java.lang.String getDisplayName()`

  Get the display name of the scenario.

- `java.lang.String getId()`

  Get the ID of the scenario.

- `java.util.List<String> getInputParameterConstraints()`

  Get the constraints of all input parameters for the scenario.

- `java.util.List<Object> getInputParameterDefaults()`

  Get the default values of all input parameters for the scenario.

- `java.util.Set<String> getInputParameterNames()`

  Get the names of all input parameters for the scenario.

- `java.util.List<ParameterTypeEnum> getInputParameterTypes()`

Get the types of all input parameters for the scenario.

- `IScenarioInstance getInstance(long instanceId)`

  Get a single specific instance of this scenario, by instance ID.

- `java.util.List<IScenarioInstance> getInstances()`

  Get all instances of this scenario.

- `java.util.List<IScenarioInstance> getInstancesForOwner(java.lang.String owner)`

  Get all instances of this scenario for a specific owner.

- `java.util.Set<String> getOutputParameterNames()`

  Get the names of all output parameters for the scenario.

- `java.util.List<ParmeterTypeEnum> getOutputParameterTypes()`

  Get the types of all output parameters for the scenario.

- `boolean isInputParameter(java.lang.String parameterName)`

  Test if a specific named parameter is an input parameter.

- `boolean isOutputParameter(java.lang.String parameterName)`

  Test if a specific named parameter is an output parameter.

- `void removeListener(java.beans.PropertyChangeListener listener)`

  Remove a `PropertyChangeListener` that was to be notified of changes to any bound property of this object.

- `void removeListener()` — defined as:

  ```
  void removeListener(
      java.lang.String propertyName,
      java.beans.PropertyChangeListener listener)
  ```

  Remove a `PropertyChangeListener` that was to be notified of changes to a specific named bound property of this object.

When an application listener receives a `PropertyChangeEvent` from this interface, the event specifies the type of change. Notifications are available when ScenarioInstances have been added, edited, updated, removed, died, changed state or when the instance discovery mechanism for that particular definition changes. For more information on `PropertyChangeEvent`s, refer to the Javadoc documentation.

The ScenarioService API

The IScenarioInstance interface

A class that implements the `IScenarioInstance` class represents an instance of a scenario. The class has methods to query and change the value of any of its parameters. It also contains methods to add and remove listeners and a method to delete the instance. The full set of methods is as follows:

- `void addListener(java.beans.PropertyChangeListener listener)`

Add a listener to the list of those interested in changes to any object property, or any scenario instance (input and/or output) parameter.

- `void addListener()` - defined as:

```
void addListener(
    java.lang.String propertyName,
    java.beans.PropertyChangeListener listener)
```

Add a listener to the list of those interested in changes to the object property with given name, or the scenario instance (input and/or output) parameter with the given name.

- `boolean delete()`

Delete this instance.

- `java.lang.Long getId()`

Get the ID of the instance.

- `long getLastUpdateTime()`

Get the timestamp (milliseconds) of the last known update event for this instance.

- `java.lang.String getOwner()`

Get the owner (username) of the instance.

- `IScenarioDefinition getScenarioDefinition()`

Get the scenario definition.

- `InstanceStateEnum getState()`

Get the current state of the instance.

- `java.lang.Object getValue(java.lang.String parameterName)`

Get the value of a single specific (input or output) parameter of this instance.

- `void removeListener(java.beans.PropertyChangeListener listener)`

Remove a listener from the list of those interested in changes to any object property, or scenario instance parameter.

- `void removeListener()` — defined as:

```
void removeListener(
    java.lang.String propertyName,
    java.beans.PropertyChangeListener listener)
```

Remove a listener from the list of those interested in changes to the object property with given name, or the scenario instance parameter with the given name.

- `boolean setValue()` — defined as:

```
boolean setValue(
    java.lang.String parameterName,
    java.lang.Object value)
```

Set the value of a single specific (input) parameter of this instance.

- `boolean setValues(java.util.Map<String, Object> valuesMap)`

Set several (input) parameter values of this instance.

The ScenarioService API

The ScenarioServiceFactory class

The `ScenarioServiceFactory` class provides factory methods for creating new instances of classes that implement the `IScenarioService` interface.

There are three variants of the factory methods:

- Client supplies correlator host and port;
- Client supplies an initialized `EngineClientInterface` (usually an `EngineClientBean`);
- Client supplies an initialized `IEventService`

The factory methods are:

- `static IScenarioService createScenarioService()` — defined as:

  ```
  static IScenarioService createScenarioService(
      EngineClientInterface engineClient,
      java.util.Map<String, Object> scenarioServiceConfig,
      java.beans.PropertyChangeListener listener)
  ```

  Create a new `ScenarioService` instance using the supplied `EngineClient` to connect to a correlator, and optionally pass a listener that will be added before any events are received.

- `static IScenarioService createScenarioService()` — defined as:

  ```
  static IScenarioService createScenarioService(
      IEventService eventService,
      java.util.Map<String, Object> scenarioServiceConfig,
      java.beans.PropertyChangeListener listener)
  ```

  Create a new `ScenarioService` instance using the supplied `IEventService` to connect to a correlator, and optionally pass a listener that will be added before any events are received.

- `static IScenarioService createScenarioService()` — defined as:

  ```
  static IScenarioService createScenarioService(
      java.lang.String socket_hostname,
      int socket_port,
      java.util.Map<String, Object> scenarioServiceConfig,
      java.beans.PropertyChangeListener listener)
  ```

  Create a new `ScenarioService` instance with an `EngineClient` connected to a correlator on the given host and port, and optionally pass a listener that will be added before any events are received.

For each of the above factory methods, the client must also supply parameters for a configuration map, and a listener. Either of those additional parameters may be null. For more details of the configuration parameter, see the Javadoc documentation for `IScenarioService`, and `ScenarioServiceConfig`. The names of the bound properties for which the listener will be notified are those properties of `IScenarioService` whose names begin "`PROPERTY_`", for example, `IScenarioService.PROPERTY_SCENARIO_ADDED`.

The ScenarioService API

The ScenarioServiceConfig class

The `ScenarioServiceConfig` class is a helper class that is used to build a properties map used by the `ScenarioSeviceFactory` when creating a new `ScenarioService`.

- `static void setAckDataTimeout(java.util.Map<String, Object> properties, long timeout)`

  Set the timeout for operations to wait for the Data channel to catch up.

- `static void setScenarioExclusionFilter()` — defined as:

  ```
  static void setScenarioExclusionFilter(
      java.util.Map<String, Object> properties,
      java.util.Set scenarioExclusionFilter)
  ```

  Set the configuration property to the set of `ScenarioID`s that the client wishes to ignore.

- `static void setScenarioInclusionFilter()` — defined as:

  ```
  static void setScenarioInclusionFilter(
      java.util.Map<String, Object> properties,
      java.util.Set scenarioInclusionFilter)
  ```

  Set the configuration property to the set of `ScenarioID`s in which the client is interested.

- `static void setAutoInstanceDiscovery()` — defined as:

  ```
  static void setAutoInstanceDiscovery(
      java.util.Map<java.lang.String,
      java.lang.Object> properties,
      boolean autoInstanceDiscovery)
  ```

  Set the boolean configuration property to indicate if the service should discover instances automatically.

  This disables the default behavior of discovering all scenario instances upon creating the scenario service and can be used as an alternative to the scenario inclusion/ exclusion filter as it allows discovering scenario definitions without necessarily having to discover all of their instances. Typically, a client will then need to call `requestInstances` on a `IScenarioDefinition` and wait for the discovery state to become `COMPLETED` before it can interrogate the instances of a scenario (the call to `requestInstances` is automatic if this property is not set).

- `static void setStrongDataInboundEventQueue()` — defined as:

  ```
  static void setStrongDataInboundEventQueue(
      java.util.Map<String, Object> properties,
      boolean strong)
  ```

  Set the boolean configuration property to indicate if the *Data* channels (`EventServiceChannel`s), created for listening to scenario instance output variable updates, should use strong or soft references to events in the inbound event queue.

- `static void setUseRawDataChannel()` — defined as:

  ```
      static void setUseRawDataChannel(
      java.util.Map<String, Object> properties,
      boolean useRawData)
  ```

  Set the boolean configuration property to indicate if the *Data* channels (`EventServiceChannel`s), created for listening to scenario instance output variable updates, should be subscribed to the normal Data channel (possibly throttled), or the Raw Data channel (always unthrottled).

- `static void setUsernameFilter()` — defined as:

  ```
  static void setUsernameFilter(
      java.util.Map<java.lang.String,
      java.lang.Object> properties,
      java.lang.String owner)
  ```

Set the configuration property to filter by a given user. Scenario service clients have the ability to listen on a per-user channel for updates from a scenario. This can significantly reduce the number of updates the client needs to discard if it is only concerned with one user's scenarios and/or DataViews.

If the `setUsernameFilter()` method is used, the scenario(s) listened for must be configured to send updates on the per-user channels — this is done by sending a `com.apama.scenario.ConfigureUpdates` event with key `sendThrottledUser` or `sendRawUser` set to true, either for all scenarios or just the scenario(s) that the client is interacting with.

Alternatively, when starting, clients can set the Java system property `com.apama.scenario.filterUser` to specify what user the ScenarioService listens to updates for (currently, one ScenarioService is only able to listen to updates for one user).

The user "`*`" is handled specially — all clients will receive updates for the user "`*`", even if filtering for a specific user.

The ScenarioService API

Examples of use

This section contains sections of code that illustrate the ScenarioService API. The complete sample applications are located in the Apama installation's `samples\engine_client\javaScenarioService` directory. The directory contains a README.txt file that describes how to build and run the sample applications including how to inject the necessary monitors. The samples are:

- `SimpleScenarioService.java` — Sample 1, demonstrates the capabilites of the ScenarioService API.
- `SimpleScenarioDefinition.java` — Sample 2, demonstrates the capabilities of the ScenarioDefinition class.
- `SimpleScenarioInstance.java` — Sample 3, demonstrates how to manipulate the ScenarioInstance class.
- `FilteredScenarioInstance.java` — Sample 4, extends Sample 3 to demonstrate how to filter a list of interested `ScenarioId`s.

The following piece of code shows the recommended method of creating a ScenarioService object. First, it creates a ScenarioServiceListener and then uses the ScenarioServiceFactory method to create a new ScenarioService instance, passing in the listener as the forth argument.

```
// Create a listener for the ScenarioService
scenarioServiceListener = new ScenarioServiceListener();
// Get a IScenarioService instance from the ScenarioServiceFactory
scenarioService =  ScenarioServiceFactory.createScenarioService(
    "localhost",
    ConnectionConstants.DEFAULT_ENGINE_PORT,
    null,
    scenarioServiceListener);
```

The following piece of code is an example of how to provide a listener for handling the PropertyChangeEvent events fired by the IScenarioService object.

```
private class ScenarioServiceListener implements PropertyChangeListener {
  public void propertyChange(PropertyChangeEvent evt) {
    if (!IScenarioService.PROPERTY_SCENARIO_ADDED.equals(
        evt.getPropertyName())) {
      // Example only cares about ADDED events and discards others
      return;
    }
```

```
    IScenarioDefinition def = (IScenarioDefinition)evt.getNewValue();
    // check if the event signifies the desired Scenario ID
    if (null!=def && SCENARIO_ID.equals(def.getId())) {
      // Now go do something useful with it...
      createEditDelete(def);
    }
  }
}
```

The ScenarioService API

Chapter 8: The .NET Engine Client Library

Apama includes a .NET assembly that wraps the Apama Engine Client library. This makes it possible to perform tasks such as the following from an assembly written in any .NET language:

- Send and receive events
- Inject or delete EPL including adding and removing monitors
- Add and remove listeners
- Issue requests to the correlator
- Invoke callback methods when responses to requests are received from the correlator
- Query a running event correlator for status information
- Manage a set of named channels
- Access scenario definitions in the correlator
- Obtain meta-information from scenario definitions in the correlator, such as parameter names, types, and constraints
- Access scenario instances in the correlator and create new instances
- Access values of parameters in scenario instances in the correlator
- Inject Correlator Deployment Packages

Reference information for the .NET client API is available here:

install_dir\doc\dotNet\engine_client_dotnet5.2.chm

Since the .NET client library provides the same features as the Apama client library for Java, you can also consult the following:

- "The JavaBeans API" on page 54
- "The EventService API" on page 63
- "The ScenarioService API" on page 70

This section provides the following information:

- "Using the .NET client library" on page 80
- "Java and .NET namespace/class mapping" on page 81

Using the .NET client library

To make use of the .NET wrapper, add the `engine_client_dotnet5.2.dll` library as a reference of your assembly.

To run an application using the wrapper:

1. Copy the following libraries in the directory that contains the compiled .NET assembly that uses them:
   - `engine_client5.2.dll`
   - `engine_client_dotnet5.2.dll`
2. Ensure that the Apama installation's `bin` directory is in the `PATH` environment variable.

To use the `log4net` implementation of the logging interface, copy `log4net.dll` as well.

For examples of using the various supported APIs, see the `samples\engine_client\DotNetSamples` directory. This directory includes several samples, all based on the Java API samples.

Both the 32-bit and 64-bit versions of the `engine_client_dotnet5.2.dll` library are built to run on the .NET 4 runtime and are tested against the 4.5.1 libraries.

The following .NET API layers use the `AppDomain.ProcessExit` event handler to run cleanup operations upon exiting the application:

- `EngineClient`
- `EventService`
- `ScenarioService`

As described in http://msdn.microsoft.com/en-us/library/system.appdomain.processexit.aspx, the total execution time of all `ProcessExit` event handlers is limited, just as the total execution time of all finalizers is limited at process shutdown. The default is three seconds.

The .NET Engine Client Library

Java and .NET namespace/class mapping

The following table compares the Apama client APIs for Java with the Apama .NET client APIs. Package names appear in bold. Class names appear in plain text. For a given plain text class name, the class's full name is the package name under which it appears followed by the plain text class name.

Table 1. Java and .NET namespace/class mapping

Java	.NET
com.apama	**Apama**
`EngineException`	`EngineException`
`InterruptedEngineException`	`(no equivalent)`
`util.LogLevel`	`LogLevel`

Java	.NET
com.apama.engine	**Apama.Engine**
EngineConnection	EngineConnection
EngineInfo	EngineInfo
EngineManagement	EngineManagement
EngineManagementFactory	Api
EngineStatus	EngineStatus
MonitorScript	MonitorScript
NameInfo	NameInfo
NamedContainerTypeInfo	NamedContainerTypeInfo
NamedContextInfo	NamedContextInfo
NamedEventTypeInfo	NamedEventTypeInfo
NamedJavaApplicationInfo	NamedJavaApplicationInfo
NamedMonitorInfo	NamedMonitorInfo
com.apama.engine.beans	**Apama.Engine.Client**
EngineClientBean	EngineClientFactory, IEngineClient
com.apama.engine.beans.interfaces	**Apama.Engine.Client**
ConnectOperationsInterface	IConnectOperations
ConsumerOperationsInterface	IConsumerOperations
DeleteOperationsInterface	(in IMessagingClient)
EngineClientInterface	IEngineClient
InjectOperationsInterface	(in IMessagingClient)
InspectOperationsInterface	(in ICorrelatorManagement)
ReceiveConsumerOperationsInterface	(in IMessagingClient)
ReceiveOperationsInterface	(in IMessagingClient)
SendOperationsInterface	(in IMessagingClient)

Java	.NET
WatchOperationsInterface	(in ICorrelatorManagement)
com.apama.event	**Apama.Event**
DisconnectableEventConsumer	DisconnectableEventConsumer
EventConsumer	EventConsumer
EventSupplier	EventSupplier
IEventListener	IEventListener
com.apama.event.parser	**Apama.Event.Parser**
BooleanFieldType	BooleanFieldType
DecimalFieldType	DecimalFieldType
DecimalFieldValue	DecimalFieldValue
DictionaryFieldType	DictionaryFieldType
EventParser	EventParser
EventType	EventType
Field	Field
FieldType	FieldType
FieldTypeFactory	FieldTypeFactory
FloatFieldType	FloatFieldType
IntegerFieldType	IntegerFieldType
LocationType	LocationType
LocationFieldType	LocationFieldType
SequenceFieldType	SequenceFieldType
StringFieldType	StringFieldType
com.apama.services.event	**Apama.Services.Event**
ChannelConfig	ChannelConfig
CommunicationException	CommunicationException

Java	.NET
EventServiceException	EventServiceException
EventServiceFactory	EventServiceFactory
IEventService	IEventService
IEventServiceChannel	IEventServiceChannel
IResponseListener	IResponseListener
IResponseWrapper	IResponseWrapper
ResponseTimeoutException	ResponseTimeoutException
com.apama.services.scenario	**Apama.Services.Scenario**
DiscoveryStatusEnum	DiscoveryStatus
IScenarioDefinition	IScenarioDefinition
IScenarioInstance	IScenarioInstance
IScenarioService	IScenarioService
IllegalCallingThreadException	IllegalCallingThreadException
InstanceStateEnum	InstanceState
InvalidInputParameterException	InvalidInputParameterException
ParameterTypeEnum	ParameterType
ScenarioServiceConfig	ScenarioServiceConfig
ScenarioServiceException	ScenarioServiceException
ScenarioServiceFactory	ScenarioServiceFactory
com.apama.util	**Apama.Util**
Logger	Logger (+ ILogger)
TimestampSet	(no equivalent)
TimeStampSetException	(no equivalent)

The .NET Engine Client Library